Design and Implementation of a Chess-Playing Program in the Java Programming Language

François Dominic Laramée

A Major Report
in
The School
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2001

ABSTRACT

Design and Implementation of a Chess-Playing Program in the Java Programming Language

François Dominic Laramée

This project describes the design and implementation of an object-oriented chess-playing program, based on current software engineering practice, recent advances, including the MTD(f) search algorithm, and time-honored techniques perfected by artificial intelligence pioneers since the late 1960's, like the transposition table, the history heuristic and an evaluation function slanted towards material advantage.

Where appropriate, a comparative survey of alternative algorithms not implemented in this project is also included.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Preamble

JavaChess is a text-only chess-playing program written in Java. Based on an extensive survey of the literature on chess software, this game includes features patterned after several of the most successful programs of all time, including *Chess 4.5* (Slate and Atkin [1983]), *Belle* (Condon and Thompson [1981]) and *Cray Blitz* (Hyatt [1983]), as well as some recent innovations (notably in the search algorithm).

Whenever the two are incompatible, the JavaChess architecture favors ease of understanding and extensibility over pure performance. The justification for this decision is threefold:

- The project is open-source. Its code was released as part of a six-part article series published on gamedev.net, a web community for professional and hobbyist game developers; a re-tooled version of the series will appear in print in 2001, as a major chapter of Magma Publishing's ***Game Programming HOWTO*** book.

- The program will be extended into a graphical application and used as a test-bed for new ideas and new games in the future.

- The author has no intention of ever entering the program in any type of competition, so that pure speed is of little consequence.

The project is limited in scope to the development of a sound, reasonably effective chess-playing engine, along with the necessary "harness" software required to play with it. No specific goals have been set in terms of the program's strength (in comparison to other software packages or human players) or suitability for tournament play.

JavaChess was developed on a Pentium II 333 MHz PC equipped with 64 megabytes of RAM and running under Windows 98, and compiled using the Borland Jbuilder 3.5 development environment.

The preliminary literature search, performed over a period of approximately eight months between October 1999 and May 2000, required approximately 25 days of work. Application design, development and documentation (including the present document and the aforementioned article series and book chapter) took 67 days, which is about 10% less than initially estimated.

1.2 Chess Terminology And Notation

The following chess-specific terms and expressions are used in this report.

- File: A column on the chess board. Files are numbered from left to right. For example: the initial positions of the two kings are located at opposite ends of the fifth file.

- Rank: A row on the chess board. Rows are numbered from bottom to top. For example, a player's pawns begin the game on the second rank, while his opponent's begin on the seventh rank.

- Minor Piece: A knight or a bishop.

- Major Piece: A rook or a queen.

- Pawn Ram: Two pawns of opposite colors located on adjacent squares in the same file, where they block each other's forward movement.

- Doubled Pawns: Two or more pawns of the same color on a single file.

- Isolated Pawn: A pawn without friendly pawns on either adjacent file.

- Passed Pawn: A pawn which has advanced beyond the positions of any opposing pawns on its own file or either adjacent files.

- Open File: A file without any pawns on it.

- Half-Open File: From a player's perspective, a file without any friendly pawns on it.

- Chess Notation: By convention, chess boards are usually printed with White's starting position at the bottom. Files are identified by the letters A to H, starting at the left side of the board (or White's Queen's Rook's file). Ranks are numbered 1 through 8 beginning with White's back rank. Also

by convention, squares are identified by file and then by rank; for example, White's King begins the game on E1.

- Opening: The first part of the game, where most of the pieces are still on the board. Chess literature often defines the opening as the phase of the game for which there exists recorded analysis of the best moves to play from a position.

- Middle Game: The phase of the game between the opening and the endgame. The middle game is characterized by the presence of numerous pieces on the board and by the absence of systematic and encyclopedic knowledge about the moves to play from a position.

- Endgame: The last phase of the game, during which few pieces remain on the board. During the endgame, the king typically becomes an offensive piece. Thompson [1986] has used retrograde analysis to build complete databases of the best moves to play from every position in several simple endgames, including king and rook vs king.

1.3 Game Trees and Chess Programming

For purposes of mathematical analysis and programming, games are commonly represented as trees (Shinghal [1992]). Each of the tree's *nodes* represents a "state" of the game. In chess, a state is defined by the positions of the pieces on the board, the identity of the current player and castling rights. The tree's *root*

represents the state of the game prior to the search. Each *edge* represents a move performed by one of the players. The *depth* of a node is the distance between the node and the root, measured in number of edges. The term ply refers to one move by a player. Thus, a node at depth 4 corresponds to 4 plies (or 4-ply for short) and represents a state resulting from two moves by each player.

When a program has to select a move, it builds the game tree, searches each path to a pre-determined length, determines how advantageous the resulting position is by applying an *evaluation function*, and picks the move leading to the most favorable outcome. By convention, a position in which the moving player obtains checkmate is valued at "plus infinity", while one in which he is checkmated is worth "minus infinity". A draw is worth zero.

A *leaf* is a node with no outgoing edge. Ideally, we would like each leaf to represent a position where the game is over and a winner (or a draw) can be determined with certainty; in practice, this would be too time-consuming, and the program must terminate a search along a path as soon as it reaches a node where a likely outcome can be assessed.

Unfortunately, the size of the game tree grows exponentially with depth. If the average number of moves available to a player in a given position, also known as the branching factor, is b, and the tree is searched to depth n, the cost of the search is $O(b^n)$. Since b is about 35 during the middle game in chess, searching

every path is expensive. Therefore, techniques which reduce the total number of nodes to visit or the amount of work performed at each visited node must be implemented.

The number of nodes to be visited during search can be reduced using a number of *pruning* techniques. In the 1960's, several programs used *selective forward pruning* to eliminate some of a node's children from consideration without searching them at all by using heuristic evaluation techniques to determine which were the most and least promising. Unfortunately, such heuristics risk eliminating strong moves from consideration because their consequences would only become evident at a deeper level in the search tree.

Contemporary programs have abandoned this approach in favor of *full-width search*, in which every node is examined until proven worse than a previously searched alternative. In chess, most positions present the player with a small number of strong moves and several poor alternatives. The alphabeta search algorithm (Shinghal[1992]) and its variants identify such weak moves after a partial search of the sub-trees rooted at their nodes. A *refutation* is a move which proves that the opponent's previous move was a mistake; for example, in many situations, capturing the opponent's queen would refute a move which leaves the queen in danger. Discovery of a refutation begets a *cutoff* of the search sub-tree, which stops the search along a branch before all of its descendants have been examined.

Since cutoffs reduce the effort required by a search without compromising its quality, algorithms seek to generate as many of them as possible. Provided that a strong move is searched early, a condition which can be met if an effective node ordering technique is employed, alphabeta can reduce the total number of nodes visited during a search to approximately twice the square root of the total size of the game tree (Shinghal [1992]). This document will discuss standard alphabeta, iterative-deepening alphabeta and the MTD(f) search algorithm.

More gains can be made by noticing that branches of the game tree can *transpose* into identical positions after different sequences of moves. Since a position's evaluation does not depend on the path taken to reach it, storing the results of a search in a *transposition table* will eliminate the need for duplicate searches of identical positions.

Finally, it has been demonstrated that not all chess positions can be evaluated properly. Since evaluation functions in most computer programs favor *material advantage*, in which one player has captured more pieces than his opponent, positions where the material balance is likely to change in the near future can not be assessed properly. *Quiescent positions,* in which the evaluation function is deemed unlikely to undergo drastic changes in the near future, must be sought to ensure proper evaluations. Determining which positions are quiescent is a difficult problem, and the effectiveness of this analysis has crucial impact on the search efficiency.

1.4 Chess Programming Data Structures

The following data structures employed in chess programs are discussed or mentioned in the text:

- Opening Book: A repertoire of positions and moves which can be made immediately if these positions are encountered, without search or analysis. The purpose of an opening book is to simulate a strong player's knowledge of opening theory; its contents are usually drawn from chess literature, for example the Encyclopedia of Chess Openings.

- Endgame Database: Similar in structure and purpose to the opening book, but for positions in the endgame. An endgame database ensures that the program will always play flawlessly in any position covered by the database.

- Transposition Table: A large array of positions which have recently been searched by the program and the results thereof. The transposition table is used to avoid duplicating search efforts when two or more branches lead to identical positions.

- History Table: A repository of information about the moves which have recently been effective at generating cutoffs during the search. The argument justifying the history table is that many chess positions are similar in overall concept, and that moves which generated cutoffs in the

recent past are likely to do so again in the near future. The history table is used to order a node's children prior to a search, in the hope that a cutoff will be generated quickly.

1.5 Summary

This document describes the development of a chess-playing program in Java. Based on the architecture of the successful Northwestern University chess program and on results obtained by the developers of several other programs of the past, Javachess implements an advanced search algorithm known as MTD(f), as well as a transposition table and a history table to accelerate the search. Chapter 2 of this report surveys the program's components. Chapter 3 describes the techniques used to represent the chess board. Chapter 4 discusses move generation. Chapter 5 covers auxiliary data structures, such as the history and transposition tables. Chapter 6 describes the evaluation function used to assess advantage in a chess position. Chapter 7 discusses search techniques. Finally, chapters 8 and 9 describe Javachess' architecture and behavior.

Experimentation with the program has demonstrated its strength in endgame situations and in solving chess problems. However, its speed and memory management are insufficient to guarantee good results in the middle game.

CHAPTER 2: PROGRAM COMPONENTS

Any chess-playing program requires certain software components (Welch [1984]). At the very least, these include:

- Some way to represent the current state of the chess board in memory.
- A move generator, which identifies the legal moves given a board position. This is required to ensure that the computer will play lawfully and be able to verify that its human opponent does so, as well.
- A search algorithm which examines legal moves and their consequences in turn.
- A position evaluation function, which allows the computer to assess the relative strengths of the various possible states of the game resulting from several variations and allows it to pick a sensible move during a search.
- A screen interface allowing the user to enter moves and examine the board.

This project adds a number of auxiliary data structures which enhance the computer's playing ability by accelerating move generation (the pre-processed move database), avoiding costly duplication of search effort (the transposition table) and ordering moves for optimal search efficiency (the history table).

However, since the current user interface is a mere test harness, intended to be replaced by a graphical application at a later date, little effort has been expended to make it ergonomically satisfying.

CHAPTER 3: BOARD REPRESENTATION TECHNIQUES

3.1 Early Efforts

In the early days of chess programming, memory was expensive. The utmost efficiency was required, as some of the pioneering programs (especially those running on early personal computers) had to make do with 8K of main memory or even less (Welch [1984]).

Given these limitations, it is hardly surprising that early programmers adopted a straightforward scheme to represent their boards internally: a 64-byte (or 32-byte) array, where each byte (or 4-bit nibble) represents a single square on the board and contains an integer constant representing the piece located in that square. (Any chess board representation also needs a few bytes of storage to track down *en passant* pawn capture opportunities and castling privileges.) For example, an empty square was allocated value 0, a black king could be represented by the number 1, etc.

A few refinements on this technique, cited by Welch [1984], soon became popular:

- The original SARGON extended the 64-byte array by surrounding it with two layers of sentinel squares containing values marking them as illegal. This trick accelerated move generation: for example, a bishop would generate moves by sliding one square at a time until it reached an illegal

square, then stop. (The second layer of protection is required by knight moves: for example, a knight sitting on a corner might attempt to jump two squares out of the board.)

- MYCHESS reversed the process and represented the board in only 32 bytes, each of which was associated with a single piece (i.e., the white king, the black King's Knight's pawn, etc.) and contained the number of the square where that piece was located, or a sentinel value if the piece had been captured. This technique had a serious drawback: it was impossible to promote a pawn to a piece which had not already been captured. Later versions of the program fixed this problem.

3.2 Bitboards

Slate and Atkin [1983] credit the KAISSA team from the Soviet Union with the invention of the bit board in the late 1960's. The bit board is a 64-bit word containing boolean information about a particular aspect of the game state, at a rate of 1 bit per square.

For example, a bitboard might contain the answer to "Is there a white piece here?" for each square of the board, while others might implement "the set of squares to which a queen on e3 can move", or "the set of white pieces currently attacked by black knights". A set of 12 bitboards, one each for the presence of

white pawns, white rooks, black pawns, etc., would therefore be sufficient to represent the whole chess board in this binary fashion.

While a bit board representation consumes more memory than the square array presented earlier, its usefulness lies in the fact that, with 64-bit bitboards and a 64-bit processor, a number of interesting chess operations can be implemented as a short sequence of (often single-cycle) logical operations. Most bitboard-based programs maintain a database of all positions which a certain piece located on a certain square can move to; as a result, logical operations can implement move generation in the fastest possible way.

For example, verifying whether the white queen is checking the black king would require the following expensive computation in a square-array program:

- Find the queen's position, which requires a linear search of the array and may take 64 load-test cycles.
- Examine the squares to which it is able to move, in all eight directions, until the black king is encountered or the procedure has looked at all possible moves.

This algorithm's most damaging characteristic is that it requires maximum running time when there is no check to be found, which is the most frequent case. With a bitboard representation, the same operation can be implemented as the following sequence of machine instructions:

- Load the "white queen position" bitboard.
- Use it to index the database of bitboards representing squares attacked by queens. This yields a list of squares where a king isn't safe from this queen.
- Logical-AND that bitboard with the one for "black king position".

If the result is non-zero, the white queen may be threatening the black king's position, and the program will need to inspect the squares located between them and look for a blocking piece. Otherwise, the work is done; assuming that the attack bitboard database is in cache memory, the entire operation may have consumed as little as 5 clock cycles.

Because of its efficiency, the bitboard technique has been selected as the basis for Javachess' board representation.

CHAPTER 4: MOVE GENERATION

4.1 Justification

Move generation (i.e., deciding which moves are legal given a specific position) is, with position evaluation, the most computationally expensive part of chess programming. In any given situation, a player may have 30 or more legal moves to choose from, some good, some suicidal. For trained humans, it is easy to characterize the majority of these moves as foolish or pointless, and chess masters know (more through instinctive pattern matching than by conscious effort) which one or two moves are likely to be the strongest in the position.

Coding this information, especially the unconscious type, into a computer has proven spectacularly difficult. Despite considerable effort, programs based on selective pruning (i.e., examination of a few likely moves) have never achieved much success. The strongest programs, except, to some extent, Hans Berliner's Hitech (Berliner [1989]) and its siblings, have given up on this approach, instead relying on a brute force approach: analyze all possible moves as fast as possible and search their consequences as far into the future as resources allow. It may not matter that the program has no clear idea of what it is trying to accomplish, because it will stumble upon a good move *eventually*.

Brute force requires that move generation and search be made as fast as possible. Since move generation is extremely repetitive, some execution time

can be saved by storing a database of all possible moves for all pieces on all squares and reducing the process to a table lookup (Goulet [1986]); this is the approach taken by this project.

4.2 The Choices

Historically, three major move generation strategies have been used:

- Selective generation: Examine the board, come up with a small number of "likely" moves and discard everything else.
- Incremental generation: Generate a few moves, hoping that one of them will trigger a cutoff before generating the others is required.
- Complete generation: Generate all moves at once, hoping that the transposition table will contain enough information about one of them to prove that there is no need to search anything at all.

Selective generation (and its associated search technique, called forward pruning) have all but disappeared since the mid 1970's. As for the other two, they represent two sides of the same coin, trading off effort in move generation for more work during search. Both strategies are sound; however, in games like Othello and GoMoku, where move generation is easy and/or there are lots of ways to transpose into the same positions, complete generation may be most

efficient, while in games where move generation rules are complicated, incremental generation will usually get the job done faster.

4.3 Forward Pruning

In a seminal paper originally published in 1949, Claude Shannon described two ways to build a chess-playing algorithm (Levy [1984]):

- Look at all possible moves, and all the possible moves resulting from each, recursively.
- Only examine the "best" moves, as determined from a detailed analysis of a position, and then only the "best" replies to each, recursively.

At first, the second alternative seemed more likely to succeed, because it closely mimics the way grandmasters play the game and because looking at only a few moves at each ply will result in a deeper search. Unfortunately, the results disproved the theory: at best, selective programs achieved low to mid-level club player ratings, often committing humiliating blunders at the worst possible time. Beating a world champion (or even playing reasonably well on a consistent basis) was beyond their reach (Levy [1984]).

The problem (Welch [1984])is that a "best move generator" has to be almost perfect to be of any value. For example, if a plausible-move generator selects the objective best move 95% of the time at the first ply (which is a risky

assumption in itself), the probability that it will never eliminate a best move from consideration during a 40-move game is $(0.95)^{40}$ or less than 13%. Even a nearly-perfect generator with 99% accuracy will blunder at least once in about a third of its games, as $(0.99)^{40} = 0.669$ is the probability that it will not miss the best move once.

When the Northwestern chess team (Slate and Atkin [1983]) decided to forgo the best-move generator and switch to full-width search, it turned out that the time saved by avoiding costly analysis during move generation was sufficient to cover the expense: it was possible to look at all legal moves in the time previously allocated to the top N. For all intents and purposes, this discovery buried forward pruning for good.

4.4 Generating All Moves At Once

The most straightforward way to implement full-width searching consists of:

- Finding all the legal moves available in a position.
- Ordering them according to a strategy designed to speed up the search.
- Searching them one at a time, until all moves have been examined or a cutoff occurs.

When this move generation scheme is combined with transposition tables, a search may occasionally be averted: if one of the moves has already been

searched to a satisfactory depth before, and if its evaluation (as retrieved from the table) is such that it triggers a cutoff, there will be no need to search anything. Obviously, the larger the transposition table, and the higher the probability of a transposition given the rules of the game, the bigger the average payoff. In chess, it turns out that this technique is extremely valuable in the endgame, where this project often gains 3-4 extra plies thanks to the high number of transpositions.

4.5 Incremental Move Generation

CHESS 4.5 (Slate and Atkin [1983]) adopted the opposite strategy: it generates a few moves at a time, searches them, and avoids generating the others if a cutoff can be found. Programs of the 1970's had to make do with small transposition tables and could not afford the memory expense required by pre-processed move databases, which made the complete generation scheme described above computationally prohibitive.

A common incremental move generation heuristic is to look at captures first, often starting with those of highly valuable pieces, and look for a quick cutoff. The "killer move" heuristic is another: it relies on the hypothesis that many pointless moves in a position can be refuted by the same counterattack.

4.6 Javachess Move Generation

Javachess' move generation algorithm, described in more detail in Chapter 8, is based on a database of possible moves for each piece type in each square on the chess board. Because this database reduces the computation time required by move generation to a minimum, Javachess generates all possible moves at the same time.

CHAPTER 5: AUXILIARY DATA STRUCTURES

This section describes two techniques which, although not absolutely required for the computer to play legal chess, speed up the search and therefore improve the quality of the machine's play: the transposition table (Slate and Atkin [1983]) and the history table (Marsland and Schaeffer [1990], Hartmann [1991]).

5.1 Transposition Tables

One way to speed up search is to take advantage of the fact that chess' search graph is not, strictly speaking, a tree.

In most positions, there are several continuations which lead to the same game state. For example, the openings "1. e2-e4 e7-e5 2. d2-d4" and "1. d2-d4 e7-e5 2. e2-e4" result in identical positions. Reaching the same game state through different continuations is called *transposing.*

If transpositions occur frequently, the program could waste a considerable amount of effort searching the same position multiple times. This is why all chess programs, since at least Richard Greenblatt's Mac Hack VI in the late 1960's, have incorporated a *transposition table,* i.e., a hash table storing recent search results. Whenever a new position is examined by the program, the table is queried first; if it contains suitable data (i.e., a previous search of the same position to a depth equal to or larger than the one needed), then no effort need be expended.

Transposition table collisions are handled by a simple mechanism: new data overwrites old data if it results from a deeper search, or if the old data is deemed obsolete. Otherwise, the new data is ignored. Results of relatively shallow searches are never stored in the table, because repeating these searches is not expensive enough to warrant overwriting the results of deeper examinations.

There are numerous advantages to this process, including:

- Speed. The more results retrieved from the transposition table, the faster the overall search process.

- Increased search depth. For example, if the transposition table already contains a six-ply result for a position which the program needs to search to a depth of 4-ply, not only does it avoid the search, but the stored results are actually more accurate than those which would have been obtained by doing the work.

- Versatility. Every chess program has an "opening book" of some sort, i.e., a list of well-known positions and best moves selected from the chess literature and fed to the program to allow it to play a reasonable game in the opening stages. Since the opening book's modus operandi is similar to the transposition table, it is possible to re-use the transposition table's code for this purpose.

The transposition table's only serious drawback is its voracity in terms of memory. To be of any use whatsoever, the table must contain several thousand to a few million entries. At 16 bytes or so per entry, this can become a problem in memory-starved environments.

5.2 Generating Hash Keys for Chess Boards

The following scheme, used to generate hash keys from chess positions, was described by Zobrist in 1970 (Welch [1984]):

- Generate 12x64 N-bit random numbers (where the transposition table has 2^N entries) and store them in an array. Each random number is associated with a given piece on a given square (i.e., black rook on H4, etc.) An empty square is represented by a null word.
- Start with a null hash key.
- For each piece on the board, XOR the random number associated with this piece on its square to the current hash key.

An interesting side effect of the scheme is that it is very easy to update the hash value after a move, without re-scanning the entire board. For example, if a white rook on H1 captures a black pawn on H4, updating the hash key merely requires three XOR operations on the hash key, using the "white rook on H1" and "black

pawn on H4" to erase them from the board and the "white rook on H4" to add it to the position.

This project uses the same method, with different random piece signatures, to generate a second key (also known as a "lock"; see Standish [1980]) which is stored in the transposition table along with a position's data. This helps to detect collisions: if two boards hash to the exact same key and collide in the transposition table, the locks will be used to differentiate between them. Odds are extremely low that two boards with the same key will also hash to the same lock; specifically, for a transposition table of size N entries, the probability of a key collision is 1/N, while the probability of collisions involving independent keys and locks at the same time is $1/N^2$.

Note that, in the event of a collision, the results of the deeper search (or, in the event of a collision between searches of equal depth, the most recent results) are stored in the table and the others are discarded. Chaining, double hashing and other canonical hash table algorithms (Standish [1980]) would be of little value here; the transposition table is a time-saving trick which would lose much of its effectiveness if it had to support linear-time searches.

5.3 History Tables

The "history heuristic" is a descendant of the "killer move" technique. In both cases, the basic idea is the following: in any given position, most moves are

either pointless or self-defeating, and if a refutation can be found for one, it is likely to refute others as well.

For example, if White's queen is in danger, pushing pawns at the opposite end of the board is likely to be ineffective. Suppose that, during a search, Black discovers that "Bishop takes Queen" is an effective refutation for White's "King's Rook's Pawn to King's Rook's 3". Then, trying "Bishop takes Queen" again when White tries instead to push the pawn to King's Rook's 4 may immediately prove that this White move is just as bad, reducing the search effort for this move to almost nothing.

A history table compiles statistics on the moves which have generated cutoffs in the recent past and should therefore be tried again at a later time. Its implementation is very simple: it consists of a 64x64 array of integer counters, indexed by a pair of squares (the source and the destination for a move.) When the search algorithm detects that a certain move has generated a cutoff, its entry in the history table is incremented. During move generation, history values are used to sort moves and make sure that more "historically powerful" ones will be tried first.

5.4 Javachess Data Structures

Javachess uses a transposition table containing 2^{17} entries and a standard history table. The size of the transposition table was determined by the amount of RAM available on the development machine.

CHAPTER 6: EVALUATION FUNCTIONS

One of the key aspects of a chess-playing program is its evaluation function, which it uses to assess who is winning and who is losing in a given position and to guide its move selection.

Chess programmers can and do spend years refining their evaluation functions (Hartmann [1989]). The one implemented in this project is rather primitive by contemporary standards; it incorporates elements drawn from such successful programs of the past as CHESS 4.5 (Slate and Atkin [1983]), Belle (Condon and Thompson [1981]) and Cray Blitz (Hyatt [1983]) and simplifies them to provide a "coarse-grained" evaluation well-suited to the MTD(f) search algorithm.

Features often found in evaluation functions include the following:

6.1 Material Balance

The dominant factor in the evaluation functions of most chess programs is material. Slate and Atkin [1983], for example, state that the combination of all other factors included in CHESS 4.5's evaluation function account for less than 1.5 times the material value of a pawn.

6.2 Mobility

One of the characteristics of checkmate is that the victim has no legal moves left. Intuitively, it also seems better to have a lot of options available: a player is

more likely to be able to find a good line of play if he has 30 legal moves to choose from than if he is limited to 3. However, it turns out that this metric is worthless, for several reasons. For one thing, in any given position, most of the moves are pointless (Fischer et al [1972]), and awarding a mobility bonus for aimless moves is obviously incorrect. For another, trying to limit the opponent's mobility at all costs may lead the program to destroy its own defensive position in search of "pointless checks": since there are usually few ways to evade check in any given position, a mobility-oriented program would be likely to make incautious moves to put the opponent in check, and after a while, it may discover that it has accomplished nothing and has dispersed its forces all over the board. More importantly, stalemate, which occurs when a player who is not in check is unable to move without putting his king in check, is by rule a draw and not a loss.

6.3 Development

An age-old maxim of chess (Znosko-Borovsky [1935]) states that minor pieces (bishops and knights) should be brought into the battle as quickly as possible, that the King should castle early and that rooks and queens should stay hidden until an opportunity for a decisive attack presents itself.

6.4 Pawn Formations

Chess literature mentions several types of valuable or dangerous pawn features, including pawn rams, doubled or isolated pawns, and passed pawns.

6.5 Tropism

Tropism is a measure of how easy it is for a piece to attack the opposing king, and is usually measured in terms of distance. The exact rules to compute tropism vary by piece type.

6.6 Javachess' Evaluation Function

Javachess uses variants of all of the features described in this chapter. We shall discuss the details of the implementation in Chapter 8.

CHAPTER 7: SEARCH TECHNIQUES

To a computer, discriminating between good and bad moves is far from obvious. Human players can draw on experience, pattern recognition and intuition, none of which are particularly well-suited to algorithmic codification. Instead, numerous algorithms based on game trees have been developed.

7.1 Minimax

One way to compare a set of moves is to look at their consequences. For example, if Black takes White's queen's pawn with his rook, what responses will be available to White? Will he be able to take the rook, capture another piece which was guarded by the rook before his move but is now vulnerable, or occupy an important square? And then, what can Black do to regain the advantage?

This is the crucial insight underlying the Minimax tree search algorithm, which is at the root of all game-playing programs: look at every continuation until it leads to a terminal position (i.e., one player has won or the game is tied) or at least to one stable enough to allow for a reasonable assessment of the situation, and decide which of the initial moves led to the most favorable consequences.

Minimax can be summarized as follows:

- Given a way to evaluate a board position as a probable or certain win for Player 1 (whom we will call Max), for Player 2 (whom we will call Min), or whether the position will lead to a draw. This evaluation takes the form of

a number: a positive number indicates that Max is leading, a negative number, that Min is ahead, and a zero, that neither has acquired an advantage.

- Max's strategy is to make moves which will increase the board's evaluation (i.e., he will try to maximize the evaluation).
- Min's strategy is to make moves which decrease the board's evaluation (i.e., he will try to minimize it).
- The algorithm assumes that both players always make the moves which optimize the evaluation from their own point of view.

Therefore, if Max is forced to choose between a move which is certain to result in a marginal advantage and one which might lead to a quick win if Min made a mistake but to a draw if Min discovered an obscure and convoluted continuation, Minimax will favor the former.

Minimax' complexity is exponential: $O(b^n)$, where b is the game's branching factor and n is the depth of the search. This is a considerable problem in chess, where the branching factor in the middle game is usually around 35. Fortunately, it turns out that a lot of the work performed by minimax is unnecessary and can be eliminated at no risk; several algorithms have been developed to do so (Slate and Atkin [1983], Althöfer [1990]), and this project implements a state-of-the-art minimax variant known as iterative deepening MTD(f) search (Plaat [1987]).

7.2 Justifications for search

Deep searches are an easy way to "teach" the machine about relatively complicated tactics, because a full-width search of a game tree to depth N will by definition examine every move combination of depth n or less, however subtle. For example, consider the knight fork, a move which places a knight on a square from which it can attack two different pieces. Finding a way to represent this type of position logically would require some effort, more so if we also had to determine whether the knight was itself protected from capture. However, a simple 3-ply search will discover the forking opportunity on its own, by moving the knight to the forking square at ply 1, developing all of the opponent's replies at ply 2, and capturing an undefended piece with every move at ply 3. If the program's evaluation function places a high value on material advantage, the forking move will be considered highly advantageous.

A full-width search will never miss an opportunity, assuming that it is apparent within its search depth. Therefore, the deeper the search, the more complicated the "plans" which the machine can stumble upon.

7.3 The Alphabeta Algorithm

Fortunately, there are sound search algorithms which only need to examine a fraction of the nodes visited by Minimax. For example, the Alphabeta algorithm

takes advantage of the fact that poor moves can quickly be eliminated from consideration once a much better one has been found.

For example, let us suppose that Max has already examined a move A and determined that it leads to a position with a value of +5. He now begins searching another move B, to which Min's first response leads to a value of –3. Therefore, there is no need to look at any other branch resulting from this B, because the temporary search results have already demonstrated that it will end up being worse than the current best choice.

7.4 Ordering Moves to Optimize Alphabeta

Alphabeta's efficiency depends heavily on the order in which moves are searched. As alphabeta can cut off searching a given move only if it already knows of a provably better one. Cutoffs occur more often when good moves are examined early. The gains and losses related to the ordering are not trivial: a perfect ordering, defined as one which will cause the largest possible number of cutoffs, will result in a search of about twice the square root of the number of nodes in the tree associated with the worst possible ordering (Shinghal [1992]). For example, a 6-ply search in the middle game (when the average branching factor is 35) will examine 35^6 nodes, or roughly 1.8 billion, while a perfectly ordered alphabeta will only need to visit approximately $2 * 35^3$ or 85,000 nodes.

Achieving the absolute best case is hard, but luckily it is not necessary. It turns out that Alphabeta performs well as long as it can quickly find a reasonable move to compare others to. This means that it is important to search a good move first; the best case happens when we always look at the best possible moves before any others. If the moves are ordered in such a way that each one is better than anything examined before, alphabeta will be unable to prune *anything* and the search will reduce to Minimax.

Several move ordering techniques have been developed, including the history table described earlier in this text. It turns out that the most effective method is one which flies in the face of human intuition, because of the fact that it seems to duplicate an enormous amount of effort on purpose: iterative deepening.

7.5 Iterative Deepening AlphaBeta (Slate and Atkin [1983])

When searching a position to depth 6, the ideal move ordering would be the one yielded by a prior search of the same position to the same depth. Since that is obviously impossible, how about using the results of a shallower search, say of depth 5?

This is one of the justifications for iterative deepening: begin by searching all moves arising from the position to depth 2, use the scores to reorder the moves, search again to depth 3, reorder, etc., until the desired depth has been reached.

This technique seems tremendously wasteful, because nodes located at shallow depths will be re-searched several times. However, given a large branching factor, the last ply dominates a search to such an extent that the rest becomes virtually insignificant.

Consider the size of a search tree of depth d with branching factor B. The tree has B nodes at depth 1, B*B at depth 2, B*B*B at depth 3, etc. Therefore, searching to depth d yields a tree B times larger than searching to depth d-1. The number of non-leaf nodes in the tree, N, is about BD – 1, while the number of leaves is (B – 1) BD. The ratio of non-leaf nodes to leaf nodes is approximately 1 / (B – 1); for chess, with a branching factor of 35, it is 1 / 35 or less than 3% (Shinghal [1982]). Therefore, the added expense linked to re-searching interior nodes in iterative deepening alphabeta is dwarfed by the effort required to visit the leaves.

However, using the results of a shallow search to order the moves prior to a deeper one also produces a significant increase in the alphabeta cutoff rate. Therefore, iterative-deepening alphabeta actually examines far fewer nodes, on average, than a non-iterative alphabeta search to the same depth. When a transposition table enters the equation, the gain is even more impressive: the work required to duplicate the shallow parts of the search disappears because the results are already stored in the table and need not be computed again.

This project uses iterative deepening to fill up the transposition table with useful results, but not to reorder moves during search. That job is left exclusively to the history heuristic, to keep the code simple and to allow easy and predictable experimentation with various history variants.

7.6 Aspirated Search

Alphabeta assumes nothing about a position's ultimate minimax value. However, given a reasonable estimate of the ultimate value (for example, the results of a shallower search in an iterative-deepening scheme), the algorithm may be able to evaluate a line of play faster without loss of precision.

Aspirated Search is a variant of alphabeta in which a small window centered on the expected value is used to bound the search. If the position's actual value falls within the window, the extra cutoffs enable the algorithm to find it much faster. If not, it will be necessary to search again with a wider window. Obviously, the more accurate the initial estimate of the position's minimax value, the more useful this technique is. A poor estimate will result in numerous repeat searches and consume whatever gains the extra cutoffs in each search may provide.

7.7 MTD(f)

Plaat [1997] extended aspirated search into the MTD(f) (Memory-enhanced Test Driver) algorithm. This algorithm consists of a binary search into the space of possible game state value, and works by repeatedly calling an aspirated alphabeta routine with a very narrow search window centered on a current estimate of the position's evaluation.

If one of the calls to aspirated alphabeta succeeds, the returned value is the real evaluation and MTD(f) returns. However, if the real evaluation is not within the search window (that is, it is not the current estimate), aspirated alphabeta will fail, but will do so very quickly because it will cutoff every search path in the game tree almost immediately. A failed aspirated alphabeta search returns a value equal to one of the bounds of its search window; if the returned value is equal to the lower bound of the search window, the actual value of the game tree is less than the lower bound of the search, and vice versa.

For example, suppose that a position's real evaluation is +5, and that the current estimate (possibly returned by a shallower MTD(f) search in an iterative scheme) is +20. MTD(f) would begin by calling an aspirated alphabeta search on the position, with a search window of, say, +19.99 to +20.01. The aspirated search will "fail low", indicating that the true value is between –INFINITY and +19.99; a new estimate will then be selected, and the process started anew.

Layered on top of an alphabeta implementation equipped with a transposition table, MTD(f) is extremely efficient and supports a high level of concurrency. (The gain over ordinary iterative alphabeta is not as spectacular in a sequential implementation.)

Additionally, MTD(f) converges on a value faster when working with a coarse-grained (and therefore simpler and faster to compute) evaluation function.

7.8 Quiescence Search

In chess, a full-width search to a fixed depth is insufficient to guarantee good results, because a great many positions cannot be evaluated accurately.

For example, suppose that Max captures a pawn at ply 5. An evaluation applied at this point would yield results indicating that Max is ahead. However, a 6-ply search might have revealed that the pawn capture left Max' queen vulnerable to capture, and that the position resulting from it was in fact very bad. This is a very simple example of the "horizon effect" initially described by Berliner [1973]; because of the horizon effect, fixed-depth searches are flawed.

An evaluation function can only be applied effectively to "quiet" (or quiescent) positions (Beal [1989], Beal [1990], Kaindl [1983]) where nothing of consequence is likely to happen in the immediate future. The definition of a quiescent position depends on the rules of the game being programmed; in chess, since most evaluation functions are heavily biased towards material balance, any position in

which the side to move has no capture, pawn promotion or checking move available is usually considered quiet. Unfortunately, performing a full-width search until every continuation leads to such a quiet position is rarely feasible. A reasonable compromise is to begin with a full-width search to a fixed depth, to continue each line of play selectively by searching captures (and possibly other "non-quiescent" moves) until a quiescent position is reached, and only then apply the evaluator.

7.9 The Null-Move Heuristic

Beal [1989] describes an additional refinement to quiescence search known as the null-move heuristic. While performing quiescence search on a position, Beal examines a null-move first, that is, his program skips a turn and lets the opponent play twice in a row. This technique presents several advantages, including the following:

- Quick detection of overwhelming advantages. For example, if Max's advantage in a position is so overwhelming that alphabeta generates a beta cutoff below the null move (i.e., Min can't defend himself appropriately even with two consecutive moves), the cutoff has effectively been found after a search one-ply shallower than expected, and therefore at a fraction of the cost.

- ❖ If the null move's evaluation is better than those of all other non-quiescent moves (for example, if each capture move leaves the queen en prise), then the real best move would be a quiet one. The position can then be considered quiescent and quiescence search terminated along the current continuation.

7.10 Javachess' Search Algorithm

Early versions of Javachess used iterative-deepening alphabeta as their search algorithm. A quiescence search algorithm (which only examines null moves and captures) was added later. Currently, Javachess implements MTD(f), which calls the iterative-deepening alphabeta with quiescence search extensions during each of its iterations. More details can be found in Chapter 8.

CHAPTER 8: DESIGN OF JAVACHESS

Javachess adopts many of the techniques developed by the creators of CHESS 4.5 and Cray Blitz and packages them into an object-oriented form. This section briefly describes the highlights of its architecture.

8.1 Board Representation

The jcBoard class encapsulates Javachess' board representation. Core data is stored in a set of twelve bitboards, each of which contains the list of squares where a given piece of a given color are located. Two supplemental bitboards, containing the lists of all squares occupied by Black and by White, are added to speed up move generation; for example, when generating moves for the white queen, all squares currently occupied by other white pieces are off-limits, so testing a possible move against a single board containing all white pieces will save time compared to checking with all types of white pieces separately.

8.2 Move Generation

Javachess includes a detailed, pre-processed list of all possible moves for all pieces on all squares, patterned after Goulet [1986] and based on the following principles:

- For move generation purposes, piece color is irrelevant except for pawns which move in opposite directions.

- There are 64 x 5 = 320 combinations of pieces (excluding pawn) and squares from which to move, 48 squares on which a black pawn can be located (they can never retreat to the back rank, and they get promoted as soon as they reach the eight rank), and 48 where a white pawn can be located.

- Let us define a "ray" of moves as a sequence of moves by a piece, from a certain square, in the same direction.

- For each piece on each square, there are a certain number of rays along which movement might be possible. For example, a king in the middle of the board may be able to move in 8 different directions, while a bishop trapped in a corner only has one ray of escape possible.

- When generating moves for a piece on a square, scan each of its rays until either the end of the ray or a piece is encountered.

With a properly designed database, move generation is reduced to a simple, mostly linear search which requires very little computation. The database requires less than 30K of memory and saves considerable time during search. Special rules for castling and en passant captures complement the procedure.

8.3 Data Structures

In addition to the move generation database, Javachess includes a history table and a transposition table containing 2^{17} entries. The results, at the current depth of search, are satisfactory: the transposition table usually contained suitable results for about 10% of positions in the middle game and over 75% (sometimes 90% or more) in the endgame; this high success rate accounts for much of the added search depth in endgame situations.

Experiments with larger transposition tables (up to 2^{20} entries) have shown significant speed improvements on machines which can store the entire table in core memory.

8.4 Search Techniques

Javachess implements an iterative-deepening MTD(f) algorithm whose maximum depth is determined dynamically, according to how much work has been expended during the earlier parts of the search. Currently, MTD(f) iterates until it reaches an iteration which visits over 100,000 nodes during its last call to an aspirated, iterative-deepening alphabeta search with quiescence and null-move augmentations. Typically, this results in searches of 4-ply (plus quiescence search to an unlimited depth) in the opening and middle game, and 7- or 8-ply in the endgame.

Javachess' quiescence search only examines captures and pawn promotions. Some other programs also look at checks in this phase; however, since it is possible to generate long and possibly cyclic sequences of checks and check evasions, adding this feature to the program would have cost a significant amount of processing power. It may be added in the future.

Experimentation has shown that the branching factor in the quiescence search is small (usually less than 2 on average) and that its depth usually reaches 6- to 8-ply before the program runs out of captures to try. Quiescence search typically consumes 50% to 75% of the total search effort in most positions; improvements which would reduce this expenditure (i.e., depth limits) without reducing search efficiency would definitely be welcome.

Experiments have shown that the null-move heuristic saves between 20% and 50% of the effort required by a given search, at the cost of a minimal amount of code.

8.5 Singular Extensions

Initially, JavaChess was to implement a version of Deep Blue's singular extensions (Anantharaman et al [1990]), a technique which consists of adding a ply of search to continuations where one move seems vastly superior to all others, in order to verify that this apparent superiority is not a mere artifact of the

horizon effect. Singular extensions add an extra "safety" factor to the search results, at a cost of roughly doubling the size of the search tree.

However, on the development computer, the program is limited to a depth of 4-ply (plus a quiescence search of unlimited depth) in the middle game, and adding the extensions made testing impractical. They were therefore removed from the code.

8.6 Evaluation Function

Javachess' evaluation function is twofold: a "quick evaluator" based exclusively on material balance is computed first, and only if the result is close to the current best move's will the other factors by included. "Close", in this case, means that the difference between the quick evaluation and the current best move's value must be less than twice the material value of a pawn.

Material balance term

The dominant factor in the evaluation function implemented in Javachess is material. The material evaluation function is a direct adaptation of the one in CHESS 4.5 and embodies the following principles:

- The queen is worth 900 points, the rook is worth 500, the bishop 350, the knight 300 and the pawn 100.
- A bigger numerical advantage in material is always desirable.

- When a player is ahead on material, he should exchange pieces of equal value (but not pawns).

CHESS 4.5's creators estimate that an enormous advantage in position, mobility and safety is worth less than 1.5 pawns; this is approximately the numerical relationship between material and the sum of the other factors in this project as well. There are positions where a material sacrifice is required to obtain an advantage in momentum or position; these, however, are best discovered through the search.

Javachess' actual material balance function, from the point of view of the player in the lead, is the following:

MB = MatDiff + (MatDiff * InciteSwaps * Pawns) / (Pawns + 1)

Where:

- MatDiff is the minimum between 2,400 points and the actual material advantage of the player in the lead (the limit of 2,400 is over 2.5 times the value of a queen; pursuing any further material advantage would be pointless)

- InciteSwaps is (12,000 – Total value of all material on board) / 6,400; it embodies the fact that once a side has achieved material advantage, swapping pieces of equal value is usually a good idea

- Pawns is the number of pawns the winning side has on the board. The (pawns / (pawns + 1)) ratio indicates that swapping pawns is rarely a good idea for the winning side, because a material advantage can often result in queening during the endgame if enough pawns are available.

If neither player has an advantage, for example at the beginning of the game, material balance is equal to zero.

Mobility factor

Javachess implements a few mobility evaluation features:

- Bad bishops: A Bishop's value is reduced by an amount equal to 8% of the material value of a pawn times the number of friendly pawns located on squares to which it is allowed to move.

- Rooks sitting on open files receive bonuses of 10% of the value of a pawn. Rooks on semi-open files receive bonuses of 4% of the value of a pawn.

Development factor

This project implements a development metric in the following ways.

- First, it penalizes the King's and Queen's pawns by 15% of their value if they have not moved at all.

- ❖ It also penalizes knights and bishops located on the back rank, where they hinder rook movement, by 10% of a pawn.

- ❖ It tries to prevent the queen from moving until all other (non-pawn) pieces have done so, by penalizing a queen not sitting on its home square by 8% of a pawn for each friendly bishop, knight or rook still on its original square

- ❖ If the opponent's queen is still alive, it gives a bonus of 10% of the value of a pawn to positions where the king has castled. It also penalizes positions where the king has not yet castled and where castling rights have been lost; the penalty varies from 40% to 120% of the value of a pawn, depending on whether some castling rights are still available or not.

The result is that the program plays very defensively in the first few moves following its first forays outside of the opening book, and attempts to castle as quickly as it can.

The development factor is important in the opening, but quickly loses much of its relevance. After ten to twelve moves, it tends to even out.

Pawn features factor

The pawn features used by this program include:

- Doubled or tripled pawns. Two or more pawns of the same color on the same file are penalized by 8% of a pawn's value, because they hinder each other's movement.

- Pawn rams. Two opposing pawns blocking each other's forward movement each receive an 8% penalty because they hinder each other's movement.

- Passed pawns. Pawns which have advanced so far that they can no longer be attacked or rammed by enemy pawns are very strong, because they threaten to reach the back rank and receive promotion. Passed pawns receive bonuses inversely proportional to their distance to the opponent's back rank (where they will be promoted); a passed pawn on the 5^{th} rank is worth 1.25 pawns, while one on the seventh rank is worth 1.5 pawns. An extra 25% of a pawn is awarded when a passed pawn is protected by a friendly rook located on the same file.

- Isolated pawns. A pawn which has no friendly pawns on either side is vulnerable to attack and should seek protection; it is penalized by 15% of its value because of the likelihood that pieces will have to be tied up defending it.

- Eight pawns. Having too many pawns on the board restricts mobility; opening at least one file for rook movement is a good idea. Positions in which a side still has its eight pawns are penalized by 10% of a pawn.

Tropism factor

Only three types of pieces are evaluated for king tropism in Javachess:

- A rook is penalized by 2% of the value of a pawn for each square of distance between its position and the opposing king's rank or file, i.e., if the rook is on the king's rank, it is not penalized at all, no matter how many files separate them.
- A queen receives only half the penalty assessed to a rook at the same distance, because of its higher mobility.
- A knight receives a penalty of 1% of a pawn for each square on the shortest path between it and the enemy king.

Final notes on evaluation

Again, it must be stated that the relative weights of the various terms included in an evaluation function are usually assigned by trial and error over a long period of time. Some of those used here have been taken from the literature, while others were selected to influence the program's playing style. No assertion is made regarding their suitability in a competitive setting.

8.7 User Interface

JavaChess' current text-based user interface is intended as a mere test harness for the artificial intelligence functionality, which will be connected to a full-fledged graphical application at a later time. Therefore, its user-friendliness is primitive by any current standards: for example, the player must follow a very strict format when entering moves, and any deviation from the norm will result in a rejection and a request for re-entry..

The input/output system consists of three components:

- Keyboard input: Human players communicate their moves to the system by specifying the source and destination squares as integer numbers between 00 and 63. Attempts at illegal moves are detected and reported to the player.

- Screen output: The state of the game is printed as an 8x8 array of squares, each of which may contain a piece tag (i.e., "WQ" for the white queen).

- Game loading: The game can load positions from disk, using a simple ASCII file. The file's structure is designed to make manipulation and modification by the user easy. Creating a new Javachess file in a standard text editor takes a few minutes at most.

The jcBoard class, which encapsulates chess board functionality, is also equipped with a game-save feature, but the user interface does not expose it as of yet.

CHAPTER 9: JAVACHESS BEHAVIOR

Experiments with the program have shown it to perform very well when analyzing chess problems. Twenty-five positions from Reinfeld [1955] were randomly selected and fed to the program; it solved most in seconds and all of them in less than five minutes, sometimes searching as deep as 9-ply to do so.

To informally assess Javachess' strength in competitive play, the author stored four endgame positions on disk and played each of them against it, from both sides. Javachess won 4 games, drew 2 and lost 2. The same test set was then used to evaluate Javachess against Chessmaster 3000; when facing Chessmaster at the "beginner" setting, Javachess won 7 games and tied 1, while against the highest setting, it won only 1 game, drew 3 and lost 4. As expected, the program plays passable chess, but provides no threat to strong players.

However, the program's performance quickly breaks down in middle game situations. Due to the size of the game tree searched by the quiescence search algorithm, Javachess usually exhausts its work allotment for a move at ply 4, sometimes even at ply 3. While some of this degradation is due to a very small allotment (100,000 nodes total) designed to make the program play in 30 seconds or less, it seems likely that spending such an overwhelming majority of the search effort in quiescence is counter-productive, and that gaining an additional one or two plies of full-width search at the cost of some accuracy in quiescence would improve the program's play.

Finally, while Javachess supports multiple opening books of various sizes, its current book contains a mere 38 positions and moves taken from Znosko-Borovsky [1935]'s discussion of the Ruy Lopez opening. The program is therefore helpless in the opening, except when the player is careful to stay within these restrictive boundaries. Further, not enough time was available to analyze the program's playing style and to determine whether the Ruy Lopez is even a suitable opening for it; the author's chess skills may be insufficient to determine this accurately in any event.

CHAPTER 10: CONCLUSION

For decades, chess has served as the major test bed of artificial intelligence research, much like the fruit fly in genetics. Thankfully, the techniques developed by the research community apply to a wide variety of games and in many other domains.

This project succeeded in implementing an object-oriented, multi-platform chess-playing program which, if weak by contemporary standards, provides a serious challenge to casual players and a useful didactic tool for new game programmers. Further work on the program will include:

- Continuing experimentation with new search algorithms and evaluation function features.
- Development of several opening libraries to transform the program into a training tool.
- Integration of endgame position databases freely available over the internet.
- Extension of the code base to cover other strategy games.

BIBLIOGRAPHY

Althöfer, I. (1990), ***An incremental negamax algorithm***, in *Artificial Intelligence* v. 43, pp. 57-65.

Anantharaman, T., Campbell, M.S. and Hsu, F.-H. (1990), ***Singular extensions: adding selectivity to brute-force searching***, in *Artificial Intelligence* v. 43, pp. 99-109.

Beal, D.F. (1989), ***Experiments with the null move***, in Beal, D.F. ed., *Advances in Computer Chess 5*, Elsevier Science Publishers, North-Holland.

Beal, D.F. (1990), ***A generalised quiescence search algorithm***, in *Artificial Intelligence* v. 43, pp. 85-98.

Berliner, H. (1973), ***Some Necessary Conditions for a Master Chess Program***, in *Proceedings of the Third International Joint Conferences on Artificial Intelligence*, Stanford, AAAI, Menlo Park (California).

Berliner, H. (1989), ***Some innovations introduced by Hitech***, in Beal, D.F. ed., *Advances in Computer Chess 5*, Elsevier Science Publishers, North-Holland.

Condon, J.H. and Thompson, K. (1981), ***BELLE chess hardware***, in *Advances in Computer Chess 3.*

Fischer, B., Margulies, S., Mosenfelder, D. (1972), ***Bobby Fischer teaches Chess***, Bantam, New York.

Goulet, J. (1986), ***Data structures for chess programs***, McGill University thesis.

Hyatt, R. (1983), ***Cray Blitz: a chess-playing program***, University of Southern Mississipi thesis.

Hartmann, D. (1989), ***Notions of evaluation functions***, in Beal, D.F. ed., *Advances in Computer Chess 5*, Elsevier Science Publishers, North-Holland.

Hartmann, D. et al (1991), ***Sundry computer chess topics***, in Beal, D.F. ed., *Advances in Computer Chess 6*, Elsevier Science Publishers, North-Holland.

Kaindl, H. (1983), ***Quiescence search in computer chess***, in Bramer, M.A., *Computer game-playing theory and practice*, Ellis Horwood.

Levy, D. (1988), ***Computer chess compendium***, Springer, New York.

Marsland, T. and Schaeffer, J. editors (1990), ***Chess, Computers and Cognition***, Springer-Verlag, New York.

Plaat, A. (1997), ***MTD(f), a minimax algorithm faster than Negascout***, published on the author's web page.

Reinfeld, F. (1955), ***1001 Brilliant ways to checkmate***, Sterling, New York.

Shinghal, R. (1992), ***Formal concepts in artificial intelligence***, Chapman & Hall, London.

Slate, D.J. and Atkin, L.R. (1983), ***Chess 4.5 - The Northwestern University chess program***, in Frey, P. ed., *Chess skill in man and machine*, 2nd ed., Springer-Verlag.

Standish, T. (1980), ***Data structure techniques***, Addison-Wesley, Reading.

Thompson, K. (1986), ***Retrograde analysis of certain endgames***, International Computer Chess Association Journal, vol. 9, issue 3.

Welch, D.E. (1984), ***Computer chess***, W.C. Brown, Dubuque (Iowa).

Znosko-Borovsky, E.A. (1935), ***How to Play the Chess Openings***, Sir Isaac Pitman & Sons, London.

APPENDIX: USER'S GUIDE

Getting started

To invoke Javachess from the command line, type the following:

```
<java> javachess.jcApp <openings> [<starting position>]
```

Where <java> is the complete command line required to start java on your system, <openings> is a text file containing the opening book, and [<starting position>] is an optional "saved game" text file to load prior to beginning play.

Javachess begins by asking the user to decide whether a human player or Javachess will control each side. It is possible to use Javachess for a two-player game between humans, or to have the program play itself.

Entering moves

Javachess' current test harness user interface expects moves to be provided in a very strict and somewhat awkward format. Each move is exactly 5 characters long, and contains:

- The move's source square as a two-digit number between 00 and 63. The top corner of the chess board (Black's Queen's Rook's starting square) is defined as square 00, Black's back rank contains squares 00 to 07, and other squares are numbered rank by rank.

- A blank space.
- The move's destination square, using the format described for the source square.

For example, if White wants to castle kingside, he would enter "60 62" for the king's move; javachess would handle the rook's move itself.

The only exception to the scheme described above is for resignation: if the player wants to quit the game, he must type "RESIG", again in exactly five characters. Javachess is unable to offer resignation of its own at this point.

APPENDIX: SOURCE CODE

```
/*****************************************************************************
 * jcApp.java - The JavaChess main program
 * by F.D. Laramée
 *
 * Purpose: Entry point, and not much else!
 *
 * History:
 * 07.06.00 Initial build.
 *****************************************************************************/

package javachess;

import javax.swing.UIManager;
import java.awt.*;
import javachess.jcGame;

/*****************************************************************************
 * public class jcApp
 * The application-level class, surrounding everything else.
 *
 * Most of this code has been auto-generated by JBuilder; my role was limited
 * to re-formatting it to make it legible, and to add the jcGame calls at the
 * end of the main program.
 *****************************************************************************/

public class jcApp
{
  // Constructor
  public jcApp()
  {
    // Make the window, since Java needs one
    // We won't be making much use of it, though; all of the i/o
    // will pass through the console
    jcFrame frame = new jcFrame();
    frame.validate();
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = frame.getSize();
    if ( frameSize.height > screenSize.height )
    {
      frameSize.height = screenSize.height;
    }
    if ( frameSize.width > screenSize.width )
    {
      frameSize.width = screenSize.width;
    }
    frame.setLocation((screenSize.width - frameSize.width) / 2, (screenSize.height - frameSize.height) /
2);
    frame.setVisible( true );
  }

  // Main method
  // Initialize and launch the jcGame object
  public static void main( String[] args )
  {
    // Extract the parameters
    String openingBook = args[ 0 ];
    String startingPos = "NONE";
    if ( args.length > 1 )
      startingPos = args[ 1 ];

    // Make the application
    try
    {
      UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
    }
    catch( Exception e )
```

```
    {
      e.printStackTrace();
    }
    new jcApp();

    // Initialize the game controller
    jcGame theGame = new jcGame();
    try
    {
      theGame.InitializeGame( openingBook, startingPos );
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }

    // Run the game
    try
    {
      theGame.RunGame();
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }

    System.exit( 0 );
  }
}
```

```
/*****************************************************************************
 * jcFrame.java - GUI for JavaChess
 * by F.D. Laramée
 *
 * Purpose: Sometime in the very distant future, I may graft a true GUI onto
 * this game (i.e., drag-and-drop pieces to move, etc.)  In the meantime, this
 * class will only contain the absolute bare minimum functionality required
 * by Java: an empty window with a "close box" allowing quick exit.
 *****************************************************************************/

package javachess;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*****************************************************************************
 * public class jcFrame
 *****************************************************************************/

public class jcFrame extends JFrame
{
  // GUI data members
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();

  // Constructor
  public jcFrame()
  {
    enableEvents( AWTEvent.WINDOW_EVENT_MASK );
    try
    {
      jbInit();
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }

  // GUI Component initialization
  private void jbInit() throws Exception
  {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout( borderLayout1 );
    this.setSize( new Dimension( 400, 300 ) );
    this.setTitle( "Java Chess 1.0" );
  }

/*****************************************************************************
 * Event handlers
 *****************************************************************************/

  // processWindowEvent: Overridden so we can exit when window is closed
  protected void processWindowEvent( WindowEvent e )
  {
    super.processWindowEvent( e );
    if ( e.getID() == WindowEvent.WINDOW_CLOSING )
    {
      System.exit(0);
    }
  }
}
```

```
/*****************************************************************************
 * jcGame.java - The JavaChess game controller
 * by François Dominic Laramée
 *
 * Purpose: Coordinate the efforts of all other game-related objects.  This
 * work has been separated from the jcApp (application-level object) because
 * the latter may come to take on more message-passing and related duties
 * if I ever add a GUI to the game, and I didn't want interface and game
 * mechanics to get mixed up in a single class.
 *
 * History
 * 08.06.00 Created
 *****************************************************************************/

package javachess;

import javachess.jcPlayer;
import javachess.jcBoard;
import javachess.jcMoveListGenerator;
import javachess.jcOpeningBook;
import java.io.*;

/*****************************************************************************
 * public class jcGame
 *****************************************************************************/

public class jcGame
{
  // The two players involved in the current game
  jcPlayer Players[];

  // The state of the game
  jcBoard GameBoard;

  // The opening book
  jcOpeningBook Openings;

  // A wrapper for the keyboard
  InputStreamReader kbd;

  // Constructor
  public jcGame()
  {
  }

  // boolean InitializeGame()
  // Select the players, create subsidiary objects and prepare to play
  public boolean InitializeGame( String openingBook, String startingPos ) throws Exception
  {
    // Read the opening book
    Openings = new jcOpeningBook();
    Openings.Load( openingBook );

    // Load the initial position, if any
    GameBoard = new jcBoard();
    if ( startingPos.equalsIgnoreCase( "NONE" ) )
      GameBoard.StartingBoard();
    else
      GameBoard.Load( startingPos );

    // Initialize the keyboard
    kbd = new InputStreamReader( System.in );
    int key;

    // Identify the two players
    Players = new jcPlayer[ 2 ];
    key = 'C';
    System.out.println( "Welcome to Java Chess.  Who plays white: [H]uman or [C]omputer? " );
    try {
      do
      {
```

```
        key = kbd.read();
      }
      while( ( key != 'H' ) && ( key != 'h' ) && ( key != 'c' ) && ( key != 'C' ) );
    } catch( IOException e ) {}

    if ( ( key == 'H' ) || ( key == 'h' ) )
    {
      Players[ jcPlayer.SIDE_WHITE ] = new jcPlayerHuman( jcPlayer.SIDE_WHITE, kbd );
    }
    else
    {
      Players[ jcPlayer.SIDE_WHITE ] =
        new jcPlayerAI( jcPlayer.SIDE_WHITE, jcAISearchAgent.AISEARCH_MTDF, Openings );
    }

    System.out.println( "And who plays black: [H]uman or [C]omputer? " );
    try {
      do
      {
        key = kbd.read();
      }
      while( ( key != 'H' ) && ( key != 'h' ) && ( key != 'c' ) && ( key != 'C' ) );
    } catch( IOException e ) {}

    if ( ( key == 'H' ) || ( key == 'h' ) )
    {
      Players[ jcPlayer.SIDE_BLACK ] = new jcPlayerHuman( jcPlayer.SIDE_BLACK, kbd );
    }
    else
    {
      Players[ jcPlayer.SIDE_BLACK ] =
        new jcPlayerAI( jcPlayer.SIDE_BLACK, jcAISearchAgent.AISEARCH_MTDF, Openings );
    }

    return true;
  }

  // boolean RunGame()
  // A simple loop getting moves from the current player until the game is over
  public boolean RunGame() throws Exception
  {
    jcPlayer CurrentPlayer;
    jcMove Mov;

    do
    {
      // Show the current game board
      GameBoard.Print();

      // Ask the next player for a move
      CurrentPlayer = Players[ GameBoard.GetCurrentPlayer() ];
      Mov = CurrentPlayer.GetMove( GameBoard );
      System.out.print( jcPlayer.PlayerStrings[ GameBoard.GetCurrentPlayer() ] );
      System.out.print( " selects move: " );
      Mov.Print();

      // Change the state of the game accordingly
      GameBoard.ApplyMove( Mov );

      // Pause
      Thread.currentThread().sleep( 2000 );

    } while( ( Mov.MoveType != jcMove.MOVE_RESIGN ) &&
             ( Mov.MoveType != jcMove.MOVE_STALEMATE ) );

    System.out.println( "Game Over.  Thanks for playing!" ) ;

    return true;
  }
}
```

```
/*****************************************************************************
 * jcBoard.java - Encapsulation of a chess board
 * by François Dominic Laramée
 *
 * Purpose: This object contains all of the data and methods required to
 * process a chess board in the game.  It uses the ubiquitous "bitboard"
 * representation.
 *
 * History:
 * 08.06.00 Created
 * 14.08.00 Made "HashLock" a relative clone of "HashKey"; the java
 *          Object.hashCode method is unsuitable to our purposes after all,
 *          probably because it includes memory addresses in the calculation.
 *****************************************************************************/

package javachess;
import java.util.Random;
import java.io.*;

/*****************************************************************************
 * public class jcBoard
 *
 * Notes:
 * 1. The squares are numbered line by line, starting in the corner occupied by
 * Black's Queen's Rook at the beginning of the game.  There are no constants
 * to represent squares, as they are usually manipulated algorithmically, in
 * sequences, instead of being explicitly identified in the code.
 *****************************************************************************/

public class jcBoard
{

/*****************************************************************************
 * CONSTANTS
 *****************************************************************************/

  // Codes representing pieces
  public static final int PAWN = 0;
  public static final int KNIGHT = 2;
  public static final int BISHOP = 4;
  public static final int ROOK = 6;
  public static final int QUEEN = 8;
  public static final int KING = 10;
  public static final int WHITE_PAWN = PAWN + jcPlayer.SIDE_WHITE;
  public static final int WHITE_KNIGHT = KNIGHT + jcPlayer.SIDE_WHITE;
  public static final int WHITE_BISHOP = BISHOP + jcPlayer.SIDE_WHITE;
  public static final int WHITE_ROOK = ROOK + jcPlayer.SIDE_WHITE;
  public static final int WHITE_QUEEN = QUEEN + jcPlayer.SIDE_WHITE;
  public static final int WHITE_KING = KING + jcPlayer.SIDE_WHITE;
  public static final int BLACK_PAWN = PAWN + jcPlayer.SIDE_BLACK;
  public static final int BLACK_KNIGHT = KNIGHT + jcPlayer.SIDE_BLACK;
  public static final int BLACK_BISHOP = BISHOP + jcPlayer.SIDE_BLACK;
  public static final int BLACK_ROOK = ROOK + jcPlayer.SIDE_BLACK;
  public static final int BLACK_QUEEN = QUEEN + jcPlayer.SIDE_BLACK;
  public static final int BLACK_KING = KING + jcPlayer.SIDE_BLACK;
  public static final int EMPTY_SQUARE = 12;

  // Useful loop boundary constants, to allow looping on all bitboards and
  // on all squares of a chessboard
  public static final int ALL_PIECES = 12;
  public static final int ALL_SQUARES = 64;

  // Indices of the "shortcut" bitboards containing information on "all black
  // pieces" and "all white pieces"
  public static final int ALL_WHITE_PIECES = ALL_PIECES + jcPlayer.SIDE_WHITE;
  public static final int ALL_BLACK_PIECES = ALL_PIECES + jcPlayer.SIDE_BLACK;
  public static final int ALL_BITBOARDS = 14;

  // The possible types of castling moves; add the "side" constant to
  // pick a specific move for a specific player
  public static final int CASTLE_KINGSIDE = 0;
```

```
  public static final int CASTLE_QUEENSIDE = 2;

/**************************************************************************
 * DATA MEMBERS
 **************************************************************************/

  // An array of bitfields, each of which contains the single bit associated
  // with a square in a bitboard
  public static long SquareBits[];

  // Private table of random numbers used to compute Zobrist hash values
  // Contains a signature for any kind of piece on any square of the board
  private static int HashKeyComponents[][];
  private static int HashLockComponents[][];

  // Private table of tokens (string representations) for all pieces
  public static String PieceStrings[];

  // Data needed to compute the evaluation function
  private int MaterialValue[ ];
  private int NumPawns[ ];
  private static int PieceValues[ ];

  // And a few flags for special conditions.  The ExtraKings are a device
  // used to detect illegal castling moves: the rules of chess forbid castling
  // when the king is in check or when the square it flies over is under
  // attack; therefore, we add "phantom kings" to the board for one move only,
  // and if the opponent can capture one of them with its next move, then
  // castling was illegal and search can be cancelled
  private long ExtraKings[];
  public static long EXTRAKINGS_WHITE_KINGSIDE;
  public static long EXTRAKINGS_WHITE_QUEENSIDE;
  public static long EXTRAKINGS_BLACK_KINGSIDE;
  public static long EXTRAKINGS_BLACK_QUEENSIDE;
  public static long EMPTYSQUARES_WHITE_KINGSIDE;
  public static long EMPTYSQUARES_WHITE_QUEENSIDE;
  public static long EMPTYSQUARES_BLACK_KINGSIDE;
  public static long EMPTYSQUARES_BLACK_QUEENSIDE;

  // static member initialization
  static
  {
    // Build the SquareBits constants
    SquareBits = new long[ ALL_SQUARES ];
    for( int i = 0; i < ALL_SQUARES; i++ )
    {
      // Note: the 1L specifies that the 1 we are shifting is a long int
      // Java would, by default, make it a 4-byte int and be unable to
      // shift the 1 to bits 32 to 63
      SquareBits[ i ] = ( 1L << i );
    }

    // Build the extrakings constants
    EXTRAKINGS_WHITE_KINGSIDE = SquareBits[ 60 ] | SquareBits[ 61 ];
    EXTRAKINGS_WHITE_QUEENSIDE = SquareBits[ 60 ] | SquareBits[ 59 ];
    EXTRAKINGS_BLACK_KINGSIDE = SquareBits[ 4 ] | SquareBits[ 5 ];
    EXTRAKINGS_BLACK_QUEENSIDE = SquareBits[ 4 ] | SquareBits[ 3 ];
    EMPTYSQUARES_WHITE_KINGSIDE = SquareBits[ 61 ] | SquareBits[ 62 ];
    EMPTYSQUARES_WHITE_QUEENSIDE = SquareBits[ 59 ] | SquareBits[ 58 ] | SquareBits[ 57 ];
    EMPTYSQUARES_BLACK_KINGSIDE = SquareBits[ 5 ] | SquareBits[ 6 ];
    EMPTYSQUARES_BLACK_QUEENSIDE = SquareBits[ 3 ] | SquareBits[ 2 ] | SquareBits[ 1 ];

    // Build the hashing database
    HashKeyComponents = new int[ ALL_PIECES ][ ALL_SQUARES ];
    HashLockComponents = new int[ ALL_PIECES ][ ALL_SQUARES ];
    Random rnd = new Random();
    for( int i = 0; i < ALL_PIECES; i++ )
    {
      for( int j = 0; j < ALL_SQUARES; j++ )
      {
        HashKeyComponents[ i ][ j ] = rnd.nextInt();
```

```
        HashLockComponents[ i ][ j ] = rnd.nextInt();
      }
    }

    // Tokens representing the various concepts in the game, for printint
    // and file i/o purposes
    // PieceStrings contains an extra string representing empty squares
    PieceStrings = new String[ ALL_PIECES + 1 ];
    PieceStrings[ WHITE_PAWN ] = "WP";
    PieceStrings[ WHITE_ROOK ] = "WR";
    PieceStrings[ WHITE_KNIGHT ] = "WN";
    PieceStrings[ WHITE_BISHOP ] = "WB";
    PieceStrings[ WHITE_QUEEN ] = "WQ";
    PieceStrings[ WHITE_KING ] = "WK";
    PieceStrings[ BLACK_PAWN ] = "BP";
    PieceStrings[ BLACK_ROOK ] = "BR";
    PieceStrings[ BLACK_KNIGHT ] = "BN";
    PieceStrings[ BLACK_BISHOP ] = "BB";
    PieceStrings[ BLACK_QUEEN ] = "BQ";
    PieceStrings[ BLACK_KING ] = "BK";
    PieceStrings[ ALL_PIECES ] = "  ";

    // Numerical evaluation of piece material values
    PieceValues = new int[ ALL_PIECES ];
    PieceValues[ WHITE_PAWN ] = 100;
    PieceValues[ BLACK_PAWN ] = 100;
    PieceValues[ WHITE_KNIGHT ] = 300;
    PieceValues[ BLACK_KNIGHT ] = 300;
    PieceValues[ WHITE_BISHOP ] = 350;
    PieceValues[ BLACK_BISHOP ] = 350;
    PieceValues[ WHITE_ROOK ] = 500;
    PieceValues[ BLACK_ROOK ] = 500;
    PieceValues[ BLACK_QUEEN ] = 900;
    PieceValues[ WHITE_QUEEN ] = 900;
    PieceValues[ WHITE_KING ] = 2000;
    PieceValues[ BLACK_KING ] = 2000;
  }

  // The actual data representation of a chess board.  First, an array of
  // bitboards, each of which contains flags for the squares where you can
  // find a specific type of piece
  private long BitBoards[];

  // And a few other flags
  private boolean CastlingStatus[];
  private boolean HasCastled[];
  private long EnPassantPawn;

  // Whose turn is it?
  int CurrentPlayer;

/**************************************************************************
 * METHODS
 **************************************************************************/

  // Accessors
  public boolean GetCastlingStatus( int which ) { return CastlingStatus[ which ]; }
  public boolean GetHasCastled( int which ) { return HasCastled[ which ]; }
  public long GetEnPassantPawn() { return EnPassantPawn; }
  public long GetExtraKings( int side ) { return ExtraKings[ side ]; }
  public void SetExtraKings( int side, long val )
  {
    // Mark a few squares as containing "phantom kings" to detect illegal
    // castling
    ExtraKings[ side ] = val;
    BitBoards[ KING + side ] |= ExtraKings[ side ];
    BitBoards[ ALL_PIECES + side ] |= ExtraKings[ side ];
  }
  public void ClearExtraKings( int side )
  {
    BitBoards[ KING + side ] ^= ExtraKings[ side ];
```

```
    BitBoards[ ALL_PIECES + side ] ^= ExtraKings[ side ];
    // Note: one of the Extra Kings is superimposed on the rook involved in
    // the castling, so the next step is required to prevent ALL_PIECES from
    // forgetting about the rook at the same time as the phantom king
    BitBoards[ ALL_PIECES + side ] |= BitBoards[ ROOK + side ];
    ExtraKings[ side ] = 0;
  }
  public int GetCurrentPlayer()  { return CurrentPlayer; }
  public long GetBitBoard( int which ) { return BitBoards[ which ]; }

  // Look for the piece located on a specific square
  public int FindBlackPiece( int square )
  {
    // Note: we look for kings first for two reasons: because it helps
    // detect check, and because there may be a phantom king (marking an
    // illegal castling move) and a rook on the same square!
    if ( ( BitBoards[ BLACK_KING ] & SquareBits[ square ] ) != 0 )
      return BLACK_KING;
    if ( ( BitBoards[ BLACK_QUEEN ] & SquareBits[ square ] ) != 0 )
      return BLACK_QUEEN;
    if ( ( BitBoards[ BLACK_ROOK ] & SquareBits[ square ] ) != 0 )
      return BLACK_ROOK;
    if ( ( BitBoards[ BLACK_KNIGHT ] & SquareBits[ square ] ) != 0 )
      return BLACK_KNIGHT;
    if ( ( BitBoards[ BLACK_BISHOP ] & SquareBits[ square ] ) != 0 )
      return BLACK_BISHOP;
    if ( ( BitBoards[ BLACK_PAWN ] & SquareBits[ square ] ) != 0 )
      return BLACK_PAWN;
    return EMPTY_SQUARE;
  }
  public int FindWhitePiece( int square )
  {
    if ( ( BitBoards[ WHITE_KING ] & SquareBits[ square ] ) != 0 )
      return WHITE_KING;
    if ( ( BitBoards[ WHITE_QUEEN ] & SquareBits[ square ] ) != 0 )
      return WHITE_QUEEN;
    if ( ( BitBoards[ WHITE_ROOK ] & SquareBits[ square ] ) != 0 )
      return WHITE_ROOK;
    if ( ( BitBoards[ WHITE_KNIGHT ] & SquareBits[ square ] ) != 0 )
      return WHITE_KNIGHT;
    if ( ( BitBoards[ WHITE_BISHOP ] & SquareBits[ square ] ) != 0 )
      return WHITE_BISHOP;
    if ( ( BitBoards[ WHITE_PAWN ] & SquareBits[ square ] ) != 0 )
      return WHITE_PAWN;
    return EMPTY_SQUARE;
  }


  // Constructor
  public jcBoard()
  {
    BitBoards = new long[ ALL_BITBOARDS ];
    CastlingStatus = new boolean[ 4 ];
    HasCastled = new boolean[ 2 ];
    ExtraKings = new long[ 2 ];
    NumPawns = new int[ 2 ];
    MaterialValue = new int[ 2 ];
    StartingBoard();
  }

  // public boolean Clone
  // Make a deep copy of a jcBoard object; assumes that memory has already
  // been allocated for the new object, which is always true since we
  // "allocate" jcBoards from a permanent array
  public boolean Clone( jcBoard target )
  {
    EnPassantPawn = target.EnPassantPawn;
    for( int i = 0; i < 4; i++ )
    {
      CastlingStatus[ i ] = target.CastlingStatus[ i ];
    }
```

```
    for( int i = 0; i < ALL_BITBOARDS; i++ )
    {
      BitBoards[ i ] = target.BitBoards[ i ];
    }
    MaterialValue[ 0 ] = target.MaterialValue[ 0 ];
    MaterialValue[ 1 ] = target.MaterialValue[ 1 ];
    NumPawns[ 0 ] = target.NumPawns[ 0 ];
    NumPawns[ 1 ] = target.NumPawns[ 1 ];
    ExtraKings[ 0 ] = target.ExtraKings[ 0 ];
    ExtraKings[ 1 ] = target.ExtraKings[ 1 ];
    HasCastled[ 0 ] = target.HasCastled[ 0 ];
    HasCastled[ 1 ] = target.HasCastled[ 1 ];
    CurrentPlayer = target.CurrentPlayer;
    return true;
  }

  // public boolean Print
  // Display the board on standard output
  public boolean Print()
  {
    for( int line = 0; line < 8; line++ )
    {
      System.out.println( "-----------------------------------------" );
      System.out.println( "|    |    |    |    |    |    |    |    |" );
      for( int col = 0; col < 8; col++ )
      {
        long bits = SquareBits[ line * 8 + col ];

        // Scan the bitboards to find a piece, if any
        int piece = 0;
        while ( ( piece < ALL_PIECES ) && ( ( bits & BitBoards[ piece ] ) == 0 ) )
          piece++;

        // One exception: don't show the "phantom kings" which the program places
        // on the board to detect illegal attempts at castling over an attacked
        // square
        if ( ( piece == BLACK_KING ) &&
             ( ( ExtraKings[ jcPlayer.SIDE_BLACK ] & SquareBits[ line * 8 + col ] ) != 0 ) )
          piece = EMPTY_SQUARE;
        if ( ( piece == WHITE_KING ) &&
             ( ( ExtraKings[ jcPlayer.SIDE_WHITE ] & SquareBits[ line * 8 + col ] ) != 0 ) )
          piece = EMPTY_SQUARE;

        // Show the piece
        System.out.print( "| " + PieceStrings[ piece ] + " " );
      }
      System.out.println( "|" );
      System.out.println( "|    |    |    |    |    |    |    |    |" );
    }
    System.out.println( "-----------------------------------------" );
    if ( CurrentPlayer == jcPlayer.SIDE_BLACK )
      System.out.println( "NEXT MOVE: BLACK ");
    else
      System.out.println( "NEXT MOVE: WHITE" );

    return true;
  }

  // public int SwitchSides
  // Change the identity of the player to move
  public int SwitchSides()
  {
    if ( CurrentPlayer == jcPlayer.SIDE_WHITE )
      SetCurrentPlayer( jcPlayer.SIDE_BLACK );
    else
      SetCurrentPlayer( jcPlayer.SIDE_WHITE );

    return CurrentPlayer;
  }

  // public int HashKey
```

```
// Compute a 32-bit integer to represent the board, according to Zobrist[70]
public int HashKey()
{
  int hash = 0;
  // Look at all pieces, one at a time
  for( int currPiece = 0; currPiece < ALL_PIECES; currPiece++ )
  {
    long tmp = BitBoards[ currPiece ];
    // Search for all pieces on all squares.  We could optimize here: not
    // looking for pawns on the back row (or the eight row), getting out
    // of the "currSqaure" loop once we found one king of one color, etc.
    // But for simplicity's sake, we'll keep things generic.
    for( int currSquare = 0; currSquare < ALL_SQUARES; currSquare++ )
    {
      // Zobrist's method: generate a bunch of random bitfields, each
      // representing a certain "piece X is on square Y" predicate; XOR
      // the bitfields associated with predicates which are true.
      // Therefore, if we find a piece (in tmp) in a certain square,
      // we accumulate the related HashKeyComponent.
      if ( ( tmp & SquareBits[ currSquare ] ) != 0 )
        hash ^= HashKeyComponents[ currPiece ][ currSquare ];
    }
  }
  return hash;
}

// public int HashLock
// Compute a second 32-bit hash key, using an entirely different set
// piece/square components.
// This is required to be able to detect hashing collisions without
// storing an entire jcBoard in each slot of the jcTranspositionTable,
// which would gobble up inordinate amounts of memory
public int HashLock()
{
  int hash = 0;
  for( int currPiece = 0; currPiece < ALL_PIECES; currPiece++ )
  {
    long tmp = BitBoards[ currPiece ];
    for( int currSquare = 0; currSquare < ALL_SQUARES; currSquare++ )
    {
      if ( ( tmp & SquareBits[ currSquare ] ) != 0 )
        hash ^= HashLockComponents[ currPiece ][ currSquare ];
    }
  }
  return hash;
}

// public boolean ApplyMove
// Change the jcBoard's internal representation to reflect the move
// received as a parameter
public boolean ApplyMove( jcMove theMove )
{
  // If the move includes a pawn promotion, an extra step will be required
  // at the end
  boolean isPromotion = ( theMove.MoveType >= jcMove.MOVE_PROMOTION_KNIGHT );
  int moveWithoutPromotion = ( theMove.MoveType & jcMove.NO_PROMOTION_MASK );
  int side = theMove.MovingPiece % 2;

  // For now, ignore pawn promotions
  switch( moveWithoutPromotion )
  {
    case jcMove.MOVE_NORMAL:
      // The simple case
      RemovePiece( theMove.SourceSquare, theMove.MovingPiece );
      AddPiece( theMove.DestinationSquare, theMove.MovingPiece );
      break;
    case jcMove.MOVE_CAPTURE_ORDINARY:
      // Don't forget to remove the captured piece!
      RemovePiece( theMove.SourceSquare, theMove.MovingPiece );
      RemovePiece( theMove.DestinationSquare, theMove.CapturedPiece );
      AddPiece( theMove.DestinationSquare, theMove.MovingPiece );
```

```
      break;
    case jcMove.MOVE_CAPTURE_EN_PASSANT:
      // Here, we can use our knowledge of the board to make a small
      // optimization, since the pawn to be captured is always
      // "behind" the moving pawn's destination square, we can compute its
      // position on the fly
      RemovePiece( theMove.SourceSquare, theMove.MovingPiece );
      AddPiece( theMove.DestinationSquare, theMove.MovingPiece );
      if ( ( theMove.MovingPiece % 2 ) == jcPlayer.SIDE_WHITE )
        RemovePiece( theMove.DestinationSquare + 8, theMove.CapturedPiece );
      else
        RemovePiece( theMove.DestinationSquare - 8, theMove.CapturedPiece );
      break;
    case jcMove.MOVE_CASTLING_QUEENSIDE:
      // Again, we can compute the rook's source and destination squares
      // because of our knowledge of the board's structure
      RemovePiece( theMove.SourceSquare, theMove.MovingPiece );
      AddPiece( theMove.DestinationSquare, theMove.MovingPiece );
      int theRook = ROOK + ( theMove.MovingPiece % 2 );
      RemovePiece( theMove.SourceSquare - 4, theRook );
      AddPiece( theMove.SourceSquare - 1, theRook );
      // We must now mark some squares as containing "phantom kings" so that
      // the castling can be cancelled by the next opponent's move, if he
      // can move to one of them
      if ( side == jcPlayer.SIDE_WHITE )
      {
        SetExtraKings( side, EXTRAKINGS_WHITE_QUEENSIDE );
      }
      else
      {
        SetExtraKings( side, EXTRAKINGS_BLACK_QUEENSIDE );
      }
      HasCastled[ side ] = true;
      break;
    case jcMove.MOVE_CASTLING_KINGSIDE:
      // Again, we can compute the rook's source and destination squares
      // because of our knowledge of the board's structure
      RemovePiece( theMove.SourceSquare, theMove.MovingPiece );
      AddPiece( theMove.DestinationSquare, theMove.MovingPiece );
      theRook = ROOK + ( theMove.MovingPiece % 2 );
      RemovePiece( theMove.SourceSquare + 3, theRook );
      AddPiece( theMove.SourceSquare + 1, theRook );
      // We must now mark some squares as containing "phantom kings" so that
      // the castling can be cancelled by the next opponent's move, if he
      // can move to one of them
      if ( side == jcPlayer.SIDE_WHITE )
      {
        SetExtraKings( side, EXTRAKINGS_WHITE_KINGSIDE );
      }
      else
      {
        SetExtraKings( side, EXTRAKINGS_BLACK_KINGSIDE );
      }
      HasCastled[ side ] = true;
      break;
    case jcMove.MOVE_RESIGN:
      // FDL Later, ask the AI player who resigned to print the continuation
      break;
    case jcMove.MOVE_STALEMATE:
      System.out.println( "Stalemate - Game is a draw." );
      break;
  }

  // And now, apply the promotion
  if ( isPromotion )
  {
    int promotionType = ( theMove.MoveType & jcMove.PROMOTION_MASK );
    int color = ( theMove.MovingPiece % 2 );
    switch( promotionType )
    {
      case jcMove.MOVE_PROMOTION_KNIGHT:
```

```
        RemovePiece( theMove.DestinationSquare, theMove.MovingPiece );
        AddPiece( theMove.DestinationSquare, KNIGHT + color );
        break;
      case jcMove.MOVE_PROMOTION_BISHOP:
        RemovePiece( theMove.DestinationSquare, theMove.MovingPiece );
        AddPiece( theMove.DestinationSquare, BISHOP + color );
        break;
      case jcMove.MOVE_PROMOTION_ROOK:
        RemovePiece( theMove.DestinationSquare, theMove.MovingPiece );
        AddPiece( theMove.DestinationSquare, ROOK + color );
        break;
      case jcMove.MOVE_PROMOTION_QUEEN:
        RemovePiece( theMove.DestinationSquare, theMove.MovingPiece );
        AddPiece( theMove.DestinationSquare, QUEEN + color );
        break;
    }
  }

  // If this was a 2-step pawn move, we now have a valid en passant
  // capture possibility.  Otherwise, no.
  if ( ( theMove.MovingPiece == jcBoard.WHITE_PAWN ) &&
       ( theMove.SourceSquare - theMove.DestinationSquare == 16 ) )
    SetEnPassantPawn( theMove.DestinationSquare + 8 );
  else if ( ( theMove.MovingPiece == jcBoard.BLACK_PAWN ) &&
            ( theMove.DestinationSquare - theMove.SourceSquare == 16 ) )
    SetEnPassantPawn( theMove.SourceSquare + 8 );
  else
    ClearEnPassantPawn();

  // And now, maintain castling status
  // If a king moves, castling becomes impossible for that side, for the
  // rest of the game
  switch( theMove.MovingPiece )
  {
    case WHITE_KING:
      SetCastlingStatus( CASTLE_KINGSIDE + jcPlayer.SIDE_WHITE, false );
      SetCastlingStatus( CASTLE_QUEENSIDE + jcPlayer.SIDE_WHITE, false );
      break;
    case BLACK_KING:
      SetCastlingStatus( CASTLE_KINGSIDE + jcPlayer.SIDE_BLACK, false );
      SetCastlingStatus( CASTLE_QUEENSIDE + jcPlayer.SIDE_BLACK, false );
      break;
    default:
      break;
  }

  // Or, if ANYTHING moves from a corner, castling becomes impossible on
  // that side (either because it's the rook that is moving, or because
  // it has been captured by whatever moves, or because it is already gone)
  switch( theMove.SourceSquare )
  {
    case 0:
      SetCastlingStatus( CASTLE_QUEENSIDE + jcPlayer.SIDE_BLACK, false );
      break;
    case 7:
      SetCastlingStatus( CASTLE_KINGSIDE + jcPlayer.SIDE_BLACK, false );
      break;
    case 56:
      SetCastlingStatus( CASTLE_QUEENSIDE + jcPlayer.SIDE_WHITE, false );
      break;
    case 63:
      SetCastlingStatus( CASTLE_KINGSIDE + jcPlayer.SIDE_WHITE, false );
      break;
    default:
      break;
  }

  // All that remains to do is switch sides
  SetCurrentPlayer( ( GetCurrentPlayer() + 1 ) % 2 );
  return true;
}
```

```
// public boolean Load
// Load a board from a file
public boolean Load( String fileName ) throws Exception
{
  // Clean the board first
  EmptyBoard();

  // Open the file as a Java tokenizer
  FileReader fr = new FileReader( fileName );
  StreamTokenizer tok = new StreamTokenizer( fr );
  tok.eolIsSignificant( false );
  tok.lowerCaseMode( false );

  // Whose turn is it to play?
  tok.nextToken();
  if ( tok.sval.equalsIgnoreCase( jcPlayer.PlayerStrings[ jcPlayer.SIDE_WHITE ] ) )
    SetCurrentPlayer( jcPlayer.SIDE_WHITE );
  else
    SetCurrentPlayer( jcPlayer.SIDE_BLACK );

  // Read the positions of all the pieces
  // First, look for the number of pieces on the board
  tok.nextToken();
  int numPieces = (int) tok.nval;

  // Now, loop on the pieces in question
  for( int i = 0; i < numPieces; i++ )
  {
    // What kind of piece is this, and where does it go?
    tok.nextToken();
    String whichPieceStr = tok.sval;

    int whichPiece = 0;
    while ( !whichPieceStr.equalsIgnoreCase( PieceStrings[ whichPiece ] ) )
      whichPiece++;

    tok.nextToken();
    int whichSquare = (int) tok.nval;

    // Add the piece to the board
    AddPiece( whichSquare, whichPiece );
  }

  // Now, read the castling status flags
  for( int i = 0; i < 4; i++ )
  {
    tok.nextToken();
    if ( "TRUE".equalsIgnoreCase( tok.sval ) )
      SetCastlingStatus( i, true );
    else
      SetCastlingStatus( i, false );
  }

  // And finally, read the bitboard representing the position of the en
  // passant pawn, if any
  tok.nextToken();
  SetEnPassantPawn( (long) tok.nval );

  fr.close();
  return true;
}

// public boolean Save
// Save the state of the game to a file
public boolean Save( String fileName ) throws Exception
{
  // Open the file for business
  FileWriter fr = new FileWriter( fileName );
  BufferedWriter bw = new BufferedWriter( fr );
```

```
  // Whose turn is it?
  bw.write( jcPlayer.PlayerStrings[ CurrentPlayer ] );
  bw.newLine();

  // Count the pieces on the board
  int numPieces = 0;
  for( int i = 0; i < ALL_SQUARES; i++ )
  {
    if ( ( SquareBits[ i ] & BitBoards[ ALL_WHITE_PIECES ] ) != 0 )
      numPieces++;
    if ( ( SquareBits[ i ] & BitBoards[ ALL_BLACK_PIECES ] ) != 0 )
      numPieces++;
  }
  bw.write( String.valueOf( numPieces ) );
  bw.newLine();

  // Dump the pieces, one by one
  for( int piece = 0; piece < ALL_PIECES; piece++ )
  {
    for( int square = 0; square < ALL_SQUARES; square++ )
    {
      if ( ( BitBoards[ piece ] & SquareBits[ square ] ) != 0 )
      {
        bw.write( PieceStrings[ piece ] + " " + String.valueOf( square ) );
        bw.newLine();
      }
    }
  }

  // And finally, dump the castling status and the en passant pawn
  for( int i = 0; i < 4; i++ )
  {
    if ( CastlingStatus[ i ] )
      bw.write( "TRUE" );
    else
      bw.write( "FALSE" );
    bw.newLine();
  }

  bw.write( String.valueOf( EnPassantPawn ) );

  bw.close();
  return true;
}

// public int EvalMaterial
// Compute the board's material balance, from the point of view of the "side"
// player.  This is an exact clone of the eval function in CHESS 4.5
public int EvalMaterial( int side )
{
  // If both sides are equal, no need to compute anything!
  if ( MaterialValue[ jcPlayer.SIDE_BLACK ] == MaterialValue[ jcPlayer.SIDE_WHITE ] )
    return 0;

  int otherSide = ( side + 1 ) % 2;
  int matTotal = MaterialValue[ side ] + MaterialValue[ otherSide ];

  // Who is leading the game, material-wise?
  if ( MaterialValue[ jcPlayer.SIDE_BLACK ] > MaterialValue[ jcPlayer.SIDE_WHITE ] )
  {
    // Black leading
    int matDiff = MaterialValue[ jcPlayer.SIDE_BLACK ] - MaterialValue[ jcPlayer.SIDE_WHITE ];
    int val = Math.min( 2400, matDiff ) +
              ( matDiff * ( 12000 - matTotal ) * NumPawns[ jcPlayer.SIDE_BLACK ] )
              / ( 6400 * ( NumPawns[ jcPlayer.SIDE_BLACK ] + 1 ) );
    if ( side == jcPlayer.SIDE_BLACK )
      return val;
    else
      return -val;
  }
  else
```

```
    {
      // White leading
      int matDiff = MaterialValue[ jcPlayer.SIDE_WHITE ] - MaterialValue[ jcPlayer.SIDE_BLACK ];
      int val = Math.min( 2400, matDiff ) +
                ( matDiff * ( 12000 - matTotal ) * NumPawns[ jcPlayer.SIDE_WHITE ] )
                / ( 6400 * ( NumPawns[ jcPlayer.SIDE_WHITE ] + 1 ) );

      if ( side == jcPlayer.SIDE_WHITE )
        return val;
      else
        return -val;
    }
  }

  // public boolean StartingBoard
  // Restore the board to a game-start position
  public boolean StartingBoard()
  {
    // Put the pieces on the board
    EmptyBoard();
    AddPiece( 0, BLACK_ROOK );
    AddPiece( 1, BLACK_KNIGHT );
    AddPiece( 2, BLACK_BISHOP );
    AddPiece( 3, BLACK_QUEEN );
    AddPiece( 4, BLACK_KING );
    AddPiece( 5, BLACK_BISHOP );
    AddPiece( 6, BLACK_KNIGHT );
    AddPiece( 7, BLACK_ROOK );
    for( int i = 8; i < 16; i++ )
    {
      AddPiece( i, BLACK_PAWN );
    }

    for( int i = 48; i < 56; i++ )
    {
      AddPiece( i, WHITE_PAWN );
    }
    AddPiece( 56, WHITE_ROOK );
    AddPiece( 57, WHITE_KNIGHT );
    AddPiece( 58, WHITE_BISHOP );
    AddPiece( 59, WHITE_QUEEN );
    AddPiece( 60, WHITE_KING );
    AddPiece( 61, WHITE_BISHOP );
    AddPiece( 62, WHITE_KNIGHT );
    AddPiece( 63, WHITE_ROOK );

    // And allow all castling moves
    for( int i = 0; i < 4; i++ )
    {
      CastlingStatus[ i ] = true;
    }
    HasCastled[ 0 ] = false;
    HasCastled[ 1 ] = false;
    ClearEnPassantPawn();

    // And ask White to play the first move
    SetCurrentPlayer( jcPlayer.SIDE_WHITE );
    return true;
  }

/*****************************************************************************
 * PRIVATE METHODS
 *****************************************************************************/

  // private boolean AddPiece
  // Place a specific piece on a specific board square
  private boolean AddPiece( int whichSquare, int whichPiece )
  {
    // Add the piece itself
    BitBoards[ whichPiece ] |= SquareBits[ whichSquare ];
```

```
    // And note the new piece position in the bitboard containing all
    // pieces of its color.  Here, we take advantage of the fact that
    // all pieces of a given color are represented by numbers of the same
    // parity
    BitBoards[ ALL_PIECES + ( whichPiece % 2 ) ] |= SquareBits[ whichSquare ];

    // And adjust material balance accordingly
    MaterialValue[ whichPiece % 2 ] += PieceValues[ whichPiece ];
    if ( whichPiece == WHITE_PAWN )
      NumPawns[ jcPlayer.SIDE_WHITE ]++;
    else if ( whichPiece == BLACK_PAWN )
      NumPawns[ jcPlayer.SIDE_BLACK ]++;

    return true;
  }

  // private boolean RemovePiece
  // Eliminate a specific piece from a specific square on the board
  // Note that you MUST know that the piece is there before calling this,
  // or the results will not be what you expect!
  private boolean RemovePiece( int whichSquare, int whichPiece )
  {
    // Remove the piece itself
    BitBoards[ whichPiece ] ^= SquareBits[ whichSquare ];
    BitBoards[ ALL_PIECES + ( whichPiece % 2 ) ] ^= SquareBits[ whichSquare ];

    // And adjust material balance accordingly
    MaterialValue[ whichPiece % 2 ] -= PieceValues[ whichPiece ];
    if ( whichPiece == WHITE_PAWN )
      NumPawns[ jcPlayer.SIDE_WHITE ]--;
    else if ( whichPiece == BLACK_PAWN )
      NumPawns[ jcPlayer.SIDE_BLACK ]--;
    return true;
  }


  // private boolean EmptyBoard
  // Remove every piece from the board
  private boolean EmptyBoard()
  {
    for( int i = 0; i < ALL_BITBOARDS; i++ )
    {
      BitBoards[ i ] = 0;
    }
    ExtraKings[ 0 ] = 0;
    ExtraKings[ 1 ] = 0;
    EnPassantPawn = 0;
    MaterialValue[ 0 ] = 0;
    MaterialValue[ 1 ] = 0;
    NumPawns[ 0 ] = 0;
    NumPawns[ 1 ] = 0;
    return true;
  }

  // private boolean SetCastlingStatus
  // Change one of the "castling status" flags
  // parameter whichFlag should be a sum of a side marker and a castling
  // move identifier, for example, jcPlayer.SIDE_WHITE + CASTLE_QUEENSIDE
  private boolean SetCastlingStatus( int whichFlag, boolean newValue )
  {
    CastlingStatus[ whichFlag ] = newValue;
    return true;
  }

  // private boolean SetEnPassantPawn
  // If a pawn move has just made en passant capture possible, mark it as
  // such in a bitboard (containing the en passant square only)
  private boolean SetEnPassantPawn( int square )
  {
    ClearEnPassantPawn();
    EnPassantPawn |= SquareBits[ square ];
```

```
    return true;
  }

  private boolean SetEnPassantPawn( long bitboard )
  {
    EnPassantPawn = bitboard;
    return true;
  }

  // private boolean ClearEnPassantPawn
  // Indicates that there is no en passant square at all.  Technically, this
  // job could have been handled by SetEnPassaantPawn( long ) with a null
  // parameter, but I have chosen to add a method to avoid problems if I ever
  // forgot to specify 0L: using 0 would call the first form of the Set method
  // and indicate an en passant pawn in a corner of the board, with possibly
  // disastrous consequences!
  private boolean ClearEnPassantPawn()
  {
    EnPassantPawn = 0;
    return true;
  }

  // private boolean SetCurrentPlayer
  // Whose turn is it?
  private boolean SetCurrentPlayer( int which )
  {
    CurrentPlayer = which;
    return true;
  }
}
```

```
/*****************************************************************************
 * jcMove.java - An encapsulation of a chess move and its consequences
 * by François Dominic Laramée
 *
 * Purpose: This class is used all over the place.  It contains a move's
 * source and target squares, a type identifier (i.e., a normal move, a pawn
 * promotion, etc.) and a score, whether an actual evaluation of the position
 * which would result from the move or a value taken from the history table.
 *
 * History
 * 11.06.00 Creation
 * 09.07.00 Added fields MovingPiece and CapturedPiece; while not absolutely
 *          needed, they do accelerate move processing and help to make code
 *          easier to understand, so I gladly keep them around as optimizations
 * 14.08.00 Added "search depth" field, so that we can determine whether a
 *          transposition table entry should be used or not.
 *****************************************************************************/
package javachess;

public class jcMove
{
  /***************************************************************************
   * CONSTANTS
   ***************************************************************************/

  // The different types of moves recognized by the game
  public static final int MOVE_NORMAL = 0;
  public static final int MOVE_CAPTURE_ORDINARY = 1;
  public static final int MOVE_CAPTURE_EN_PASSANT = 2;
  public static final int MOVE_CASTLING_KINGSIDE = 4;
  public static final int MOVE_CASTLING_QUEENSIDE = 8;
  public static final int MOVE_RESIGN = 16;
  public static final int MOVE_STALEMATE = 17;
  public static final int MOVE_PROMOTION_KNIGHT = 32;
  public static final int MOVE_PROMOTION_BISHOP = 64;
  public static final int MOVE_PROMOTION_ROOK = 128;
  public static final int MOVE_PROMOTION_QUEEN = 256;

  // A pair of masks used to split the promotion and the non-promotion part of
  // a move type ID
  public static final int PROMOTION_MASK = 480;
  public static final int NO_PROMOTION_MASK = 31;

  // Alphabeta may return an actual move potency evaluation, or an upper or
  // lower bound only (in case a cutoff happens).  We need to store this
  // information in the transposition table to make sure that a given
  // value is actually useful in given circumstances.
  public static final int EVALTYPE_ACCURATE = 0;
  public static final int EVALTYPE_UPPERBOUND = 1;
  public static final int EVALTYPE_LOWERBOUND = 2;

  // A sentinel value used to identify jcMove fields without valid data
  public static final int NULL_MOVE = -1;

  /***************************************************************************
   * DATA MEMBERS
   * Note: this class is intended as a C++ structure, so all data members
   * have public access.
   ***************************************************************************/

  // The moving piece; one of the constants defined by jcBoard
  public int MovingPiece;

  // The piece being captured by this move, if any; another jcBoard constant
  public int CapturedPiece;

  // The squares involved in the move
  public int SourceSquare, DestinationSquare;

  // A type ID: is this a regular move, a capture, a capture AND promotion from
  // Pawn to Rook, etc.  Move generation determines this, by definition; storing
```

```
// it here avoids having to "re-discover" the information in jcBoard.ApplyMove
// at the cost of a few bytes
public int MoveType;

// An evaluation of the move's potency, either as a result of an alphabeta
// search of some kind or of a retrieval in the transposition table
public int MoveEvaluation;
public int MoveEvaluationType;
public int SearchDepth;

/**************************************************************************
 * PUBLIC METHODS
 **************************************************************************/

public jcMove()
{
  this.Reset();
}

public void Copy( jcMove target )
{
  MovingPiece = target.MovingPiece;
  CapturedPiece = target.CapturedPiece;
  SourceSquare = target.SourceSquare;
  DestinationSquare = target.DestinationSquare;
  MoveType = target.MoveType;
  MoveEvaluation = target.MoveEvaluation;
  MoveEvaluationType = target.MoveEvaluationType;
  SearchDepth = target.SearchDepth;
}

// public boolean Equals( jcMove target )
// Check whether two jcMove objects contain the same data (not necessarily
// whether they are the same object in memory)
public boolean Equals( jcMove target )
{
  if ( MovingPiece != target.MovingPiece )
    return false;
  if ( CapturedPiece != target.CapturedPiece )
    return false;
  if ( MoveType != target.MoveType )
    return false;
  if ( SourceSquare != target.SourceSquare )
    return false;
  if ( DestinationSquare != target.DestinationSquare )
    return false;
  return true;
}

public boolean Reset()
{
  MovingPiece = jcBoard.EMPTY_SQUARE;
  CapturedPiece = jcBoard.EMPTY_SQUARE;
  SourceSquare = NULL_MOVE;
  DestinationSquare = NULL_MOVE;
  MoveType = NULL_MOVE;
  MoveEvaluation = NULL_MOVE;
  MoveEvaluationType = NULL_MOVE;
  SearchDepth = NULL_MOVE;
  return true;
}

public void Print()
{
  System.out.print( "Move: " );
  if ( MoveType == MOVE_STALEMATE )
  {
    System.out.println( "STALEMATE!!!" );
  }
  if ( MoveType != MOVE_RESIGN )
  {
```

```
      System.out.print( jcBoard.PieceStrings[ MovingPiece ] );
      System.out.print( "   [ " );
      System.out.print( SourceSquare );
      System.out.print( ", " );
      System.out.print( DestinationSquare );
      System.out.print( " ] TYPE: " );
      System.out.println( MoveType );
    }
    else
    {
      System.out.println( "RESIGNATION!" );
    }
  }
}
```

```
/*****************************************************************************
 * jcMoveListGenerator.java - Find all pseudo-legal moves given a board state
 * by F.D. Laramée
 *
 * Purpose: Identify a list of possible moves
 *
 * History:
 * 27.07.00 Creation
 *****************************************************************************/

package javachess;
import javachess.jcBoard;
import javachess.jcMove;
import java.util.*;

public class jcMoveListGenerator
{

  /*****************************************************************************
   * INSTANCE VARIABLES
   *****************************************************************************/

  // The list of moves, implemented as a java collection class, namely the
  // ArrayList (dynamic array)
  ArrayList Moves;
  Iterator MovesIt;

  /*****************************************************************************
   * PUBLIC METHODS
   *****************************************************************************/

  // Construction
  public jcMoveListGenerator()
  {
    Moves = new ArrayList( 10 );
    MovesIt = null;
    ResetIterator();
  }


  // public void ResetIterator
  // Prepare an iterator for scanning through the list of moves
  public void ResetIterator()
  {
    // Mark the old iterator, if any, for garbage collection
    if ( MovesIt != null )
      MovesIt = null;

    // Make a new iterator ready for scanning
    MovesIt = Moves.iterator();
  }

  // Accessors
  public ArrayList GetMoveList() { return Moves; }
  public int Size() { return Moves.size(); }

  // public boolean Find( jcMove mov )
  // Look for a specific move in the list; if it is there, return true
  // This is used by the jcPlayerHuman object, to verify whether a move entered
  // by the player is actually valid
  public boolean Find( jcMove mov )
  {
    ResetIterator();
    jcMove testMove;
    while( ( testMove = Next() ) != null )
    {
      if ( mov.Equals( testMove ) )
        return true;
    }
    return false;
  }
```

```
// public jcMove FindMoveForSquares( int source, int dest )
// look for a move from "source" to "dest" in the list
public jcMove FindMoveForSquares( int source, int dest )
{
  ResetIterator();
  jcMove testMove;
  while( ( testMove = Next() ) != null )
  {
    if ( ( testMove.SourceSquare == source ) && ( testMove.DestinationSquare == dest ) )
      return testMove;
  }
  return null;
}

// public jcMove Next()
// Find the next move in the list, if any
public jcMove Next()
{
  if ( MovesIt.hasNext() )
    return (jcMove) MovesIt.next();
  else
    return null;
}

// public boolean ComputeLegalMoves
// Look at the board received as a parameter, and build a list of legal
// moves which can be derived from it.  If there are no legal moves, or if
// one of the moves is a king capture (which means that the opponent's
// previous move left the king in check, which is illegal), return false.
public boolean ComputeLegalMoves( jcBoard theBoard )
{
  // First, clean up the old list of moves, if any
  Moves.clear();

  // Now, compute the moves, one piece type at a time
  if ( theBoard.GetCurrentPlayer() == jcPlayer.SIDE_WHITE )
  {
    // Clean up the data structures indicating that the last white move
    // was a castling, if any
    if ( theBoard.GetExtraKings( jcPlayer.SIDE_WHITE ) != 0 )
    {
      theBoard.ClearExtraKings( jcPlayer.SIDE_WHITE );
    }
    // Check for white moves, one piece type at a time
    // if any one type can capture the king, stop the work immediately
    // because the board position is illegal
    if ( !ComputeWhiteQueenMoves( theBoard ) ) return false;
    if ( !ComputeWhiteKingMoves( theBoard ) ) return false;
    if ( !ComputeWhiteRookMoves( theBoard, jcBoard.WHITE_ROOK ) ) return false;
    if ( !ComputeWhiteBishopMoves( theBoard, jcBoard.WHITE_BISHOP ) ) return false;
    if ( !ComputeWhiteKnightMoves( theBoard ) ) return false;
    if ( !ComputeWhitePawnMoves( theBoard ) ) return false;
  }
  else  // Compute Black's moves
  {
    if ( theBoard.GetExtraKings( jcPlayer.SIDE_BLACK ) != 0 )
    {
      theBoard.ClearExtraKings( jcPlayer.SIDE_BLACK );
    }
    if ( !ComputeBlackQueenMoves( theBoard ) ) return false;
    if ( !ComputeBlackKingMoves( theBoard ) ) return false;
    if ( !ComputeBlackRookMoves( theBoard, jcBoard.BLACK_ROOK ) ) return false;
    if ( !ComputeBlackBishopMoves( theBoard, jcBoard.BLACK_BISHOP ) ) return false;
    if ( !ComputeBlackKnightMoves( theBoard ) ) return false;
    if ( !ComputeBlackPawnMoves( theBoard ) ) return false;
  }

  // And finally, if there are no pseudo-legal moves at all, we have an
  // obvious error (there are no pieces on the board!); flag the condition
  if ( Moves.size() == 0 )
```

```
    return false;
  else
  {
    ResetIterator();
    return true;
  }
}


// public boolean ComputeQuiescenceMoves
// Find only the moves which are relevant to quiescence search; i.e., captures
public boolean ComputeQuiescenceMoves( jcBoard theBoard )
{
  ComputeLegalMoves( theBoard );
  for( int i = Moves.size() - 1; i >= 0; i-- )
  {
    jcMove mov = (jcMove) Moves.get( i );
    if ( ( mov.MoveType != jcMove.MOVE_CAPTURE_ORDINARY ) &&
         ( mov.MoveType != jcMove.MOVE_CAPTURE_EN_PASSANT ) )
      Moves.remove( i );
  }
  ResetIterator();
  return( Moves.size() > 0 );
}

// public void Print()
// Dump the move list to standard output, for debugging purposes
public void Print()
{
  // Do not use the iterator, to avoid messing up a regular operation!
  for( int it = 0; it < Moves.size(); it++ )
  {
    jcMove mov = (jcMove) Moves.get( it );
    mov.Print();
  }
}


/**************************************************************************
 * PRIVATE METHODS
 * For move generation
 **************************************************************************/

 private boolean ComputeWhiteQueenMoves( jcBoard theBoard )
 {
   if ( !ComputeWhiteBishopMoves( theBoard, jcBoard.WHITE_QUEEN ) ) return false;
   if ( !ComputeWhiteRookMoves( theBoard, jcBoard.WHITE_QUEEN ) ) return false;
   return true;
 }

 private boolean ComputeWhiteKingMoves( jcBoard theBoard )
 {
   // Fetch the bitboard containing position of the king
   long pieces = theBoard.GetBitBoard( jcBoard.WHITE_KING );

   // Find it!  There is only one king, so look for it and stop
   int square;
   for( square = 0; square < 64; square++ )
   {
     if ( ( jcBoard.SquareBits[ square ] & pieces ) != 0 )
       break;
   }

   // Find its moves
   for( int i = 0; i < KingMoves[ square ].length; i++ )
   {
     // Get the destination square
     int dest = KingMoves[ square ][ i ];

     // Is it occupied by a friendly piece?  If so, can't move there
     if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
```

```
            jcBoard.SquareBits[ dest ] ) != 0 )
          continue;

      // Otherwise, the move is legal, so we must prepare to add it
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_KING;

      // Is the destination occupied by an enemy?  If so, we have a capture
      if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
           jcBoard.SquareBits[ dest ] ) != 0 )
      {
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
        mov.CapturedPiece = theBoard.FindBlackPiece( dest );

        // If the piece we find is a king, abort because the board
        // position is illegal!
        if ( mov.CapturedPiece == jcBoard.BLACK_KING )
        {
           return false;
        }
      }

      // otherwise, it is a simple move
      else
      {
        mov.MoveType = jcMove.MOVE_NORMAL;
        mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
      }

      // And we add the move to the list
      Moves.add( mov );
    }

    // Now, let's consider castling...
    // Kingside first
    if ( theBoard.GetCastlingStatus( jcBoard.CASTLE_KINGSIDE + jcPlayer.SIDE_WHITE ) )
    {
      // First, check whether there are empty squares between king and rook
      if ( ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.EMPTYSQUARES_WHITE_KINGSIDE ) == 0 ) &&
           ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.EMPTYSQUARES_WHITE_KINGSIDE ) == 0 ) )
      {
        jcMove mov = new jcMove();
        mov.MovingPiece = jcBoard.WHITE_KING;
        mov.SourceSquare = 60;
        mov.DestinationSquare = 62;
        mov.MoveType = jcMove.MOVE_CASTLING_KINGSIDE;
        mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
        Moves.add( mov );
      }
    }
    if ( theBoard.GetCastlingStatus( jcBoard.CASTLE_QUEENSIDE + jcPlayer.SIDE_WHITE ) )
    {
      if ( ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.EMPTYSQUARES_WHITE_QUEENSIDE ) == 0 ) &&
           ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.EMPTYSQUARES_WHITE_QUEENSIDE ) == 0 ) )
      {
        jcMove mov = new jcMove();
        mov.MovingPiece = jcBoard.WHITE_KING;
        mov.SourceSquare = 60;
        mov.DestinationSquare = 58;
        mov.MoveType = jcMove.MOVE_CASTLING_QUEENSIDE;
        mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
        Moves.add( mov );
      }
    }
    return true;
```

```
}

// private boolean ComputeWhiteRookMoves
// Receives an extra "pieceType" parameter, because the queen AND the rook
// need to use this function
private boolean ComputeWhiteRookMoves( jcBoard theBoard, int pieceType )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( pieceType );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a white piece, so let's start looking at the bottom
  // of the board
  for( int square = 63; square >= 0; square-- )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int ray = 0; ray < RookMoves[ square ].length; ray++ )
      {
        for( int i = 0; i < RookMoves[ square ][ ray ].length; i++ )
        {
          // Get the destination square
          int dest = RookMoves[ square ][ ray ][ i ];

          // Is it occupied by a friendly piece?  If so, can't move there
          // AND we must discontinue the current ray
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
            break;

          // Otherwise, the move is legal, so we must prepare to add it
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MovingPiece = pieceType;

          // Is the destination occupied by an enemy?  If so, we have a capture
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
          {
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
            mov.CapturedPiece = theBoard.FindBlackPiece( dest );

            // If the piece we find is a king, abort because the board
            // position is illegal!
            if ( mov.CapturedPiece == jcBoard.BLACK_KING )
            {
              return false;
            }

            Moves.add( mov );
            break;
          }
          // otherwise, it is a simple move
          else
          {
            mov.MoveType = jcMove.MOVE_NORMAL;
            mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
            Moves.add( mov );
          }
        }
      }
      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
```

```
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
   return true;
}

private boolean ComputeWhiteBishopMoves( jcBoard theBoard, int pieceType )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( pieceType );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a white piece, so let's start looking at the bottom
  // of the board
  for( int square = 63; square >= 0; square-- )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int ray = 0; ray < BishopMoves[ square ].length; ray++ )
      {
        for( int i = 0; i < BishopMoves[ square ][ ray ].length; i++ )
        {
          // Get the destination square
          int dest = BishopMoves[ square ][ ray ][ i ];

          // Is it occupied by a friendly piece?  If so, can't move there
          // AND we must discontinue the current ray
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
            break;

          // Otherwise, the move is legal, so we must prepare to add it
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MovingPiece = pieceType;

          // Is the destination occupied by an enemy?  If so, we have a capture
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
          {
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
            mov.CapturedPiece = theBoard.FindBlackPiece( dest );

            // If the piece we find is a king, abort because the board
            // position is illegal!
            if ( mov.CapturedPiece == jcBoard.BLACK_KING )
            {
              return false;
            }

            Moves.add( mov );
            break;
          }
          // otherwise, it is a simple move
          else
          {
            mov.MoveType = jcMove.MOVE_NORMAL;
            mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
            Moves.add( mov );
```

```
          }
        }
      }
      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
  return true;
}

private boolean ComputeWhiteKnightMoves( jcBoard theBoard )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( jcBoard.WHITE_KNIGHT );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a white piece, so let's start looking at the bottom
  // of the board
  for( int square = 63; square >= 0; square-- )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int i = 0; i < KnightMoves[ square ].length; i++ )
      {
        // Get the destination square
        int dest = KnightMoves[ square ][ i ];

        // Is it occupied by a friendly piece?  If so, can't move there
        if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
             jcBoard.SquareBits[ dest ] ) != 0 )
          continue;

        // Otherwise, the move is legal, so we must prepare to add it
        jcMove mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.WHITE_KNIGHT;

        // Is the destination occupied by an enemy?  If so, we have a capture
        if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
             jcBoard.SquareBits[ dest ] ) != 0 )
        {
          mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
          mov.CapturedPiece = theBoard.FindBlackPiece( dest );

          // If the piece we find is a king, abort because the board
          // position is illegal!
          if ( mov.CapturedPiece == jcBoard.BLACK_KING )
          {
            return false;
          }
        }
        // otherwise, it is a simple move
        else
        {
          mov.MoveType = jcMove.MOVE_NORMAL;
          mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
        }
```

```
        // And we add the move to the list
        Moves.add( mov );
      }

      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
  return true;
}

private boolean ComputeWhitePawnMoves( jcBoard theBoard )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( jcBoard.WHITE_PAWN );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // a small optimization
  long allPieces = theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) |
                   theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES );

  // This is a white piece, so let's start looking at the bottom
  // of the board... But only consider positions where a pawn can
  // actually dwell!
  int dest;
  for( int square = 55; square >= 8; square-- )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) == 0 )
      continue;

    // First, try a normal pawn pushing
    dest = square - 8;
    if ( ( allPieces & jcBoard.SquareBits[ dest ] ) == 0 )
    {
      // Unless this push results in a promotion...
      if ( square > 15 )
      {
        jcMove mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.WHITE_PAWN;
        mov.MoveType = jcMove.MOVE_NORMAL;
        Moves.add( mov );

        // Is there a chance to perform a double push? Only if the piece
        // is in its original square
        if ( square >= 48 )
        {
          dest -= 8;
          if ( ( allPieces & jcBoard.SquareBits[ dest ] ) == 0 )
          {
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.WHITE_PAWN;
            mov.MoveType = jcMove.MOVE_NORMAL;
            Moves.add( mov );
          }
```

```
    }
  }
  else  // if square < 16
  {
    // We are now looking at pawn promotion!
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_QUEEN + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_KNIGHT + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_ROOK + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_BISHOP + jcMove.MOVE_NORMAL;
    Moves.add( mov );

  }
}

// Now, let's try a capture
// Three cases: the pawn is on the 1st file, the 8th file, or elsewhere
if ( ( square % 8 ) == 0 )
{
  dest = square - 7;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest < 8 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.CapturedPiece = theBoard.FindBlackPiece( dest );
    Moves.add( mov );

    // Other promotion captures
    if ( dest < 8 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
```

```
        mov.MovingPiece = jcBoard.WHITE_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
        mov.CapturedPiece = theBoard.FindBlackPiece( dest );
        Moves.add( mov );
      }
    }
    // Now, try an en passant capture
    else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
      mov.CapturedPiece = jcBoard.BLACK_PAWN;
      Moves.add( mov );
    }
  }
  else if ( ( square % 8 ) == 7 )
  {
    dest = square - 9;
    // Try an ordinary capture first
    if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
      if ( dest < 8 )
        mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
      // Other promotion captures
      if ( dest < 8 )
      {
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.WHITE_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
        mov.CapturedPiece = theBoard.FindBlackPiece( dest );
        Moves.add( mov );
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.WHITE_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
        mov.CapturedPiece = theBoard.FindBlackPiece( dest );
        Moves.add( mov );
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.WHITE_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
        mov.CapturedPiece = theBoard.FindBlackPiece( dest );
        Moves.add( mov );
      }
    }
    // Now, try an en passant capture
    else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.CapturedPiece = jcBoard.BLACK_PAWN;
      Moves.add( mov );
    }
  }
```

```
else
{
  dest = square - 7;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest < 8 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.CapturedPiece = theBoard.FindBlackPiece( dest );
    Moves.add( mov );

    // Other promotion captures
    if ( dest < 8 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.WHITE_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
      mov.CapturedPiece = theBoard.FindBlackPiece( dest );
      Moves.add( mov );
    }
  }
  // Now, try an en passant capture
  else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
    mov.CapturedPiece = jcBoard.BLACK_PAWN;
    Moves.add( mov );
  }
  dest = square - 9;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest < 8 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.MovingPiece = jcBoard.WHITE_PAWN;
    mov.CapturedPiece = theBoard.FindBlackPiece( dest );
    Moves.add( mov );
    // Other promotion captures
    if ( dest < 8 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
```

```
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.WHITE_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
            mov.CapturedPiece = theBoard.FindBlackPiece( dest );
            Moves.add( mov );
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.WHITE_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
            mov.CapturedPiece = theBoard.FindBlackPiece( dest );
            Moves.add( mov );
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.WHITE_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
            mov.CapturedPiece = theBoard.FindBlackPiece( dest );
            Moves.add( mov );
          }
        }
        // Now, try an en passant capture
        else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
        {
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
          mov.MovingPiece = jcBoard.WHITE_PAWN;
          mov.CapturedPiece = jcBoard.BLACK_PAWN;
          Moves.add( mov );
        }
      }

      // And perform the usual trick to abort the loop when we no longer
      // have any pieces to look for
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;

    }
    return true;
  }


  private boolean ComputeBlackQueenMoves( jcBoard theBoard )
  {
    if ( !ComputeBlackRookMoves( theBoard, jcBoard.BLACK_QUEEN ) ) return false;
    if ( !ComputeBlackBishopMoves( theBoard, jcBoard.BLACK_QUEEN ) ) return false;
    return true;
  }

  private boolean ComputeBlackKingMoves( jcBoard theBoard )
  {
    // Fetch the bitboard containing position of the king
    long pieces = theBoard.GetBitBoard( jcBoard.BLACK_KING );

    // Find it!  There is only one king, so look for it and stop
    int square;
    for( square = 0; square < 64; square++ )
    {
      if ( ( jcBoard.SquareBits[ square ] & pieces ) != 0 )
        break;
    }

    // Find its moves
    for( int i = 0; i < KingMoves[ square ].length; i++ )
    {
      // Get the destination square
      int dest = KingMoves[ square ][ i ];
```

```
  // Is it occupied by a friendly piece?  If so, can't move there
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
        jcBoard.SquareBits[ dest ] ) != 0 )
     continue;

  // Otherwise, the move is legal, so we must prepare to add it
  jcMove mov = new jcMove();
  mov.SourceSquare = square;
  mov.DestinationSquare = dest;
  mov.MovingPiece = jcBoard.BLACK_KING;

  // Is the destination occupied by an enemy?  If so, we have a capture
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
       jcBoard.SquareBits[ dest ] ) != 0 )
  {
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    mov.CapturedPiece = theBoard.FindWhitePiece( dest );

    // If the piece we find is a king, abort because the board
    // position is illegal!
    if ( mov.CapturedPiece == jcBoard.WHITE_KING )
    {
       return false;
    }
  }

  // otherwise, it is a simple move
  else
  {
    mov.MoveType = jcMove.MOVE_NORMAL;
    mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
  }

  // And we add the move to the list
  Moves.add( mov );
}

// Now, let's consider castling...
// Kingside first
if ( theBoard.GetCastlingStatus( jcBoard.CASTLE_KINGSIDE + jcPlayer.SIDE_BLACK ) )
{
  // First, check whether there are empty squares between king and rook
  if ( ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
           jcBoard.EMPTYSQUARES_BLACK_KINGSIDE ) == 0 ) &&
       ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
           jcBoard.EMPTYSQUARES_BLACK_KINGSIDE ) == 0 ) )
  {
    jcMove mov = new jcMove();
    mov.MovingPiece = jcBoard.BLACK_KING;
    mov.SourceSquare = 4;
    mov.DestinationSquare = 6;
    mov.MoveType = jcMove.MOVE_CASTLING_KINGSIDE;
    mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
    Moves.add( mov );
  }
}
if ( theBoard.GetCastlingStatus( jcBoard.CASTLE_QUEENSIDE + jcPlayer.SIDE_BLACK ) )
{
  if ( ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
           jcBoard.EMPTYSQUARES_BLACK_QUEENSIDE ) == 0 ) &&
       ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
           jcBoard.EMPTYSQUARES_BLACK_QUEENSIDE ) == 0 ) )
  {
    jcMove mov = new jcMove();
    mov.MovingPiece = jcBoard.BLACK_KING;
    mov.SourceSquare = 4;
    mov.DestinationSquare = 2;
    mov.MoveType = jcMove.MOVE_CASTLING_QUEENSIDE;
    mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
    Moves.add( mov );
  }
```

```
  }
  return true;
}

private boolean ComputeBlackRookMoves( jcBoard theBoard, int pieceType )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( pieceType );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a black piece, so let's start looking at the top
  // of the board
  for( int square = 0; square < 64; square++ )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int ray = 0; ray < RookMoves[ square ].length; ray++ )
      {
        for( int i = 0; i < RookMoves[ square ][ ray ].length; i++ )
        {
          // Get the destination square
          int dest = RookMoves[ square ][ ray ][ i ];

          // Is it occupied by a friendly piece?  If so, can't move there
          // AND we must discontinue the current ray
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
            break;

          // Otherwise, the move is legal, so we must prepare to add it
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MovingPiece = pieceType;

          // Is the destination occupied by an enemy?  If so, we have a capture
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
          {
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
            mov.CapturedPiece = theBoard.FindWhitePiece( dest );

            // If the piece we find is a king, abort because the board
            // position is illegal!
            if ( mov.CapturedPiece == jcBoard.WHITE_KING )
            {
              return false;
            }

            Moves.add( mov );
            break;
          }
          // otherwise, it is a simple move
          else
          {
            mov.MoveType = jcMove.MOVE_NORMAL;
            mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
            Moves.add( mov );
          }
        }
      }
      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
      pieces ^= jcBoard.SquareBits[ square ];
```

```
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
  return true;
}

private boolean ComputeBlackBishopMoves( jcBoard theBoard, int pieceType )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( pieceType );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a black piece, so let's start looking at the top
  // of the board
  for( int square = 0; square < 64; square++ )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int ray = 0; ray < BishopMoves[ square ].length; ray++ )
      {
        for( int i = 0; i < BishopMoves[ square ][ ray ].length; i++ )
        {
          // Get the destination square
          int dest = BishopMoves[ square ][ ray ][ i ];

          // Is it occupied by a friendly piece?  If so, can't move there
          // AND we must discontinue the current ray
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
            break;

          // Otherwise, the move is legal, so we must prepare to add it
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MovingPiece = pieceType;

          // Is the destination occupied by an enemy?  If so, we have a capture
          if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
               jcBoard.SquareBits[ dest ] ) != 0 )
          {
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
            mov.CapturedPiece = theBoard.FindWhitePiece( dest );

            // If the piece we find is a king, abort because the board
            // position is illegal!
            if ( mov.CapturedPiece == jcBoard.WHITE_KING )
            {
              return false;
            }

            // Otherwise, add the move to the list and interrupt the ray
            Moves.add( mov );
            break;
          }
          // otherwise, it is a simple move
          else
          {
            mov.MoveType = jcMove.MOVE_NORMAL;
            mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
            Moves.add( mov );
```

```
          }
        }
      }
      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
  return true;
}

private boolean ComputeBlackKnightMoves( jcBoard theBoard )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( jcBoard.BLACK_KNIGHT );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // This is a black piece, so let's start looking at the top
  // of the board
  for( int square = 0; square < 64; square++ )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // There is a piece here; find its moves
      for( int i = 0; i < KnightMoves[ square ].length; i++ )
      {
        // Get the destination square
        int dest = KnightMoves[ square ][ i ];

        // Is it occupied by a friendly piece?  If so, can't move there
        if ( ( theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) &
             jcBoard.SquareBits[ dest ] ) != 0 )
          continue;

        // Otherwise, the move is legal, so we must prepare to add it
        jcMove mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.BLACK_KNIGHT;

        // Is the destination occupied by an enemy?  If so, we have a capture
        if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) &
             jcBoard.SquareBits[ dest ] ) != 0 )
        {
          mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
          mov.CapturedPiece = theBoard.FindWhitePiece( dest );

          // If the piece we find is a king, abort because the board
          // position is illegal!
          if ( mov.CapturedPiece == jcBoard.WHITE_KING )
          {
            return false;
          }
        }
        // otherwise, it is a simple move
        else
        {
          mov.MoveType = jcMove.MOVE_NORMAL;
          mov.CapturedPiece = jcBoard.EMPTY_SQUARE;
        }
```

```
        // And we add the move to the list
        Moves.add( mov );
      }

      // Turn off the bit in the temporary bitboard; this way, we can
      // detect whether we have found the last of this type of piece
      // and short-circuit the loop
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;
    }
  }

  // We should never get here, but the return statement is added to prevent
  // obnoxious compiler warnings
  return true;
}

private boolean ComputeBlackPawnMoves( jcBoard theBoard )
{
  // Fetch the bitboard containing positions of these pieces
  long pieces = theBoard.GetBitBoard( jcBoard.BLACK_PAWN );

  // If there are no pieces of this type, no need to work very hard!
  if ( pieces == 0 )
  {
    return true;
  }

  // a small optimization
  long allPieces = theBoard.GetBitBoard( jcBoard.ALL_BLACK_PIECES ) |
                   theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES );

  // This is a black piece, so let's start looking at the top
  // of the board... But only consider positions where a pawn can
  // actually dwell!
  int dest;
  for( int square = 8; square < 56; square++ )
  {
    if ( ( pieces & jcBoard.SquareBits[ square ] ) == 0 )
      continue;

    // First, try a normal pawn pushing
    dest = square + 8;
    if ( ( allPieces & jcBoard.SquareBits[ dest ] ) == 0 )
    {
      // Unless this push results in a promotion...
      if ( square < 48 )
      {
        jcMove mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.BLACK_PAWN;
        mov.MoveType = jcMove.MOVE_NORMAL;
        Moves.add( mov );

        // Is there a chance to perform a double push? Only if the piece
        // is in its original square
        if ( square < 16 )
        {
          dest += 8;
          if ( ( allPieces & jcBoard.SquareBits[ dest ] ) == 0 )
          {
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.BLACK_PAWN;
            mov.MoveType = jcMove.MOVE_NORMAL;
            Moves.add( mov );
          }
```

```
    }
  }
  else  // if square >= 48
  {
    // We are now looking at pawn promotion!
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_QUEEN + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_KNIGHT + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_ROOK + jcMove.MOVE_NORMAL;
    Moves.add( mov );
    mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_PROMOTION_BISHOP + jcMove.MOVE_NORMAL;
    Moves.add( mov );

  }
}

// Now, let's try a capture
// Three cases: the pawn is on the 1st file, the 8th file, or elsewhere
if ( ( square % 8 ) == 0 )
{
  dest = square + 9;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest >= 56 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.CapturedPiece = theBoard.FindWhitePiece( dest );
    Moves.add( mov );

    // Other promotion captures
    if ( dest >= 56 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
```

```
        mov.MovingPiece = jcBoard.BLACK_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
        mov.CapturedPiece = theBoard.FindWhitePiece( dest );
        Moves.add( mov );
      }
    }
    // Now, try an en passant capture
    else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
      mov.CapturedPiece = jcBoard.WHITE_PAWN;
      Moves.add( mov );
    }
  }
  else if ( ( square % 8 ) == 7 )
  {
    dest = square + 7;
    // Try an ordinary capture first
    if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
      if ( dest >= 56 )
        mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
      // Other promotion captures
      if ( dest >= 56 )
      {
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.BLACK_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
        mov.CapturedPiece = theBoard.FindWhitePiece( dest );
        Moves.add( mov );
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.BLACK_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
        mov.CapturedPiece = theBoard.FindWhitePiece( dest );
        Moves.add( mov );
        mov = new jcMove();
        mov.SourceSquare = square;
        mov.DestinationSquare = dest;
        mov.MovingPiece = jcBoard.BLACK_PAWN;
        mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
        mov.CapturedPiece = theBoard.FindWhitePiece( dest );
        Moves.add( mov );
      }
    }
    // Now, try an en passant capture
    else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
    {
      jcMove mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.CapturedPiece = jcBoard.WHITE_PAWN;
      Moves.add( mov );
    }
  }
```

```
else
{
  dest = square + 9;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest >= 56 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.CapturedPiece = theBoard.FindWhitePiece( dest );
    Moves.add( mov );
    // Other promotion captures
    if ( dest >= 56 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
      mov.MovingPiece = jcBoard.BLACK_PAWN;
      mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
      mov.CapturedPiece = theBoard.FindWhitePiece( dest );
      Moves.add( mov );
    }
  }
  // Now, try an en passant capture
  else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.CapturedPiece = jcBoard.WHITE_PAWN;
    Moves.add( mov );
  }
  dest = square + 7;
  // Try an ordinary capture first
  if ( ( theBoard.GetBitBoard( jcBoard.ALL_WHITE_PIECES ) & jcBoard.SquareBits[ dest ] ) != 0 )
  {
    jcMove mov = new jcMove();
    mov.SourceSquare = square;
    mov.DestinationSquare = dest;
    mov.MovingPiece = jcBoard.BLACK_PAWN;
    mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
    if ( dest >= 56 )
      mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
    mov.CapturedPiece = theBoard.FindWhitePiece( dest );
    Moves.add( mov );
    // Other promotion captures
    if ( dest >= 56 )
    {
      mov = new jcMove();
      mov.SourceSquare = square;
      mov.DestinationSquare = dest;
```

```
            mov.MovingPiece = jcBoard.BLACK_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_KNIGHT;
            mov.CapturedPiece = theBoard.FindWhitePiece( dest );
            Moves.add( mov );
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.BLACK_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_BISHOP;
            mov.CapturedPiece = theBoard.FindWhitePiece( dest );
            Moves.add( mov );
            mov = new jcMove();
            mov.SourceSquare = square;
            mov.DestinationSquare = dest;
            mov.MovingPiece = jcBoard.BLACK_PAWN;
            mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY + jcMove.MOVE_PROMOTION_ROOK;
            mov.CapturedPiece = theBoard.FindWhitePiece( dest );
            Moves.add( mov );
          }
        }
        // Now, try an en passant capture
        else if ( ( theBoard.GetEnPassantPawn() & jcBoard.SquareBits[ dest ] ) != 0 )
        {
          jcMove mov = new jcMove();
          mov.SourceSquare = square;
          mov.DestinationSquare = dest;
          mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
          mov.MovingPiece = jcBoard.BLACK_PAWN;
          mov.CapturedPiece = jcBoard.WHITE_PAWN;
          Moves.add( mov );
        }
      }

      // And perform the usual trick to abort the loop when we no longer
      // have any pieces to look for
      pieces ^= jcBoard.SquareBits[ square ];
      if ( pieces == 0 )
        return true;

    }
    return true;
  }

  /**************************************************************************
  * STATIC BLOCK
  **************************************************************************/

// Pre-processed data structures containing all possible moves from all
// possible squares, by piece type
private static int KnightMoves[][];
private static int KingMoves[][];
private static int BishopMoves[][][];
private static int RookMoves[][][];

static
{
  // Define the KnightMoves data structure;
  KnightMoves = new int[ 64 ][];
  KnightMoves[ 0 ] = new int[ 2 ];
  KnightMoves[ 0 ][ 0 ] = 10;
  KnightMoves[ 0 ][ 1 ] = 17;
  KnightMoves[ 1 ] = new int[ 3 ];
  KnightMoves[ 1 ][ 0 ] = 16;
  KnightMoves[ 1 ][ 1 ] = 18;
  KnightMoves[ 1 ][ 2 ] = 11;
  KnightMoves[ 2 ] = new int[ 4 ];
  KnightMoves[ 2 ][ 0 ] = 8;
  KnightMoves[ 2 ][ 1 ] = 12;
  KnightMoves[ 2 ][ 2 ] = 17;
  KnightMoves[ 2 ][ 3 ] = 19;
  KnightMoves[ 3 ] = new int[ 4 ];
```

```
    KnightMoves[ 3 ][ 0 ] = 9;
    KnightMoves[ 3 ][ 1 ] = 13;
    KnightMoves[ 3 ][ 2 ] = 18;
    KnightMoves[ 3 ][ 3 ] = 20;

    // NOTE: Many pages of similar declarations eliminated from printed version
  }
}
```

```
/*****************************************************************************
 * jcPlayer.java - An abstract base class for all types of players
 * by François Dominic Laramée
 *
 * Purpose: There are currently two types of players in JavaChess: the computer
 * player and the human player.  At a later time, other types may be added,
 * including the demo player (which picks its moves from a file, in which a
 * game has been recorded move by move) and a network player (an entity from
 * which the game obtains moves via a socket connection).
 *
 * History:
 * 08.06.00 Created
 *****************************************************************************/

package javachess;


/*****************************************************************************
 * abstract public class jcPlayer
 *****************************************************************************/

abstract public class jcPlayer
{
 /*****************************************************************************
  * Constants
  *****************************************************************************/

  public static final int SIDE_BLACK = 1;
  public static final int SIDE_WHITE = 0;
  public static final String PlayerStrings[] = { "WHITE", "BLACK" };

  // Data member: which side is this player representing?
  int Side;

  // Constructor
  public jcPlayer()
  {
  }

  // Accessors
  int GetSide()
  {
    return Side;
  }
  void SetSide( int s )
  {
    Side = s;
  }

  // abstract jcMove GetMove()
  // Ask the player to provide a move, given the current board situation
  public abstract jcMove GetMove( jcBoard theBoard );
}
```

```
/*****************************************************************************
 * jcPlayerHuman.java - Interface to a human player
 * by François Dominic Laramée
 *
 * Purpose: This object allows a human player to play JavaChess.  Its only
 * real job is to query the human player for his move.
 *
 * Note that this is not the cleanest, most user-friendly piece of code around;
 * it is only intended as a test harness for the AI player, not as a full-
 * fledged application (which would be graphical, for one thing!)
 *
 * History:
 * 11.06.00 Creation
 *****************************************************************************/
package javachess;
import javachess.jcMove;
import javachess.jcBoard;
import javachess.jcMoveListGenerator;
import java.io.*;

public class jcPlayerHuman extends jcPlayer
{
  // The keyboard
  InputStreamReader kbd;
  char linebuf[];

  // Validation help
  jcMoveListGenerator Pseudos;
  jcBoard Successor;

  // Constructor
  public jcPlayerHuman( int which, InputStreamReader syskbd )
  {
    this.SetSide( which );
    linebuf = new char[ 10 ];
    kbd = syskbd;
    Pseudos = new jcMoveListGenerator();
    Successor = new jcBoard();
  }

  // public jcMove GetMove( theBoard )
  // Getting a move from the human player.  Sorry, but this is very, very
  // primitive: you need to enter square numbers instead of piece ID's, and
  // both square numbers must be entered with two digits.  Ex.: 04 00
  public jcMove GetMove( jcBoard theBoard )
  {
    // Read the move from the command line
    boolean ok = false;
    jcMove Mov = new jcMove();
    do
    {
      System.out.println( "Your move, " + PlayerStrings[ this.GetSide() ] + "?" );

      // Get data from the command line
      int len = 0;
      do {
        try{
          len = kbd.read( linebuf, 0, 5 );
        } catch( IOException e ) {}
      } while ( len < 3 );

      String line = new String( linebuf, 0, 5 );

      if ( line.equalsIgnoreCase( "RESIG" ) )
      {
        Mov.MoveType = jcMove.MOVE_RESIGN;
        return( Mov );
      }

      // Extract the source and destination squares from the line buffer
      Mov.SourceSquare = Integer.parseInt( line.substring( 0, 2 ) );
```

```
Mov.DestinationSquare = Integer.parseInt( line.substring( 3, 5 ) );
if ( ( Mov.SourceSquare < 0 ) || ( Mov.SourceSquare > 63 ) )
{
  System.out.println( "Sorry, illegal source square " + Mov.SourceSquare );
  continue;
}
if ( ( Mov.DestinationSquare < 0 ) || ( Mov.DestinationSquare > 63 ) )
{
  System.out.println( "Sorry, illegal destination square " + Mov.DestinationSquare );
  continue;
}

// Time to try to figure out what the move means!
if ( theBoard.GetCurrentPlayer() == jcPlayer.SIDE_WHITE )
{
  // Is there a piece (of the moving player) on SourceSquare?
  // If not, abort
  Mov.MovingPiece = theBoard.FindWhitePiece( Mov.SourceSquare );
  if ( Mov.MovingPiece == jcBoard.EMPTY_SQUARE )
  {
    System.out.println( "Sorry, You don't have a piece at square " + Mov.SourceSquare );
    continue;
  }

  // Three cases: there is a piece on the destination square (a capture),
  // the destination square allows an en passant capture, or it is a
  // simple non-capture move.  If the destination contains a piece of the
  // moving side, abort
  if ( theBoard.FindWhitePiece( Mov.DestinationSquare ) != jcBoard.EMPTY_SQUARE )
  {
    System.out.println( "Sorry, can't capture your own piece!" );
    continue;
  }
  Mov.CapturedPiece = theBoard.FindBlackPiece( Mov.DestinationSquare );
  if ( Mov.CapturedPiece != jcBoard.EMPTY_SQUARE )
    Mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
  else if ( ( theBoard.GetEnPassantPawn() == ( 1 << Mov.DestinationSquare ) ) &&
            ( Mov.MovingPiece == jcBoard.WHITE_PAWN ) )
  {
    Mov.CapturedPiece = jcBoard.BLACK_PAWN;
    Mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
  }

  // If the move isn't a capture, it may be a castling attempt
  else if ( ( Mov.MovingPiece == jcBoard.WHITE_KING ) &&
            ( ( Mov.SourceSquare - Mov.DestinationSquare ) == 2 ) )
    Mov.MoveType = jcMove.MOVE_CASTLING_KINGSIDE;
  else if ( ( Mov.MovingPiece == jcBoard.WHITE_KING ) &&
            ( ( Mov.SourceSquare - Mov.DestinationSquare ) == -2 ) )
    Mov.MoveType = jcMove.MOVE_CASTLING_QUEENSIDE;
  else
    Mov.MoveType = jcMove.MOVE_NORMAL;
}
else
{
  Mov.MovingPiece = theBoard.FindBlackPiece( Mov.SourceSquare );
  if ( Mov.MovingPiece == jcBoard.EMPTY_SQUARE )
  {
    System.out.println( "Sorry, you don't have a piece in square " + Mov.SourceSquare );
    continue;
  }

  if ( theBoard.FindBlackPiece( Mov.DestinationSquare ) != jcBoard.EMPTY_SQUARE )
  {
    System.out.println( "Sorry, you can't capture your own piece in square " +
                        Mov.DestinationSquare );
    continue;
  }
  Mov.CapturedPiece = theBoard.FindWhitePiece( Mov.DestinationSquare );
  if ( Mov.CapturedPiece != jcBoard.EMPTY_SQUARE )
    Mov.MoveType = jcMove.MOVE_CAPTURE_ORDINARY;
```

```
        else if ( ( theBoard.GetEnPassantPawn() == ( 1 << Mov.DestinationSquare ) ) &&
                  ( Mov.MovingPiece == jcBoard.BLACK_PAWN ) )
        {
          Mov.CapturedPiece = jcBoard.WHITE_PAWN;
          Mov.MoveType = jcMove.MOVE_CAPTURE_EN_PASSANT;
        }
        else if ( ( Mov.MovingPiece == jcBoard.BLACK_KING ) &&
                  ( ( Mov.SourceSquare - Mov.DestinationSquare ) == 2 ) )
          Mov.MoveType = jcMove.MOVE_CASTLING_KINGSIDE;
        else if ( ( Mov.MovingPiece == jcBoard.BLACK_KING ) &&
                  ( ( Mov.SourceSquare - Mov.DestinationSquare ) == -2 ) )
          Mov.MoveType = jcMove.MOVE_CASTLING_QUEENSIDE;
        else
          Mov.MoveType = jcMove.MOVE_NORMAL;
      }

      // Now, if the move results in a pawn promotion, we must ask the user
      // for the type of promotion!
      if ( ( ( Mov.MovingPiece == jcBoard.WHITE_PAWN ) && ( Mov.DestinationSquare < 8 ) ) ||
           ( ( Mov.MovingPiece == jcBoard.BLACK_PAWN ) && ( Mov.DestinationSquare > 55 ) ) )
      {
        int car = -1;
        System.out.println( "Promote the pawn to [K]night, [R]ook, [B]ishop, [Q]ueen?" );
        do
        {
          try { car = kbd.read(); } catch( IOException e ) {}
        } while ( ( car != 'K' ) && ( car != 'k' ) && ( car != 'b' ) && ( car != 'B' )
              && ( car != 'R' ) && ( car != 'r' ) && ( car != 'Q' ) && ( car != 'q' ) );
        if ( ( car == 'K' ) || ( car == 'k' ) )
          Mov.MoveType += jcMove.MOVE_PROMOTION_KNIGHT;
        else if ( ( car == 'B' ) || ( car == 'b' ) )
          Mov.MoveType += jcMove.MOVE_PROMOTION_BISHOP;
        else if ( ( car == 'R' ) || ( car == 'r' ) )
          Mov.MoveType += jcMove.MOVE_PROMOTION_ROOK;
        else
          Mov.MoveType += jcMove.MOVE_PROMOTION_QUEEN;
      }

      // OK, now let's see if the move is actually legal!  First step: a check
      // for pseudo-legality, i.e., is it a valid successor to the current
      // board?
      Pseudos.ComputeLegalMoves( theBoard );
      if ( !Pseudos.Find( Mov ) )
      {
        System.out.print( "Sorry, this move is not in the pseudo-legal list: " );
        Mov.Print();
        Pseudos.Print();
        continue;
      }

      // If pseudo-legal, then verify whether it leaves the king in check
      Successor.Clone( theBoard );
      Successor.ApplyMove( Mov );
      if ( !Pseudos.ComputeLegalMoves( Successor ) )
      {
        System.out.print( "Sorry, this move leaves your king in check: " );
        Mov.Print();
        continue;
      }

      // If we have made it here, we have a valid move to play!
      System.out.println( "Move is accepted..." );
      ok = true;

    } while ( !ok );

    return( Mov );
  }
}
```

```
/**************************************************************************
 *
 * jcPlayerAI.java - Interface to a computer player
 * by François Dominic Laramée
 *
 * Purpose: This object allows a computer player to play JavaChess.  Its only
 * real job is to query an AI Search Agent for his move.
 *
 * History:
 * 11.06.00 Creation
 * 07.08.00 Association with the search agent
 *
 **************************************************************************/

package javachess;
import javachess.jcAISearchAgent;

public class jcPlayerAI extends jcPlayer
{
  /************************************************************************
   * DATA MEMBERS
   ************************************************************************/

  // The search agent in charge of the moves
  jcAISearchAgent Agent;

  /************************************************************************
   * PUBLIC METHODS
   ************************************************************************/

  // Constructor
  public jcPlayerAI( int whichPlayer, int whichType, jcOpeningBook ref )
  {
    this.SetSide( whichPlayer );
    Agent = jcAISearchAgent.MakeNewAgent( whichType, ref );
  }

  // Attach a search agent to the AI player
  public boolean AttachSearchAgent( jcAISearchAgent theAgent )
  {
    Agent = theAgent;
    return true;
  }

  // Getting a move from the machine
  public jcMove GetMove( jcBoard theBoard )
  {
    return( Agent.PickBestMove( theBoard ) );
  }
}
```

```
/**************************************************************************
 * jcAISearchAgent - An object which picks a best move according to a
 *                   variant of alphabeta search or another
 *
 * Purpose:
 * This is the object which picks a move for the computer player.  Implemented
 * as an abstract class to allow multiple search strategies to be played with.
 *
 * History
 * 07.08.00 Creation
 * 05.10.00 Added statistics and some corrections
 **************************************************************************/

package javachess;
import javachess.jcBoard;
import javachess.jcBoardEvaluator;
import javachess.jcAISearchAgentMTDF;
import javachess.jcAISearchAgentAlphabeta;
import javachess.jcTranspositionTable;
import java.util.Random;

public abstract class jcAISearchAgent
{
  /************************************************************************
   * DATA MEMBERS
   ************************************************************************/

  // A transposition table for this object
  jcTranspositionTable TransTable;

  // A handle to the system's history table
  jcHistoryTable HistoryTable;

  // How will we assess position strengths?
  protected jcBoardEvaluator Evaluator;
  protected int FromWhosePerspective;

  // ID's for concrete subclasses; jcAISearchAgent works as a factory for its
  // concrete subclasses
  public static final int AISEARCH_ALPHABETA = 0;
  public static final int AISEARCH_MTDF = 1;

  // Search node types: MAXNODEs are nodes where the computer player is the
  // one to move; MINNODEs are positions where the opponent is to move.
  protected static final boolean MAXNODE = true;
  protected static final boolean MINNODE = false;

  // Alphabeta search boundaries
  protected static final int ALPHABETA_MAXVAL = 30000;
  protected static final int ALPHABETA_MINVAL = -30000;
  protected static final int ALPHABETA_ILLEGAL = -31000;

  // An approximate upper bound on the total value of all positional
  // terms in the evaluation function
  protected static final int EVAL_THRESHOLD = 200;

  // A score below which we give up: if Alphabeta ever returns a value lower
  // than this threshold, then all is lost and we might as well resign.  Here,
  // the value is equivalent to "mated by the opponent in 3 moves or less".
  protected static final int ALPHABETA_GIVEUP = -29995;
          .
  Random Rnd;

  // Statistics
  int NumRegularNodes;
  int NumQuiescenceNodes;
  int NumRegularTTHits;
  int NumQuiescenceTTHits;
  int NumRegularCutoffs;
  int NumQuiescenceCutoffs;
```

```
// A move counter, so that the agent knows when it can delete old stuff from
// its transposition table
int MoveCounter;

/**************************************************************************
 * PUBLIC METHODS
 **************************************************************************/

// Construction
public jcAISearchAgent()
{
  TransTable = new jcTranspositionTable();
  HistoryTable = jcHistoryTable.GetInstance();
  Evaluator = new jcBoardEvaluator();
  Rnd = new Random();
  MoveCounter = 0;
}

public jcAISearchAgent( jcBoardEvaluator eval )
{
  AttachEvaluator( eval );
}

// boolean AttachEvaluator( jcBoardEvaluator eval )
// Pick a function which the agent will use to assess the potency of a
// position.  This may change during the game; for example, a special
// "mop-up" evaluator may replace the standard when it comes time to drive
// a decisive advantage home at the end of the game.
public boolean AttachEvaluator( jcBoardEvaluator eval )
{
  Evaluator = eval;
  return true;
}

// int AlphaBeta
// The basic alpha-beta algorithm, used in one disguise or another by
// every search agent class
public int AlphaBeta( boolean nodeType, jcBoard theBoard, int depth,
                      int alpha, int beta )
{
  jcMove mov = new jcMove();

  // Count the number of nodes visited in the full-width search
  NumRegularNodes++;

  // First things first: let's see if there is already something useful
  // in the transposition table, which might save us from having to search
  // anything at all
  if ( TransTable.LookupBoard( theBoard, mov ) && ( mov.SearchDepth >= depth ) )
  {
    if ( nodeType == MAXNODE )
    {
      if ( ( mov.MoveEvaluationType == jcMove.EVALTYPE_ACCURATE ) ||
           ( mov.MoveEvaluationType == jcMove.EVALTYPE_LOWERBOUND ) )
      {
        if ( mov.MoveEvaluation >= beta )
        {
          NumRegularTTHits++;
          return mov.MoveEvaluation;
        }
      }
    }
    else
    {
      if ( ( mov.MoveEvaluationType == jcMove.EVALTYPE_ACCURATE ) ||
           ( mov.MoveEvaluationType == jcMove.EVALTYPE_UPPERBOUND ) )
      {
        if ( mov.MoveEvaluation <= alpha )
        {
          NumRegularTTHits++;
          return mov.MoveEvaluation;
```

```
      }
    }
  }
}

// If we have reached the maximum depth of the search, stop recursion
// and begin quiescence search
if ( depth == 0 )
{
  return QuiescenceSearch( nodeType, theBoard, alpha, beta );
}

// Otherwise, generate successors and search them in turn
// If ComputeLegalMoves returns false, then the current position is illegal
// because one or more moves could capture a king!
// In order to slant the computer's strategy in favor of quick mates, we
// give a bonus to king captures which occur at shallow depths, i.e., the
// more plies left, the better.  On the other hand, if you are losing, it
// really doesn't matter how fast...
jcMoveListGenerator movegen = new jcMoveListGenerator();
if ( !movegen.ComputeLegalMoves( theBoard ) )
{
  return ALPHABETA_ILLEGAL;
}

// Sort the moves according to History heuristic values
HistoryTable.SortMoveList( movegen, theBoard.GetCurrentPlayer() );


// OK, now, get ready to search
jcBoard newBoard = new jcBoard();
int bestSoFar;

// Case #1: We are searching a Max Node
if ( nodeType == jcAISearchAgent.MAXNODE )
{
  bestSoFar = ALPHABETA_MINVAL;
  int currentAlpha = alpha;

  // Loop on the successors
  while( ( mov = movegen.Next() ) != null )
  {
    // Compute a board position resulting from the current successor
    newBoard.Clone( theBoard );
    newBoard.ApplyMove( mov );

    // And search it in turn
    int movScore = AlphaBeta( !nodeType, newBoard, depth - 1, currentAlpha,
                              beta );
    // Ignore illegal moves in the alphabeta evaluation
    if ( movScore == ALPHABETA_ILLEGAL )
      continue;

    currentAlpha = Math.max( currentAlpha, movScore );

    // Is the current successor better than the previous best?
    if ( movScore > bestSoFar )
    {
      bestSoFar = movScore;
      // Can we cutoff now?
      if ( bestSoFar >= beta )
      {
        // Store this best move in the TransTable
        TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_UPPERBOUND,
                               depth, MoveCounter );

        // Add this move's efficiency in the HistoryTable
        HistoryTable.AddCount( theBoard.GetCurrentPlayer(), mov );
        NumRegularCutoffs++;
        return bestSoFar;
      }
```

```
    }
  }

  // Test for checkmate or stalemate
  // Both cases occur if and only if there is no legal move for MAX, i.e.,
  // if "bestSoFar" is ALPHABETA_MINVAL.  There are two cases: we
  // have checkmate (in which case the score is accurate) or stalemate (in
  // which case the position should be re-scored as a draw with value 0.
  if ( bestSoFar <= ALPHABETA_MINVAL )
  {
    // Can MIN capture MAX's king?  First, ask the machine to generate
    // moves for MIN
    newBoard.Clone( theBoard );
    if( newBoard.GetCurrentPlayer() == FromWhosePerspective )
      newBoard.SwitchSides();

    // And if one of MIN's moves is a king capture, indicating that the
    // position is illegal, we have checkmate and must return MINVAL.  We
    // add the depth simply to "favor" delaying tactics: a mate in 5 will
    // score higher than a mate in 3, because the likelihood that the
    // opponent will miss it is higher; might as well make life difficult!
    if ( !movegen.ComputeLegalMoves( newBoard ) )
      return bestSoFar + depth;
    else
      return 0;
  }
}
else
// Case #2: Min Node
{
  bestSoFar = ALPHABETA_MAXVAL;
  int currentBeta = beta;
  while( ( mov = movegen.Next() ) != null )
  {
    newBoard.Clone( theBoard );
    newBoard.ApplyMove( mov );

    int movScore = AlphaBeta( !nodeType, newBoard, depth - 1, alpha,
                              currentBeta );
    if ( movScore == ALPHABETA_ILLEGAL )
      continue;
    currentBeta = Math.min( currentBeta, movScore );
    if ( movScore < bestSoFar )
    {
      bestSoFar = movScore;
      // Cutoff?
      if ( bestSoFar <= alpha )
      {
        TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_UPPERBOUND,
                               depth, MoveCounter );
        HistoryTable.AddCount( theBoard.GetCurrentPlayer(), mov );
        NumRegularCutoffs++;
        return bestSoFar;
      }
    }
  }
  // Test for checkmate or stalemate
  if ( bestSoFar >= ALPHABETA_MAXVAL )
  {
    // Can MAX capture MIN's king?
    newBoard.Clone( theBoard );
    if( newBoard.GetCurrentPlayer() != FromWhosePerspective )
      newBoard.SwitchSides();
    if ( !movegen.ComputeLegalMoves( newBoard ) )
      return bestSoFar + depth;
    else
      return 0;
  }
}

// If we haven't returned yet, we have found an accurate minimax score
```

```
  // for a position which is neither a checkmate nor a stalemate
  TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_ACCURATE, depth, MoveCounter );
  return bestSoFar;
}

// int QuiescenceSearch
// A slight variant of alphabeta which only considers captures and null moves
// This is necesary because the evaluation function can only be applied to
// "quiet" positions where the tactical situation (i.e., material balance) is
// unlikely to change in the near future.
// Note that, in this version of the code, the quiescence search is not limited
// by depth; we continue digging for as long as we can find captures.  Some other
// programs impose a depth limit for time-management purposes.
public int QuiescenceSearch( boolean nodeType, jcBoard theBoard, int alpha, int beta )
{
  jcMove mov = new jcMove();
  NumQuiescenceNodes++;

  // First things first: let's see if there is already something useful
  // in the transposition table, which might save us from having to search
  // anything at all
  if ( TransTable.LookupBoard( theBoard, mov ) )
  {
    if ( nodeType == MAXNODE )
    {
      if ( ( mov.MoveEvaluationType == jcMove.EVALTYPE_ACCURATE ) ||
           ( mov.MoveEvaluationType == jcMove.EVALTYPE_LOWERBOUND ) )
      {
        if ( mov.MoveEvaluation >= beta )
        {
          NumQuiescenceTTHits++;
          return mov.MoveEvaluation;
        }
      }
    }
    else
    {
      if ( ( mov.MoveEvaluationType == jcMove.EVALTYPE_ACCURATE ) ||
           ( mov.MoveEvaluationType == jcMove.EVALTYPE_UPPERBOUND ) )
      {
        if ( mov.MoveEvaluation <= alpha )
        {
          NumQuiescenceTTHits++;
          return mov.MoveEvaluation;
        }
      }
    }
  }

  int bestSoFar = ALPHABETA_MINVAL;

  // Start with evaluation of the null-move, just to see whether it is more
  // effective than any capture, in which case we must stop looking at
  // captures and damaging our position
  // NOTE: If the quick evaluation is enough to cause a cutoff, we don't store
  // the value in the transposition table.  EvaluateQuickie is so fast that we
  // wouldn't gain anything, and storing the value might force us to erase a
  // more valuable entry in the table.
  bestSoFar = Evaluator.EvaluateQuickie( theBoard, FromWhosePerspective );
  if ( ( bestSoFar > ( beta + EVAL_THRESHOLD ) ) || ( bestSoFar < ( alpha - EVAL_THRESHOLD ) ) )
    return bestSoFar;
  else
    bestSoFar = Evaluator.EvaluateComplete( theBoard, FromWhosePerspective );

  // Now, look at captures
  jcMoveListGenerator movegen = new jcMoveListGenerator();
  if ( !movegen.ComputeQuiescenceMoves( theBoard ) )
  {
    return bestSoFar;
  }
```

```
jcBoard newBoard = new jcBoard();

// Case #1: We are searching a Max Node
if ( nodeType == jcAISearchAgent.MAXNODE )
{
  int currentAlpha = alpha;
  // Loop on the successors
  while( ( mov = movegen.Next() ) != null )
  {
    // Compute a board position resulting from the current successor
    newBoard.Clone( theBoard );
    newBoard.ApplyMove( mov );

    // And search it in turn
    int movScore = QuiescenceSearch( !nodeType, newBoard, currentAlpha, beta );
    // Ignore illegal moves in the alphabeta evaluation
    if ( movScore == ALPHABETA_ILLEGAL )
      continue;
    currentAlpha = Math.max( currentAlpha, movScore );

    // Is the current successor better than the previous best?
    if ( movScore > bestSoFar )
    {
      bestSoFar = movScore;
      // Can we cutoff now?
      if ( bestSoFar >= beta )
      {
        TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_UPPERBOUND, 0, MoveCounter );
        // Add this move's efficiency in the HistoryTable
        HistoryTable.AddCount( theBoard.GetCurrentPlayer(), mov );
        NumQuiescenceCutoffs++;
        return bestSoFar;
      }
    }
  }

  // Test for checkmate or stalemate
  // Both cases occur if and only if there is no legal move for MAX, i.e.,
  // if "bestSoFar" is ALPHABETA_MINVAL.  There are two cases: we
  // have checkmate (in which case the score is accurate) or stalemate (in
  // which case the position should be re-scored as a draw with value 0.
  if ( bestSoFar <= ALPHABETA_MINVAL )
  {
    // Can MIN capture MAX's king?  First, ask the machine to generate
    // moves for MIN
    newBoard.Clone( theBoard );
    if( newBoard.GetCurrentPlayer() == FromWhosePerspective )
      newBoard.SwitchSides();
    // And if one of MIN's moves is a king capture, indicating that the
    // position is illegal, we have checkmate and must return MINVAL.  We
    // add the depth simply to "favor" delaying tactics: a mate in 5 will
    // score higher than a mate in 3, because the likelihood that the
    // opponent will miss it is higher; might as well make life difficult!
    if ( !movegen.ComputeLegalMoves( newBoard ) )
      return bestSoFar;
    else
      return 0;
  }
}
else
// Case #2: Min Node
{
  int currentBeta = beta;
  while( ( mov = movegen.Next() ) != null )
  {
    newBoard.Clone( theBoard );
    newBoard.ApplyMove( mov );

    int movScore = QuiescenceSearch( !nodeType, newBoard, alpha, currentBeta );
    if ( movScore == ALPHABETA_ILLEGAL )
      continue;
```

```
        currentBeta = Math.min( currentBeta, movScore );
        if ( movScore < bestSoFar )
        {
          bestSoFar = movScore;
          // Cutoff?
          if ( bestSoFar <= alpha )
          {
            TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_UPPERBOUND, 0, MoveCounter );
            HistoryTable.AddCount( theBoard.GetCurrentPlayer(), mov );
            NumQuiescenceCutoffs++;
            return bestSoFar;
          }
        }
      }
      // Test for checkmate or stalemate
      if ( bestSoFar >= ALPHABETA_MAXVAL )
      {
        // Can MAX capture MIN's king?
        newBoard.Clone( theBoard );
        if( newBoard.GetCurrentPlayer() != FromWhosePerspective )
          newBoard.SwitchSides();
        if ( !movegen.ComputeLegalMoves( newBoard ) )
          return bestSoFar;
        else
          return 0;
      }
    }

    // If we haven't returned yet, we have found an accurate minimax score
    // for a position which is neither a checkmate nor a stalemate
    TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_ACCURATE, 0, MoveCounter );
    return bestSoFar;
  }


  // jcAISearchAgent MakeNewAgent
  // Standard "subclass factory" design pattern
  public static jcAISearchAgent MakeNewAgent( int type, jcOpeningBook ref )
  {
    switch( type )
    {
      case AISEARCH_ALPHABETA:
        return( new jcAISearchAgentAlphabeta() );
      case AISEARCH_MTDF:
        return( new jcAISearchAgentMTDF( ref ) );
      default:
        return null;
    }
  }

  // jcMove PickBestMove( jcBoard theBoard )
  // Each agent class needs some way of picking a move!
  public abstract jcMove PickBestMove( jcBoard theBoard );
}
```

```
/**************************************************************************
 * jcAISearchAgentAlphabeta - The most basic search agent
 *
 * Purpose: Pick a best move using a simple, fixed-depth, full-width
 * alphabeta search, like they did in the Dark Ages ;-)
 *
 * History
 * 07.08.00 Initial writing
 **************************************************************************/

package javachess;
import javachess.jcAISearchAgent;
import javachess.jcBoard;

public class jcAISearchAgentAlphabeta extends jcAISearchAgent
{
  // Construction
  public jcAISearchAgentAlphabeta()
  {
    super();
  }

  // jcMove PickBestMove
  // Implementation of the abstract method defined in the superclass
  public jcMove PickBestMove( jcBoard theBoard )
  {
    // Store the identity of the moving side, so that we can tell Evaluator
    // from whose perspective we need to evaluate positions
    FromWhosePerspective = theBoard.GetCurrentPlayer();
    MoveCounter++;

    // Should we erase the history table?
    if ( ( Rnd.nextInt() % 4 ) == 2 )
      HistoryTable.Forget();

    NumRegularNodes = 0; NumQuiescenceNodes = 0;
    NumRegularTTHits = 0; NumQuiescenceTTHits = 0;

    // Find the moves
    jcMove theMove = new jcMove();
    jcMoveListGenerator movegen = new jcMoveListGenerator();
    movegen.ComputeLegalMoves( theBoard );
    HistoryTable.SortMoveList( movegen, theBoard.GetCurrentPlayer() );

    // The following code blocks look a lot like the MAX node case from
    // jcAISearchAgent.Alphabeta, with an added twist: we need to store the
    // actual best move, and not only pass around its minimax value
    int bestSoFar = ALPHABETA_MINVAL;
    jcBoard newBoard = new jcBoard();
    jcMove mov;
    int currentAlpha = ALPHABETA_MINVAL;

    // Loop on all pseudo-legal moves
    while( ( mov = movegen.Next() ) != null )
    {
      newBoard.Clone( theBoard );
      newBoard.ApplyMove( mov );
      int movScore = AlphaBeta( MINNODE, newBoard, 5, currentAlpha, ALPHABETA_MAXVAL );
      if ( movScore == ALPHABETA_ILLEGAL )
        continue;

      currentAlpha = Math.max( currentAlpha, movScore );

      if ( movScore > bestSoFar )
      {
        theMove.Copy( mov );
        bestSoFar = movScore;
        theMove.MoveEvaluation = movScore;
      }
    }
```

```
    // And now, if the best we can do is ALPHABETA_GIVEUP or worse, then it is
    // time to resign...  Unless the opponent was kind wnough to put us in
    // stalemate!
    if ( bestSoFar <= ALPHABETA_GIVEUP )
    {
      // Check for a stalemate
      // Stalemate occurs if the player's king is NOT in check, but all of his
      // moves are illegal.
      // First, verify whether we are in check
      newBoard.Clone( theBoard );
      jcMoveListGenerator secondary = new jcMoveListGenerator();
      newBoard.SwitchSides();
      if ( secondary.ComputeLegalMoves( newBoard ) )
      {
        // Then, we are not in check and may continue our efforts.
        // We must now examine all possible moves; if at least one resuls in
        // a legal position, there is no stalemate and we must assume that
        // we are doomed
        HistoryTable.SortMoveList( movegen, newBoard.GetCurrentPlayer() );
        movegen.ResetIterator();
        // If we can scan all moves without finding one which results
        // in a legal position, we have a stalemate
        theMove.MoveType = jcMove.MOVE_STALEMATE;
        theMove.MovingPiece = jcBoard.KING + theBoard.GetCurrentPlayer();
        // Look at the moves
        while( ( mov = movegen.Next() ) != null )
        {
          newBoard.Clone( theBoard );
          newBoard.ApplyMove( mov );
          if ( secondary.ComputeLegalMoves( newBoard ) )
          {
            theMove.MoveType = jcMove.MOVE_RESIGN;
          }
        }
      }
      else
      {
        // We're in check and our best hope is GIVEUP or worse, so either we are
        // already checkmated or will be soon, without hope of escape
        theMove.MovingPiece = jcBoard.KING + theBoard.GetCurrentPlayer();
        theMove.MoveType = jcMove.MOVE_RESIGN;
      }
    }

    System.out.print( "  --> Transposition Table hits for regular nodes: " );
    System.out.println( NumRegularTTHits + " of " + NumRegularNodes );
    System.out.print( "  --> Transposition Table hits for quiescence nodes: " );
    System.out.println( NumQuiescenceTTHits + " of " + NumQuiescenceNodes );

    return theMove;
  }
}
```

```
/**************************************************************************
 * jcAISearchAgentMTDF - A sophisticated search agent
 *
 * Purpose: A (mostly) state-of-the-art search agent, implementing advanced
 * techniques like the iterative-deepening MTDF search algorithm, transposition
 * table, opening book and history table.
 *
 * History:
 * 05.10.00 Completed initial version
 **************************************************************************/

package javachess;
import javachess.jcAISearchAgent;
import javachess.jcBoard;
import javachess.jcOpeningBook;

public class jcAISearchAgentMTDF extends jcAISearchAgent
{
  // A reference to the game's opening book
  private jcOpeningBook Openings;

  // A measure of the effort we are willing to expend on search
  private static final int MaxSearchSize = 50000;

  // Construction
  public jcAISearchAgentMTDF( jcOpeningBook ref )
  {
    super();
    Openings = ref;
  }

  /**************************************************************************
   * PUBLIC METHODS
   **************************************************************************/

  // Move selection: An iterative-deepening paradigm calling MTD(f) repeatedly
  public jcMove PickBestMove( jcBoard theBoard )
  {
    // First things first: look in the Opening Book, and if it contains a
    // move for this position, don't search anything
    MoveCounter++;
    jcMove Mov = null;
    Mov = Openings.Query( theBoard );
    if ( Mov != null )
      return Mov;

    // Store the identity of the moving side, so that we can tell Evaluator
    // from whose perspective we need to evaluate positions
    FromWhosePerspective = theBoard.GetCurrentPlayer();

    // Should we erase the history table?
    if ( ( Rnd.nextInt() % 6 ) == 2 )
      HistoryTable.Forget();

    // Begin search.  The search's maximum depth is determined on the fly,
    // according to how much effort has been spent; if it's possible to search
    // to depth 8 in 5 seconds, then by all means, do it!
    int bestGuess = 0;
    int iterdepth = 1;

    while( true )
    {
      // Searching to depth 1 is not very effective, so we begin at 2
      iterdepth++;

      // Compute efficiency statistics
      NumRegularNodes = 0; NumQuiescenceNodes = 0;
      NumRegularTTHits = 0; NumQuiescenceTTHits = 0;
      NumRegularCutoffs = 0; NumQuiescenceCutoffs = 0;

      // Look for a move at the current depth
```

```
    Mov = MTDF( theBoard, bestGuess, iterdepth );
    bestGuess = Mov.MoveEvaluation;

    // Feedback!
    System.out.print( "Iteration of depth " + iterdepth + "; best move = " );
    Mov.Print();
    System.out.print( "  --> Transposition Table hits for regular nodes: " );
    System.out.println( NumRegularTTHits + " of " + NumRegularNodes );
    System.out.print( "  --> Transposition Table hits for quiescence nodes: " );
    System.out.println( NumQuiescenceTTHits + " of " + NumQuiescenceNodes );
    System.out.println( "  --> Number of cutoffs for regular nodes: " + NumRegularCutoffs );
    System.out.println( "  --> Number of cutoffs in quiescence search: " + NumQuiescenceCutoffs );

    // Get out if we have searched deep enough
    if ( ( NumRegularNodes + NumQuiescenceNodes ) > MaxSearchSize )
      break;
    if ( iterdepth >= 15 )
      break;
  }

  return Mov;
}

/**************************************************************************
 * PRIVATE METHODS
 **************************************************************************/

// private jcMove MTDF
// Use the MTDF algorithm to find a good move.  MTDF repeatedly calls
// alphabeta with a zero-width search window, which creates very many quick
// cutoffs.  If alphabeta fails low, the next call will place the search
// window lower; in a sense, MTDF is a sort of binary search mechanism into
// the minimax space.
private jcMove MTDF( jcBoard theBoard, int target, int depth )
{
  int beta;
  jcMove Mov;
  int currentEstimate = target;
  int upperbound = ALPHABETA_MAXVAL;
  int lowerbound = ALPHABETA_MINVAL;

  // This is the trick: make repeated calls to alphabeta, zeroing in on the
  // actual minimax value of theBoard by narrowing the bounds
  do {
    if ( currentEstimate == lowerbound )
      beta = currentEstimate + 1;
    else
      beta = currentEstimate;

    Mov = UnrolledAlphabeta( theBoard, depth, beta - 1, beta );
    currentEstimate = Mov.MoveEvaluation;

    if ( currentEstimate < beta )
      upperbound = currentEstimate;
    else
      lowerbound = currentEstimate;

  } while ( lowerbound < upperbound );

  return Mov;
}

// private jcMove UnrolledAlphabeta
// The standard alphabeta, with the top level "unrolled" so that it can
// return a jcMove structure instead of a mere minimax value
// See jcAISearchAgent.Alphabeta for detailed comments on this code
private jcMove UnrolledAlphabeta( jcBoard theBoard, int depth, int alpha,
                                  int beta )
{
  jcMove BestMov = new jcMove();
```

```
jcMoveListGenerator movegen = new jcMoveListGenerator();
movegen.ComputeLegalMoves( theBoard );
HistoryTable.SortMoveList( movegen, theBoard.GetCurrentPlayer() );

jcBoard newBoard = new jcBoard();
int bestSoFar;

bestSoFar = ALPHABETA_MINVAL;
int currentAlpha = alpha;
jcMove mov;

// Loop on the successors
while( ( mov = movegen.Next() ) != null )
{
  // Compute a board position resulting from the current successor
  newBoard.Clone( theBoard );
  newBoard.ApplyMove( mov );

  // And search it in turn
  int movScore = AlphaBeta( MINNODE, newBoard, depth - 1, currentAlpha, beta );

  // Ignore illegal moves in the alphabeta evaluation
  if ( movScore == ALPHABETA_ILLEGAL )
    continue;
  currentAlpha = Math.max( currentAlpha, movScore );

  // Is the current successor better than the previous best?
  if ( movScore > bestSoFar )
  {
    BestMov.Copy( mov );
    bestSoFar = movScore;
    BestMov.MoveEvaluation = bestSoFar;

    // Can we cutoff now?
    if ( bestSoFar >= beta )
    {
      TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_UPPERBOUND, depth, MoveCounter );

      // Add this move's efficiency in the HistoryTable
      HistoryTable.AddCount( theBoard.GetCurrentPlayer(), mov );
      return BestMov;
    }
  }
}

// Test for checkmate or stalemate
if ( bestSoFar <= ALPHABETA_GIVEUP )
{
  newBoard.Clone( theBoard );
  jcMoveListGenerator secondary = new jcMoveListGenerator();
  newBoard.SwitchSides();
  if ( secondary.ComputeLegalMoves( newBoard ) )
  {
    // Then, we are not in check and may continue our efforts.
    HistoryTable.SortMoveList( movegen, newBoard.GetCurrentPlayer() );
    movegen.ResetIterator();
    BestMov.MoveType = jcMove.MOVE_STALEMATE;
    BestMov.MovingPiece = jcBoard.KING + theBoard.GetCurrentPlayer();
    while( ( mov = movegen.Next() ) != null )
    {
      newBoard.Clone( theBoard );
      newBoard.ApplyMove( mov );
      if ( secondary.ComputeLegalMoves( newBoard ) )
      {
        BestMov.MoveType = jcMove.MOVE_RESIGN;
      }
    }
  }
  else
  {
    // We're in check and our best hope is GIVEUP or worse, so either we are
```

```
            // already checkmated or will be soon, without hope of escape
            BestMov.MovingPiece = jcBoard.KING + theBoard.GetCurrentPlayer();
            BestMov.MoveType = jcMove.MOVE_RESIGN;
          }
        }

        // If we haven't returned yet, we have found an accurate minimax score
        // for a position which is neither a checkmate nor a stalemate
        TransTable.StoreBoard( theBoard, bestSoFar, jcMove.EVALTYPE_ACCURATE, depth, MoveCounter );

        return BestMov;
      }
    }
```

```
/**************************************************************************
 * jcHistoryTable - A heuristic used to pick an order of evaluation for moves
 *
 * The history heuristic is an extension of the old "killer move" system: if
 * a move has caused a lot of cutoffs recently, it will be tried early in the
 * hope that it will do so again.
 *
 * Using the history table is a gamble.  We could do without it entirely,
 * compute "successor positions" for each possible moves, look them up in
 * the transposition table and hope to get a cutoff this way, which would
 * insure fast cutoffs whenever possible.  On the other hand, HistoryTable
 * has no knowledge of the contents of the transposition table, so it may
 * cause a deep search of several moves even though another one would result
 * in an immediate cutoff...  However, History requires far less memory
 * and computation than creating a ton of successor jcBoard objects, so we
 * hope that, on average, it will still be more efficient overall.
 *
 * History
 * 14.08.00 Creation
 **************************************************************************/

package javachess;
import javachess.jcMoveListGenerator;
import javachess.jcMove;
import java.util.*;

public class jcHistoryTable
{
  /************************************************************************
   * DATA MEMBERS
   ************************************************************************/

  // the table itself; a separate set of cutoff counters exists for each
  // side
  int History[][][];
  int CurrentHistory[][];

  // This is a singleton class; the same history can be shared by two AI's
  private static jcHistoryTable theInstance;

  // A comparator, used to sort the moves
  private jcMoveComparator MoveComparator;

  /************************************************************************
   * STATIC BLOCK
   ************************************************************************/
  static
  {
    theInstance = new jcHistoryTable();
  }

  /************************************************************************
   *  jcMoveComparator - Inner class used in sorting moves
   ************************************************************************/
   class jcMoveComparator implements Comparator
   {
     public int compare( Object o1, Object o2 )
     {
       jcMove mov1 = (jcMove) o1;
       jcMove mov2 = (jcMove) o2;
       if ( CurrentHistory[ mov1.SourceSquare ][ mov1.DestinationSquare ] >
            CurrentHistory[ mov2.SourceSquare ][ mov2.DestinationSquare ] )
         return -1;
       else
         return 1;
     }
   }

  /************************************************************************
   * PUBLIC METHODS
   ************************************************************************/
```

```
  // Accessor
  public static jcHistoryTable GetInstance()
  {
    return theInstance;
  }

  // Sort a list of moves, using the Java "Arrays" class as a helper
  public boolean SortMoveList( jcMoveListGenerator theList, int movingPlayer )
  {
    // Which history will we use?
    CurrentHistory = History[ movingPlayer ];

    // Arrays can't sort a dynamic array like jcMoveListGenerator's ArrayList
    // member, so we have to use an intermediate.  Annoying and not too clean,
    // but it works...
    Arrays.sort( theList.GetMoveList().toArray(), 0, theList.Size(), MoveComparator );
    return true;
  }

  // History table compilation
  public boolean AddCount( int whichPlayer, jcMove mov )
  {
    History[ whichPlayer ][ mov.SourceSquare ][ mov.DestinationSquare ]++;
    return true;
  }


  // public boolean Forget
  // Once in a while, we must erase the history table to avoid ordering
  // moves according to the results of very old searches
  public boolean Forget()
  {
    for( int i = 0; i < 2; i++ )
      for( int j = 0; j < 64; j++ )
        for( int k = 0; k < 64; k++ )
          History[ i ][ j ][ k ] = 0;
    return true;
  }

  /**************************************************************************
   * PRIVATE METHODS
   *************************************************************************/
  private jcHistoryTable()
  {
    History = new int[ 2 ][ 64 ][ 64 ];
    MoveComparator = new jcMoveComparator();
  }
}
```

```
/**************************************************************************
 * jcBoardEvaluator - Analyzes and evaluates a chess board position
 * by F.D. Laramee
 *
 * History
 * 07.08.00 Creation
 * ************************************************************************/

package javachess;
import javachess.jcBoard;

public class jcBoardEvaluator
{

  /**************************************************************************
   * DATA MEMBERS
   *************************************************************************/

  // Data counters to evaluate pawn structure
  int MaxPawnFileBins[];
  int MaxPawnColorBins[];
  int MaxTotalPawns;
  int PawnRams;
  int MaxMostAdvanced[];
  int MaxPassedPawns[];
  int MinPawnFileBins[];
  int MinMostBackward[];

  // The "graininess" of the evaluation.  MTD(f) works a lot faster if the
  // evaluation is relatively coarse
  private static final int Grain = 3;

  /**************************************************************************
   * PUBLIC METHODS
   *************************************************************************/

  // Construction
  public jcBoardEvaluator()
  {
    MaxPawnFileBins = new int[ 8 ];
    MaxPawnColorBins = new int[ 2 ];
    MaxMostAdvanced = new int[ 8 ];
    MaxPassedPawns = new int[ 8 ];
    MinPawnFileBins = new int[ 8 ];
    MinMostBackward = new int[ 8 ];
  }

  // int EvaluateQuickie( jcBoard theBoard, int FromWhosePerspective )
  // A simple, fast evaluation based exclusively on material.  Since material
  // is overwhelmingly more important than anything else, we assume that if a
  // position's material value is much lower (or much higher) than another,
  // then there is no need to waste time on positional factors because they
  // won't be enough to tip the scales the other way, so to speak.
  public int EvaluateQuickie( jcBoard theBoard, int fromWhosePerspective )
  {
    return ( ( theBoard.EvalMaterial( fromWhosePerspective ) >> Grain ) << Grain );
  }

  // int EvaluateComplete( jcBoard theBoard )
  // A detailed evaluation function, taking into account several positional
  // factors
  public int EvaluateComplete( jcBoard theBoard, int fromWhosePerspective )
  {
    AnalyzePawnStructure( theBoard, fromWhosePerspective );
    return(((theBoard.EvalMaterial( fromWhosePerspective ) +
             EvalPawnStructure( fromWhosePerspective ) +
             EvalBadBishops( theBoard, fromWhosePerspective ) +
             EvalDevelopment( theBoard, fromWhosePerspective ) +
             EvalRookBonus( theBoard, fromWhosePerspective ) +
             EvalKingTropism( theBoard, fromWhosePerspective ) ) >> Grain ) << Grain );
  }
```

```
/**************************************************************************
 * PRIVATE METHODS
 *************************************************************************/

// private EvalKingTropism
// All other things being equal, having your Knights, Queens and Rooks close
// to the opponent's king is a good thing
// This method is a bit slow and dirty, but it gets the job done
private int EvalKingTropism( jcBoard theBoard, int fromWhosePerspective )
{
  int score = 0;

  // Square coordinates
  int kingRank = 0, kingFile = 0;
  int pieceRank = 0, pieceFile = 0;

  if ( fromWhosePerspective == jcPlayer.SIDE_WHITE )
  {
    // Look for enemy king first!
    for( int i = 0; i < 64; i++ )
    {
      if ( theBoard.FindBlackPiece( i ) == jcBoard.BLACK_KING )
      {
        kingRank = i >> 8;
        kingFile = i % 8;
        break;
      }
    }

    // Now, look for pieces which need to be evaluated
    for( int i = 0; i < 64; i++ )
    {
      pieceRank = i >> 8;
      pieceFile = i % 8;
      switch( theBoard.FindWhitePiece( i ) )
      {
        case jcBoard.WHITE_ROOK:
          score -= ( Math.min( Math.abs( kingRank - pieceRank ),
                               Math.abs( kingFile - pieceFile ) ) << 1 );
          break;
        case jcBoard.WHITE_KNIGHT:
          score += 5 - Math.abs( kingRank - pieceRank ) -
                       Math.abs( kingFile - pieceFile );
          break;
        case jcBoard.WHITE_QUEEN:
          score -= Math.min( Math.abs( kingRank - pieceRank ),
                             Math.abs( kingFile - pieceFile ) );
          break;
        default:
          break;
      }
    }
  }
  else
  {
    // Look for enemy king first!
    for( int i = 0; i < 64; i++ )
    {
      if ( theBoard.FindWhitePiece( i ) == jcBoard.WHITE_KING )
      {
        kingRank = i >> 8;
        kingFile = i % 8;
        break;
      }
    }

    // Now, look for pieces which need to be evaluated
    for( int i = 0; i < 64; i++ )
    {
      pieceRank = i >> 8;
```

```
      pieceFile = i % 8;
      switch( theBoard.FindBlackPiece( i ) )
      {
        case jcBoard.BLACK_ROOK:
          score -= ( Math.min( Math.abs( kingRank - pieceRank ),
                               Math.abs( kingFile - pieceFile ) ) << 1 );
          break;
        case jcBoard.BLACK_KNIGHT:
          score += 5 - Math.abs( kingRank - pieceRank ) -
                       Math.abs( kingFile - pieceFile );
          break;
        case jcBoard.BLACK_QUEEN:
          score -= Math.min( Math.abs( kingRank - pieceRank ),
                             Math.abs( kingFile - pieceFile ) );
          break;
        default:
          break;
      }
    }
  }
  return score;
}

// private EvalRookBonus
// Rooks are more effective on the seventh rank, on open files and behind
// passed pawns
private int EvalRookBonus( jcBoard theBoard, int fromWhosePerspective )
{
  long rookboard = theBoard.GetBitBoard( jcBoard.ROOK + fromWhosePerspective );
  if ( rookboard == 0 )
    return 0;

  int score = 0;
  for( int square = 0; square < 64; square++ )
  {
    // Find a rook
    if ( ( rookboard & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // Is this rook on the seventh rank?
      int rank = ( square >> 3 );
      int file = ( square % 8 );
      if ( ( fromWhosePerspective == jcPlayer.SIDE_WHITE ) &&
           ( rank == 1 ) )
        score += 22;
      if ( ( fromWhosePerspective == jcPlayer.SIDE_BLACK ) &&
           ( rank == 7 ) )
        score += 22;

      // Is this rook on a semi- or completely open file?
      if ( MaxPawnFileBins[ file ] == 0 )
      {
        if ( MinPawnFileBins[ file ] == 0 )
          score += 10;
        else
          score += 4;
      }

      // Is this rook behind a passed pawn?
      if ( ( fromWhosePerspective == jcPlayer.SIDE_WHITE ) &&
           ( MaxPassedPawns[ file ] < square ) )
          score += 25;
      if ( ( fromWhosePerspective == jcPlayer.SIDE_BLACK ) &&
           ( MaxPassedPawns[ file ] > square ) )
          score += 25;

      // Use the bitboard erasure trick to avoid looking for additional
      // rooks once they have all been seen
      rookboard ^= jcBoard.SquareBits[ square ];
      if ( rookboard == 0 )
        break;
    }
```

```
  }
  return score;
}

// private EvalDevelopment
// Mostly useful in the opening, this term encourages the machine to move
// its bishops and knights into play, to control the center with its queen's
// and king's pawns, and to castle if the opponent has many major pieces on
// the board
private int EvalDevelopment( jcBoard theBoard, int fromWhosePerspective )
{
  int score = 0;

  if ( fromWhosePerspective == jcPlayer.SIDE_WHITE )
  {
    // Has the machine advanced its center pawns?
    if ( theBoard.FindWhitePiece( 51 ) == jcBoard.WHITE_PAWN )
      score -= 15;
    if ( theBoard.FindWhitePiece( 52 ) == jcBoard.WHITE_PAWN )
      score -= 15;

    // Penalize bishops and knights on the back rank
    for( int square = 56; square < 64; square++ )
    {
      if ( ( theBoard.FindWhitePiece( square ) == jcBoard.WHITE_KNIGHT ) ||
           ( theBoard.FindWhitePiece( square ) == jcBoard.WHITE_BISHOP ) )
        score -= 10;
    }

    // Penalize too-early queen movement
    long queenboard = theBoard.GetBitBoard( jcBoard.WHITE_QUEEN );
    if ( ( queenboard != 0 ) && ( ( queenboard & jcBoard.SquareBits[ 59 ] ) == 0 ) )
    {
      // First, count friendly pieces on their original squares
      int cnt = 0;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_BISHOP ) & jcBoard.SquareBits[ 58 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_BISHOP ) & jcBoard.SquareBits[ 61 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_KNIGHT ) & jcBoard.SquareBits[ 57 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_KNIGHT ) & jcBoard.SquareBits[ 62 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_ROOK ) & jcBoard.SquareBits[ 56 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_ROOK ) & jcBoard.SquareBits[ 63 ] ) != 0 )
        cnt++;
      if ( ( theBoard.GetBitBoard( jcBoard.WHITE_KING ) & jcBoard.SquareBits[ 60 ] ) != 0 )
        cnt++;
      score -= ( cnt << 3 );
    }

    // And finally, incite castling when the enemy has a queen on the board
    // This is a slightly simpler version of a factor used by Cray Blitz
    if ( theBoard.GetBitBoard( jcBoard.BLACK_QUEEN ) != 0 )
    {
      // Being castled deserves a bonus
      if ( theBoard.GetHasCastled( jcPlayer.SIDE_WHITE ) )
        score += 10;
      // small penalty if you can still castle on both sides
      else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_WHITE + jcBoard.CASTLE_QUEENSIDE ) &&
                theBoard.GetCastlingStatus( jcPlayer.SIDE_WHITE + jcBoard.CASTLE_QUEENSIDE ) )
        score -= 24;
      // bigger penalty if you can only castle kingside
      else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_WHITE + jcBoard.CASTLE_KINGSIDE ) )
        score -= 40;
      // bigger penalty if you can only castle queenside
      else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_WHITE + jcBoard.CASTLE_QUEENSIDE ) )
        score -= 80;
      // biggest penalty if you can't castle at all
      else
```

```
        score -= 120;
      }
    }
    else // from black's perspective
    {
      // Has the machine advanced its center pawns?
      if ( theBoard.FindBlackPiece( 11 ) == jcBoard.BLACK_PAWN )
        score -= 15;
      if ( theBoard.FindBlackPiece( 12 ) == jcBoard.BLACK_PAWN )
        score -= 15;

      // Penalize bishops and knights on the back rank
      for( int square = 0; square < 8; square++ )
      {
        if ( ( theBoard.FindBlackPiece( square ) == jcBoard.BLACK_KNIGHT ) ||
             ( theBoard.FindBlackPiece( square ) == jcBoard.BLACK_BISHOP ) )
          score -= 10;
      }

      // Penalize too-early queen movement
      long queenboard = theBoard.GetBitBoard( jcBoard.BLACK_QUEEN );
      if ( ( queenboard != 0 ) && ( ( queenboard & jcBoard.SquareBits[ 3 ] ) == 0 ) )
      {
        // First, count friendly pieces on their original squares
        int cnt = 0;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_BISHOP ) & jcBoard.SquareBits[ 2 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_BISHOP ) & jcBoard.SquareBits[ 5 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_KNIGHT ) & jcBoard.SquareBits[ 1 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_KNIGHT ) & jcBoard.SquareBits[ 6 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_ROOK ) & jcBoard.SquareBits[ 0 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_ROOK ) & jcBoard.SquareBits[ 7 ] ) != 0 )
          cnt++;
        if ( ( theBoard.GetBitBoard( jcBoard.BLACK_KING ) & jcBoard.SquareBits[ 4 ] ) != 0 )
          cnt++;
        score -= ( cnt << 3 );
      }

      // And finally, incite castling when the enemy has a queen on the board
      // This is a slightly simpler version of a factor used by Cray Blitz
      if ( theBoard.GetBitBoard( jcBoard.WHITE_QUEEN ) != 0 )
      {
        // Being castled deserves a bonus
        if ( theBoard.GetHasCastled( jcPlayer.SIDE_BLACK ) )
          score += 10;
        // small penalty if you can still castle on both sides
        else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_BLACK + jcBoard.CASTLE_QUEENSIDE ) &&
                  theBoard.GetCastlingStatus( jcPlayer.SIDE_BLACK + jcBoard.CASTLE_QUEENSIDE ) )
          score -= 24;
        // bigger penalty if you can only castle kingside
        else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_BLACK + jcBoard.CASTLE_KINGSIDE ) )
          score -= 40;
        // bigger penalty if you can only castle queenside
        else if ( theBoard.GetCastlingStatus( jcPlayer.SIDE_BLACK + jcBoard.CASTLE_QUEENSIDE ) )
          score -= 80;
        // biggest penalty if you can't castle at all
        else
          score -= 120;
      }
    }
    return score;
}

// private EvalBadBishops
// If Max has too many pawns on squares of the color of his surviving bishops,
// the bishops may be limited in their movement
private int EvalBadBishops( jcBoard theBoard, int fromWhosePerspective )
```

```
{
  long where = theBoard.GetBitBoard( jcBoard.BISHOP + fromWhosePerspective );
  if ( where == 0 )
    return 0;

  int score = 0;
  for( int square = 0; square < 64; square++ )
  {
    // Find a bishop
    if ( ( where & jcBoard.SquareBits[ square ] ) != 0 )
    {
      // What is the bishop's square color?
      int rank = ( square >> 3 );
      int file = ( square % 8 );
      if ( ( rank % 2 ) == ( file % 2 ) )
        score -= ( MaxPawnColorBins[ 0 ] << 3 );
      else
        score -= ( MaxPawnColorBins[ 1 ] << 3 );

      // Use the bitboard erasure trick to avoid looking for additional
      // bishops once they have all been seen
      where ^= jcBoard.SquareBits[ square ];
      if ( where == 0 )
        break;
    }
  }
  return score;
}

// private EvalPawnStructure
// Given the pawn formations, penalize or bonify the position according to
// the features it contains
private int EvalPawnStructure( int fromWhosePerspective )
{
  int score = 0;

  // First, look for doubled pawns
  // In chess, two or more pawns on the same file usually hinder each other,
  // so we assign a minor penalty
  for( int bin = 0; bin < 8; bin++ )
    if ( MaxPawnFileBins[ bin ] > 1 )
      score -= 8;

  // Now, look for an isolated pawn, i.e., one which has no neighbor pawns
  // capable of protecting it from attack at some point in the future
  if ( ( MaxPawnFileBins[ 0 ] > 0 ) && ( MaxPawnFileBins[ 1 ] == 0 ) )
    score -= 15;
  if ( ( MaxPawnFileBins[ 7 ] > 0 ) && ( MaxPawnFileBins[ 6 ] == 0 ) )
    score -= 15;
  for( int bin = 1; bin < 7; bin++ )
  {
    if ( ( MaxPawnFileBins[ bin ] > 0 ) && ( MaxPawnFileBins[ bin - 1 ] == 0 )
         && ( MaxPawnFileBins[ bin + 1 ] == 0 ) )
      score -= 15;
  }

  // Assign a small penalty to positions in which Max still has all of his
  // pawns; this incites a single pawn trade (to open a file), but not by
  // much
  if ( MaxTotalPawns == 8 )
    score -= 10;

  // Penalize pawn rams, because they restrict movement
  score -= 8 * PawnRams;

  // Finally, look for a passed pawn; i.e., a pawn which can no longer be
  // blocked or attacked by a rival pawn
  if ( fromWhosePerspective == jcPlayer.SIDE_WHITE )
  {
    if ( MaxMostAdvanced[ 0 ] < Math.min( MinMostBackward[ 0 ], MinMostBackward[ 1 ] ) )
      score += ( 8 - ( MaxMostAdvanced[ 0 ] >> 3 ) ) *
```

```
            ( 8 - ( MaxMostAdvanced[ 0 ] >> 3 ) );
    if ( MaxMostAdvanced[ 7 ] < Math.min( MinMostBackward[ 7 ], MinMostBackward[ 6 ] ) )
      score += ( 8 - ( MaxMostAdvanced[ 7 ] >> 3 ) ) *
               ( 8 - ( MaxMostAdvanced[ 7 ] >> 3 ) );
    for( int i = 1; i < 7; i++ )
    {
      if ( ( MaxMostAdvanced[ i ] < MinMostBackward[ i ] ) &&
           ( MaxMostAdvanced[ i ] < MinMostBackward[ i - 1 ] ) &&
           ( MaxMostAdvanced[ i ] < MinMostBackward[ i + 1 ] ) )
        score += ( 8 - ( MaxMostAdvanced[ i ] >> 3 ) ) *
                 ( 8 - ( MaxMostAdvanced[ i ] >> 3 ) );
    }
  }
  else // from Black's perspective
  {
    if ( MaxMostAdvanced[ 0 ] > Math.max( MinMostBackward[ 0 ], MinMostBackward[ 1 ] ) )
      score += ( MaxMostAdvanced[ 0 ] >> 3 ) *
               ( MaxMostAdvanced[ 0 ] >> 3 );
    if ( MaxMostAdvanced[ 7 ] > Math.max( MinMostBackward[ 7 ], MinMostBackward[ 6 ] ) )
      score += ( MaxMostAdvanced[ 7 ] >> 3 ) *
               ( MaxMostAdvanced[ 7 ] >> 3 );
    for( int i = 1; i < 7; i++ )
    {
      if ( ( MaxMostAdvanced[ i ] > MinMostBackward[ i ] ) &&
           ( MaxMostAdvanced[ i ] > MinMostBackward[ i - 1 ] ) &&
           ( MaxMostAdvanced[ i ] > MinMostBackward[ i + 1 ] ) )
        score += ( MaxMostAdvanced[ i ] >> 3 ) *
                 ( MaxMostAdvanced[ i ] >> 3 );
    }
  }

  return score;
}

// private AnalyzePawnStructure
// Look at pawn positions to be able to detect features such as doubled,
// isolated or passed pawns
private boolean AnalyzePawnStructure( jcBoard theBoard, int fromWhosePerspective )
{
  // Reset the counters
  for( int i = 0; i < 8; i++ )
  {
    MaxPawnFileBins[ i ] = 0;
    MinPawnFileBins[ i ] = 0;
  }
  MaxPawnColorBins[ 0 ] = 0;
  MaxPawnColorBins[ 1 ] = 0;
  PawnRams = 0;
  MaxTotalPawns = 0;

  // Now, perform the analysis
  if ( fromWhosePerspective == jcPlayer.SIDE_WHITE )
  {
    for( int i = 0; i < 8; i++ )
    {
      MaxMostAdvanced[ i ] = 63;
      MinMostBackward[ i ] = 63;
      MaxPassedPawns[ i ] = 63;
    }
    for( int square = 55; square >= 8; square-- )
    {
      // Look for a white pawn first, and count its properties
      if ( theBoard.FindWhitePiece( square ) == jcBoard.WHITE_PAWN )
      {
        // What is the pawn's position, in rank-file terms?
        int rank = square >> 3;
        int file = square % 8;

        // This pawn is now the most advanced of all white pawns on its file
        MaxPawnFileBins[ file ]++;
        MaxTotalPawns++;
```

```
          MaxMostAdvanced[ file ] = square;

          // Is this pawn on a white or a black square?
          if ( ( rank % 2 ) == ( file % 2 ) )
            MaxPawnColorBins[ 0 ]++;
          else
            MaxPawnColorBins[ 1 ]++;

          // Look for a "pawn ram", i.e., a situation where a black pawn
          // is located in the square immediately ahead of this one.
          if ( theBoard.FindBlackPiece( square - 8 ) == jcBoard.BLACK_PAWN )
            PawnRams++;
        }
        // Now, look for a BLACK pawn
        else if ( theBoard.FindBlackPiece( square ) == jcBoard.BLACK_PAWN )
        {
          // If the black pawn exists, it is the most backward found so far
          // on its file
          int file = square % 8;
          MinPawnFileBins[ file ]++;
          MinMostBackward[ file ] = square;
        }
      }
    }
    else // Analyze from Black's perspective
    {
      for( int i = 0; i < 8; i++ )
      {
        MaxMostAdvanced[ i ] = 0;
        MaxPassedPawns[ i ] = 0;
        MinMostBackward[ i ] = 0;
      }
      for( int square = 8; square < 56; square++ )
      {
        if ( theBoard.FindBlackPiece( square ) == jcBoard.BLACK_PAWN )
        {
          // What is the pawn's position, in rank-file terms?
          int rank = square >> 3;
          int file = square % 8;

          // This pawn is now the most advanced of all white pawns on its file
          MaxPawnFileBins[ file ]++;
          MaxTotalPawns++;
          MaxMostAdvanced[ file ] = square;

          if ( ( rank % 2 ) == ( file % 2 ) )
            MaxPawnColorBins[ 0 ]++;
          else
            MaxPawnColorBins[ 1 ]++;

          if ( theBoard.FindWhitePiece( square + 8 ) == jcBoard.WHITE_PAWN )
            PawnRams++;
        }
        else if ( theBoard.FindWhitePiece( square ) == jcBoard.WHITE_PAWN )
        {
          int file = square % 8;
          MinPawnFileBins[ file ]++;
          MinMostBackward[ file ] = square;
        }
      }
    }
    return true;
  }
}
```

```
/**************************************************************************
 * jcTranspositionTable - Alphabeta's memory
 * by F.D. Laramee
 *
 * Purpose:
 * There are many ways to transpose (i.e., achieve the same position)
 * via different move sequences in chess and in most other 2-player games.
 * This object allows the AISearchAgent to save its search results, so that
 * transpositions will not have to be searched again.
 *
 * Notes:
 * As it is currently implemented, the transposition table is exclusive to its
 * AI player.  Therefore, if the machine is to play against itself (for
 * example, to test new versions of an evaluation function against an old
 * one), there will be two instances of the table active.  For some types of
 * evaluation functions, it would be easy to share a singleton table between
 * two AI players by adding a flag to each entry to identify from whose
 * perspective the evaluation was performed; if we have evaluated from Black's
 * perspective and need the results from White's, we could simply change the
 * sign of the evaluation, and voila.  However, since my evaluation function is
 * NOT entirely symmetrical (i.e., material value depends on the number of pawns
 * owned by the *winning* side, not necessarily the *moving* side), this might
 * introduce errors in the search process.  Memory being dirt cheap these days,
 * this isn't much of an issue.
 *
 * History
 * 14.08.00 Creation
 **************************************************************************/

package javachess;
import javachess.jcBoard;

/**************************************************************************
 * A small internal class containing an AB value for a given position, and
 * a "hash lock" signature used to identify collisions between board positions
 * with the same basic hashing values.
 *
 * Note that there is no need to store the actual move leading to this value,
 * for two reasons: first, by the time we check on the transposition table, the
 * move has already been applied; second, our version of alphabeta only handles
 * moves themselves at the top level of the search, so this information would
 * be passed to non one!
 **************************************************************************/
class jcTranspositionEntry
{
  // Data fields, beginning with the actual value of the board and whether this
  // value represents an accurate evaluation or only a boundary
  public int theEvalType;
  public int theEval;

  // This value was obtained through a search to what depth?  0 means that
  // it was obtained during quiescence search (which is always effectively
  // of infinite depth but only within the quiescence domain; full-width
  // search of depth 1 is still more valuable than whatever Qsearch result)
  public int theDepth;

  // Board position signature, used to detect collisions
  public long theLock;

  // What this entry stored so long ago that it may no longer be useful?
  // Without this, the table will slowly become clogged with old, deep search
  // results for positions with no chance of happening again, and new positions
  // (specifically the 0-depth quiescence search positions) will never be
  // stored!
  public int timeStamp;

  public static final int NULL_ENTRY = -1;

  // construction
  jcTranspositionEntry()
  {
```

```
    theEvalType = NULL_ENTRY;
  }
}


public class jcTranspositionTable
{
  /**************************************************************************
   * DATA MEMBERS
   *************************************************************************/

  // The size of a transposition table, in entries
  private static final int TABLE_SIZE = 131072;

  // Data
  private jcTranspositionEntry Table[];

  /**************************************************************************
   * PUBLIC METHODS
   *************************************************************************/

  // Construction
  public jcTranspositionTable()
  {
    Table = new jcTranspositionEntry[ TABLE_SIZE ];
    for ( int i = 0; i < TABLE_SIZE; i++ )
    {
      Table[ i ] = new jcTranspositionEntry();
    }
  }

  // boolean LookupBoard( jcBoard theBoard, jcMove theMove )
  // Verify whether there is a stored evaluation for a given board.
  // If so, return TRUE and copy the appropriate values into the
  // output parameter
  public boolean LookupBoard( jcBoard theBoard, jcMove theMove )
  {
    // Find the board's hash position in Table
    int key = Math.abs( theBoard.HashKey() % TABLE_SIZE );
    jcTranspositionEntry entry = Table[ key ];

    // If the entry is an empty placeholder, we don't have a match
    if ( entry.theEvalType == -1 ) // jcTranspositionEntry.NULL_ENTRY )
      return false;

    // Check for a hashing collision!
    if ( entry.theLock != theBoard.HashLock() )
      return false;

    // Now, we know that we have a match!  Copy it into the output parameter
    // and return
    theMove.MoveEvaluation = entry.theEval;
    theMove.MoveEvaluationType = entry.theEvalType;
    theMove.SearchDepth = entry.theDepth;
    return true;
  }

  // public StoreBoard( theBoard, eval, evalType, depth, timeStamp )
  // Store a good evaluation found through alphabeta for a certain board pos
  public boolean StoreBoard( jcBoard theBoard, int eval, int evalType,
    int depth, int timeStamp )
  {
    int key = Math.abs( theBoard.HashKey() % TABLE_SIZE );

    // Would we erase a more useful (i.e., higher) position if we stored this
    // one?  If so, don't bother!
    if ( ( Table[ key ].theEvalType != jcTranspositionEntry.NULL_ENTRY ) &&
         ( Table[ key ].theDepth > depth ) &&
         ( Table[ key ].timeStamp >= timeStamp ) )
      return true;
```

```
    // And now, do the actual work
    Table[ key ].theLock = theBoard.HashLock();
    Table[ key ].theEval = eval;
    Table[ key ].theDepth = depth;
    Table[ key ].theEvalType = evalType;
    Table[ key ].timeStamp = timeStamp;
    return true;
  }
}
```

```
/*****************************************************************************
 * jcOpeningBook - A hash table of well-known positions and moves
 *
 * Chess programs are notoriously bad at deciding what to do with complicated
 * positions, so everyone "cheats" by giving them a library of opening positions
 * taken from the ECO or something like that.  This one is very primitive and
 * contains very little, but it gets the job done.
 *
 * History:
 * 19.09.00 Creation
 *
 *****************************************************************************/

package javachess;
import javachess.jcMove;
import javachess.jcBoard;
import java.io.*;

/*****************************************************************************
 * PRIVATE class jcOpeningBookEntry
 * A signature for a board position, and the best moves for White and Black
 * in that position.
 *****************************************************************************/

class jcOpeningBookEntry
{
  // A signature for the board position stored in the entry
  int theLock;

  // Moves
  jcMove WhiteMove;
  jcMove BlackMove;

  // A sentinel indicating that a move is invalid
  public static final int NO_MOVE = -1;

  // Construction
  jcOpeningBookEntry()
  {
    theLock = 0;
    WhiteMove = new jcMove();
    WhiteMove.MoveType = NO_MOVE;
    BlackMove = new jcMove();
    BlackMove.MoveType = NO_MOVE;
  }
}


/*****************************************************************************
 * PUBLIC class jcOpeningBook
 * A hash table containing a certain number of slots for well-known positions
 *****************************************************************************/

public class jcOpeningBook
{
  // The hash table itself
  private static final int TABLE_SIZE = 1024;
  private jcOpeningBookEntry Table[];

  // Construction
  public jcOpeningBook()
  {
    Table = new jcOpeningBookEntry[ TABLE_SIZE ];
    for ( int i = 0; i < TABLE_SIZE; i++ )
    {
      Table[ i ] = new jcOpeningBookEntry();
    }
  }

  // public jcMove Query
  // Querying the table for a ready-made move to play.  Return null if there
```

```
// is none
public jcMove Query( jcBoard theBoard )
{
  // First, look for a match in the table
  int key = Math.abs( theBoard.HashKey() % TABLE_SIZE );
  int lock = theBoard.HashLock();

  // If the hash lock doesn't match the one for our position, get out
  if ( Table[ key ].theLock != lock )
    return null;

  // If there is an entry for this board in the table, verify that it
  // contains a move for the current side
  if ( theBoard.GetCurrentPlayer() == jcPlayer.SIDE_BLACK )
  {
    if ( Table[ key ].BlackMove.MoveType != jcOpeningBookEntry.NO_MOVE )
      return Table[ key ].BlackMove;
  }
  else
  {
    if ( Table[ key ].WhiteMove.MoveType != jcOpeningBookEntry.NO_MOVE )
      return Table[ key ].WhiteMove;
  }

  // If we haven't found anything useful, quit
  return null;
}

// Loading the table from a file
public boolean Load( String fileName ) throws Exception
{
  // Open the file as a Java tokenizer
  FileReader fr = new FileReader( fileName );
  StreamTokenizer tok = new StreamTokenizer( fr );
  tok.eolIsSignificant( false );
  tok.lowerCaseMode( false );

  // Create a game board on which to "play" the opening sequences stored in
  // the book, so that we know which position to associate with which move
  jcBoard board = new jcBoard();
  jcMove mov = new jcMove();
  jcMoveListGenerator successors = new jcMoveListGenerator();

  // How many lines of play do we have in the book?
  tok.nextToken();
  int numLines = (int) tok.nval;

  for( int wak = 0; wak < numLines; wak++ )
  {
    // Begin the line of play with a clean board
    board.StartingBoard();

    // Load the continuation
    while( true )
    {
      successors.ComputeLegalMoves( board );

      // Is the token an end-of-continuation marker?
      // If so, go on to the next continuation
      if( ( tok.nextToken() == StreamTokenizer.TT_WORD ) && ( tok.sval.equalsIgnoreCase( "END" ) ) )
      {
        break;
      }

      if ( tok.ttype == StreamTokenizer.TT_EOL )
        tok.nextToken();

      // If not, gather the source and destination squares of the next move
      int source = (int) tok.nval;
      tok.nextToken();
      int destination = (int) tok.nval;
```

```
        // Make a jcMove structure out of the source and destination squares;
        // this determines whether there is a capture involved, a castling, etc.
        mov = successors.FindMoveForSquares( source, destination );

        // And now, store the move in the table
        StoreMove( board, mov );

        // Finally, apply the move and get ready for the next one
        board.ApplyMove( mov );

      }
    }

    fr.close();
    return true;
  }


  // private StoreMove( jcBoard, jcMov )
  private boolean StoreMove( jcBoard theBoard, jcMove theMove )
  {
    // Where should we store this data?
    int key = Math.abs( theBoard.HashKey() % TABLE_SIZE );
    int lock = theBoard.HashLock();

    // Is there already an entry for a different board position where we
    // want to put this?  If so, mark it deleted
    if ( Table[ key ].theLock != lock )
    {
      Table[ key ].BlackMove.MoveType = jcOpeningBookEntry.NO_MOVE;
      Table[ key ].WhiteMove.MoveType = jcOpeningBookEntry.NO_MOVE;
    }

    // And store the new move
    Table[ key ].theLock = lock;
    if ( theBoard.GetCurrentPlayer() == jcPlayer.SIDE_BLACK )
    {
      Table[ key ].BlackMove.Copy( theMove );
    }
    else
    {
      Table[ key ].WhiteMove.Copy( theMove );
    }

    return true;
  }
}
```