Lurgee

The Lurgee project consists of a Java framework for abstract strategy games and some example games built with the framework. Wikipedia defines an abstract strategy game as follows:

An abstract strategy game is a board game with perfect information, no chance, and (usually) two players. Many of the world's classic board games, including chess, go and mancala, fit into this category. Play is sometimes said to resemble a series of puzzles the players pose to each other.

The strategy game framework provides the means for defining the entities that comprise the game (board, players, moves); the logic for determining what is a good position to be in; and the engine for determining which is the best move to play based on the state of play. The complexity of the game is mostly irrelevant - this framework could be used to implement simple games such as tic-tac-toe or more complex games such as chess.

Additional code is provided with the framework, which illustrates its use, including fully-featured reversi, connect-four and nine men's morris applets.

Objectives

The objectives of this project are to:

- Provide a resource to others interested in abstract strategy games.
- Use the framework to implement several different games.
- Use the framework on a variety of platforms.

Abstract Strategy Games

Elements

Player

An abstract strategy game is generally played between two players. The aim of each player is to make their position in the game more favourable in an effort to achieve the objective of the game, whilst making the opponent's position less favourable. Applications using the strategy game framework will generally have one of the players controlled by the user and the other by the computer, or have both players controlled by the computer (for example, to assess the effectiveness of various evaluation methods).

Board

The board is the playing area where the two players meet. The rules of the game dictate the characteristics of the board and what moves each player can make on the board. At any particular time, it is the turn of one of the players to make a move.

Move

An abstract strategy game involves each player making a move in turn. A move may involve placing one or more pieces on the board, moving existing pieces to different positions on the board, removing pieces from the board, capturing opponent pieces, flipping opponent pieces, and so on. The rules of the game dictate which moves are valid based on the state of the board and which player's turn it is. The conditions for the end of the game may include one or both of the players not being able to make a move.

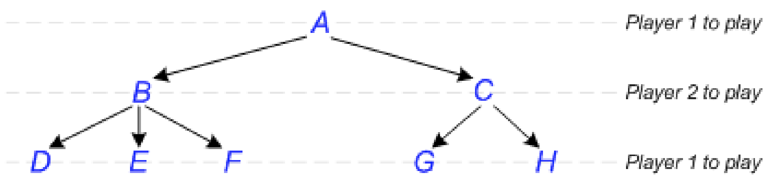
Objective

The objective of most abstract strategy games is to achieve a particular game state, such as the checkmate state in chess, or the removal of all the opponent's pieces from the board in nine men's morris (or mills). Some games may assign a score to each player, with the objective being to achieve the end-game state with a higher score than the opponent.

Game Trees

At any point in an abstract strategy game, the board is in a particular state and it is the turn of one of the players to make a move. There may be several possible moves that may be made. For each of these moves, the opponent will have a set of possible moves that they can play. This situation may be represented by a tree, with each state of the game represented as a node in the tree.

For example, in the diagram below, the game is initially in state A, with player 1 (the *initial player*) to play. There are two possible moves, the first of which changes the game state to state B and the second to state C with player 2 to play. There are three possible moves from state B, and two possible moves from state C.



The above game tree is only plotted two moves ahead. As the tree is plotted deeper, the number of nodes increases exponentially. Almost all algorithms for determining the best move to play in an abstract strategy game generate a game tree and evaluate the game states at the leaves (the nodes in the tree that have no successors) to determine which branch, and hence move, is the most advantageous to play. In a simple game such a tic-tac-toe a complete game tree could be quickly generated by a computer for the entire game; however in a more complex game, there are too many possible game states (a full game tree for connect-four has in the order of 10^{14} nodes, reversi 10^{28} nodes and chess 10^{46} - this is referred to as the *state-space complexity* of the game). As a result, a computer uses a subset of the full tree to a predetermined depth.

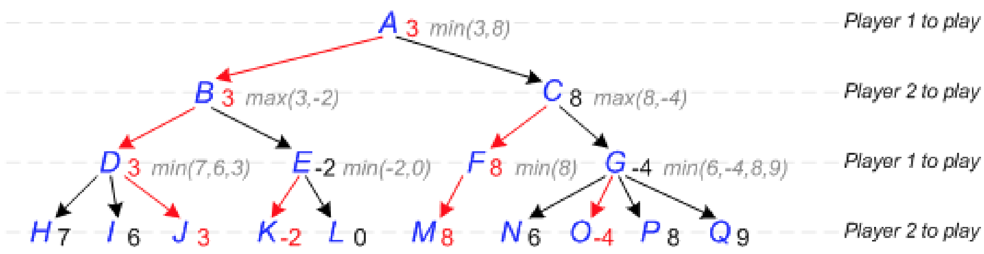
Evaluating the Game State

In order to determine if a move is a good one, a quantitative measure of the game state is required. Scoring the game state is a key aspect of searching game trees. Only the leaves of the tree are scored, as these represent the game state after each of the possible moves are played, up to the tree depth (or earlier if a particular game state results in the game ending - see No-Move Scenarios below). Scoring is done for a specific player as what is good for one player is usually not good for the other, with higher scores representing better positions.

Searching a Game Tree With Minimax

In an abstract strategy game with two players, the players make moves alternately with each player attempting to improve their chance of winning, while decreasing the opponent's chance. If each player makes the best move available to them from each game state, the path through the tree which represents the best end game state for the initial player may be determined. This relies on picking the best move from each node.

An algorithm that implements this approach is *minimax*. The leaves are scored and each node in the tree is assigned the maximum score (for nodes representing the player's moves) or the minimum score (for nodes representing the opponent's moves) of the nodes directly under them. In the example below, which is done to a depth of 3, the move to state B is determined to be the best move from state A.



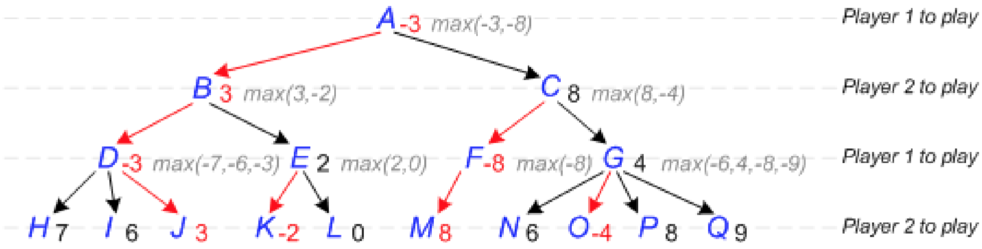
Pseudo-code for the minimax algorithm is as follows:

```
minimax(node)
    if node is a leaf
        return an evaluated score for the node
    if node is a minimising node
        minscore = +infinity;
        for each child of node
            minscore = min(minscore, minimax(child))
        return minscore
    if node is a maximising node
        maxscore = -infinity;
        for each child of node
            maxscore = max(maxscore, minimax(child))
        return maxscore
```

Game trees are generally searched depth-first, which allows large trees to be searched as it only requires enough memory corresponding to the depth of the tree (there is never a requirement to have more than the tree-depth number of game states in memory).

Searching a Game Tree with Negamax

Negamax is an extension of the minimax algorithm, where the maximum score is always selected for every node; however scores from child nodes are negated by each node.



Pseudo-code for the negamax algorithm is as follows:

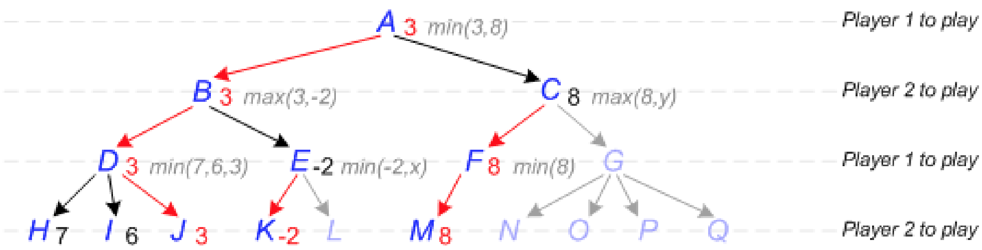
```
negamax(node)
    if node is a leaf
        return an evaluated score for the node
    maxscore = -infinity;
    for each child of node
        maxscore = max(maxscore, -negamax(child))
    return maxscore
```

Optimising the Search Process

It is not always necessary to traverse the entire game tree when performing a search. Optimisations may be applied that decrease the number of evaluations that are done. This is advantageous as every evaluation done increases the time it takes for the computer to determine a move to play.

Alpha-Beta Pruning

Alpha-beta pruning involves ignoring parts of the tree that will have no effect on the search process. For example, `max(3, min(-2, x))` and `min(3, max(8, y))` are both always 3, irrespective of the value of x and y. These two situations can be applied to the nodes for game states B/E and A/C in the minimax game tree below respectively. This implies that the branches to game states L and G (and therefore everything that follows them too) are irrelevant to the outcome and can be pruned. The result on the tree below is that five of the evaluations which would be done by minimax are not necessary.



In order to determine when a branch may be pruned, a search window bound by an upper and lower value (alpha and beta) is maintained. Anything outside of the search window (less than alpha or greater than beta) is pruned. Alpha and beta are changed as the tree is searched, according to the values encountered. Pseudo-code for alpha-beta with minimax is as follows (with initial values of alpha and beta being `-infinity` and `+infinity` respectively):

```
minimax(node, alpha, beta)
    if node is a leaf
        return an evaluated score for the node
    if node is a minimising node
        minscore = beta
        for each child of node
            minscore = min(minscore, minimax(child, alpha, beta))
            beta = minscore
            if alpha >= beta
                return minscore     // cut-off
        return minscore
    if node is a maximising node
            maxscore = alpha
            for each child of node
                maxscore = max(maxscore, minimax(child, alpha, beta))
                alpha = maxscore
                if alpha >= beta
                    return maxscore     // cut-off
            return maxscore
```

Alpha-beta pruning may be applied to negamax too by negating and swapping the alpha and beta values when passing them to a child node. Pseudo-code for alpha-beta with negamax is as follows (again, with initial values of alpha and beta being `-infinity` and `+infinity` respectively):

```
negamax(node, alpha, beta)
    if node is a leaf
        return an evaluated score for the node
    maxscore = alpha
    for each child of node
        maxscore = max(maxscore, -negamax(child, -beta, -alpha))
        alpha = maxscore
        if alpha >= beta
            return alpha    // cut-off
    return alpha
```

Move Ordering

The efficiency of alpha-beta pruning is affected by the order in which nodes are examined. Adjusting the order so that moves that are more likely to be good moves are examined first increases the probability of alpha-beta cut-offs happening early. A separate heuristic is used to perform the ordering; for example, in a chess game, moves that result in pieces being captured might be examined before moves that don't.

Other Optimisations

Iterative deepening involves starting with a tree depth of 1 and determining the best move, then increasing the tree depth and determining the best move again, and so on, until a particular criteria is met (for example, a predetermined period of time has passed or a maximum number of board evaluations has been done), at which point the best move determined to that point is returned. This allows the amount of time or effort for a tree search to be constrained (although the depth reached will vary depending on the number of nodes in the tree) and can also improve the efficiency of alpha-beta cut-offs if the results of each iteration are used to order the moves for the next iteration. If move ordering is done effectively iterative deepening can even be more efficient than a single pass search. Iterative deepening effectively allows a depth-first search to become a breadth-first search whilst retaining the properties of a depth-first search, such as low memory requirements.

The killer heuristic involves examining the last move that caused a beta cut-off at the same level in the tree search first, as it is likely that this move will result in a cut-off again. Put differently, a move that got a very good score at a particular search depth is likely to get a good score at this depth again. Although this is an assumption and not always true, and bearing in mind that a particular move at a specific depth may be considered many times during a search, it is generally true often enough for this small optimisation to have a big effect; particularly when iterative deepening is being used. Storing more than one killer move per level can further increase the effectiveness of this optimisation.

Refutation tables are a generalisation of the killer heuristic, where sub-tree information is recorded for specific game states. If that particular state is encountered again during a tree search, this information may be reused without having to generate and search this sub-tree, or if the depth was not sufficient, the stored information may be used to order the child nodes.

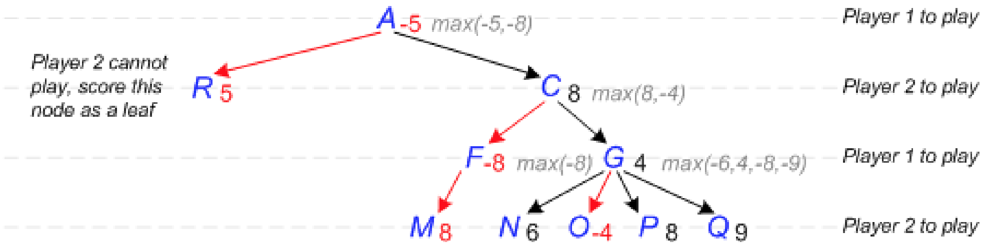
Searching a Game Tree with Negascout

Negascout is also an extension of the minimax algorithm and can be faster than negamax. Its principle is to narrow the alpha-beta search window to increase the possibility of a cut-off. Negascout uses a narrowed search window to perform an initial search and then a subsequent search with a wide window if values are found to lie outside of the initial window. Negascout is generally only an improvement on alpha-beta pruning if move ordering is applied effectively, and is particularly effective when used in conjunction with iterative deepening and associated move ordering. Conversely, negascout may be less efficient than negamax if move ordering is not applied effectively. Pseudo-code for negascout is as follows (again, with initial values of alpha and beta being `-infinity` and `+infinity` respectively):

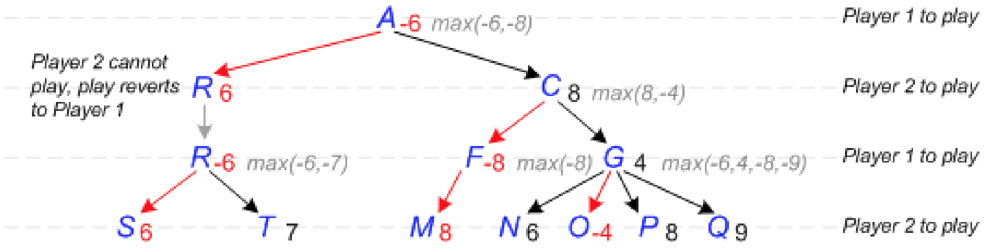
```
negascout(node, alpha, beta)
    if node is a leaf
        return an evaluated score for the node
    maxscore = alpha
    b = beta
    for each child of node
        v = -negascout(child, -b, -alpha)
        if alpha < v < beta and not the first child and depth > 1
            v = -negascout(child, -beta, -v)    // re-search
        alpha = max(alpha, v)
        if alpha >= beta
            return alpha    // cut-off
        b = alpha + 1    // set new null window
    return alpha
```

No-Move Scenarios

Circumstances may arise where a player cannot make a move. Depending on the game, this might represent an end-game state or play might revert to the next player to go (a 'bye'). These scenarios are handled quite differently. The first scenario means that no further branches exist off this node, so the node should be considered to be a leaf and evaluated at this point, as shown in the negamax tree below.



The scenario where a bye is allowed can be handled by creating a single branch from this point, keeping the board state the same and changing the player, as shown in the negamax tree below.



The Strategy Game Framework

The interfaces and classes that comprise the strategy game framework all reside in the `net.lurgee.sgf` package. The code may be found in the `sgf` directory in the source code. The structure of the `sgf` directory is as follows:

- `src` - Java source for the strategy game framework.
- `test-src` - Unit tests for the strategy game framework based on JUnit.
- `doc-res` - Resources referenced by javadoc comments in the code.
- `bin` - Target directory for classes when the code is built with Ant.
- `test-bin` - Target directory for unit tests when the code is built with Ant.
- `test-results` - Target directory for JUnit test results when the code is built with Ant.
- `doc` - Target directory for javadocs when the code is build with Ant.

The compiled classes are not packaged into a jar file, as it is more convenient to include the required classes directly when creating applets or midlets.

Features

The features of the strategy game framework include:

- Extensible, reusable framework for abstract strategy games.
- Lightweight and efficient.
- Employs a variety of search mechanisms, which can be tailored to the game being implemented.
- Includes supporting classes for rapid implementation of Java console applications and applets using the framework.

Object Pooling

The recursive process of searching game trees results in the creation of many short-lived objects, which keeps the garbage collector very busy and can have a detrimental affect on performance. It is therefore advantageous to reuse short-lived objects (especially those whose construction is expensive) as much as possible rather than creating new ones. For this reason, the strategy game framework employs an object pool, which maintains a pool of these objects for reuse.

The `ObjectPool` class realises this functionality by pooling homogenous objects that implement the *`Poolable`* interface. The responsibility lies with the code that uses objects from the pool to check the objects out of the pool, and to then check them back into the pool when it is done with them. Not checking objects back into the pool will result in a memory leak. `ObjectPool` creates new objects by reflection as they are required, so that the pool only contains the maximum number of objects that are required concurrently and no more.

Game Elements

Player

The *`Player`* interface represents a player in the game and should be implemented by games that use the strategy game framework to encapsulate the required properties of a player (such as the colour of the player's pieces).

Board

The *`AbstractBoard`* abstract class represents the board in the game and should be subclassed by games that use the strategy game framework to encapsulate the behaviour of the game being implemented. *`AbstractBoard`* implements the *`Poolable`* interface, as the game tree search process requires many short-lived boards, which are obtained from an object pool. *`AbstractBoard`* keeps track of the player whose turn it is to go, the last move that was played on the board and whether or not the game has ended.

The *`Position`* interface represents a position on the board and only needs to be implemented if used. *`Position`* is a **marker** interface - it does not specify any methods that need to be implemented. A marker interface is used to convey **intention** throughout the rest of the framework, which would not be evident if just `Objects` were passed around.

Move

The *`Move`* interface represents a move in the game and should be implemented by games that use the strategy game framework to encapsulate the required properties of a move (such as the start and end position on the board of the piece being played). *`Move`* is a **marker** interface - it does not specify any methods that need to be implemented. A marker interface is used to convey **intention** throughout the rest of the framework, which would not be evident if just `Objects` were passed around.

A factory class implementing the `MoveFactory` is used to create moves. This allows for reuse of immutable move objects in games that are suitable - these are games with a small finite set of possible moves (like reversi, which has 60 and connect-four, which has 7).

Evaluating Board State

Evaluator

An implementation of the *`Evaluator`* interface is used to score the game state for leaf nodes in a game tree. A game that uses the strategy game framework needs to implement this interface in a manner appropriate to the game being built. The range of scores returned by the evaluator is determined by the implementation.

Library

An implementation of the *`Library`* interface is used to select some moves from a library rather than generating and searching a game tree. A game that uses the strategy game framework that requires a move library (such as an opening book) needs to implement this interface in a manner appropriate to the game being built.

Game Engine

Searchers

The engine of the strategy game framework is the *`AbstractSearcher`* abstract class, which defines the base contract for all searchers that find the best move given a particular game state, and provides common functionality. Two subclasses are defined: *`AbstractSinglePassSearcher`*, which defines the base contract for a single-pass searcher, and `IterativeSearcher`, which implements iterative-deepening using an instance of a *`AbstractSinglePassSearcher`* to perform the actual searches.

`AbstractSinglePassSearcher` brings together the game elements, an evaluator and a move library to allow a move to be selected for a game state either by generation and search of a game tree or by lookup in a move library. Two subclasses are defined: `NegamaxSearcher`, which provides an implementation of a negamax-based game tree search with or without alpha-beta pruning, and `NegascoutSearcher`, which extends `NegamaxSearcher` to provide an implementation of a negascout-based game tree search.

Move Ranking

Ranking moves improves the search process. The *`MoveRanker`* interface defines a means to generically rank moves and should be implemented by games that use the strategy game framework to encapsulate a suitable ranking mechanism for the game. The interface includes a callback method that is invoked during the search process to allow move rankers to alter their state based on information gleaned during the search process. The `KillerHeuristicMoveRanker` class implements a move ranker that does just this to rank 'killer moves' appropriately so that they are searched first. `KillerHeuristicMoveRanker` can delegate ranking to another ranker if it is determined to not be a 'killer move'.

The `MoveList` class stores a list of moves ordered by rank. It can be configured to randomly rank moves with equal rank.

Listeners

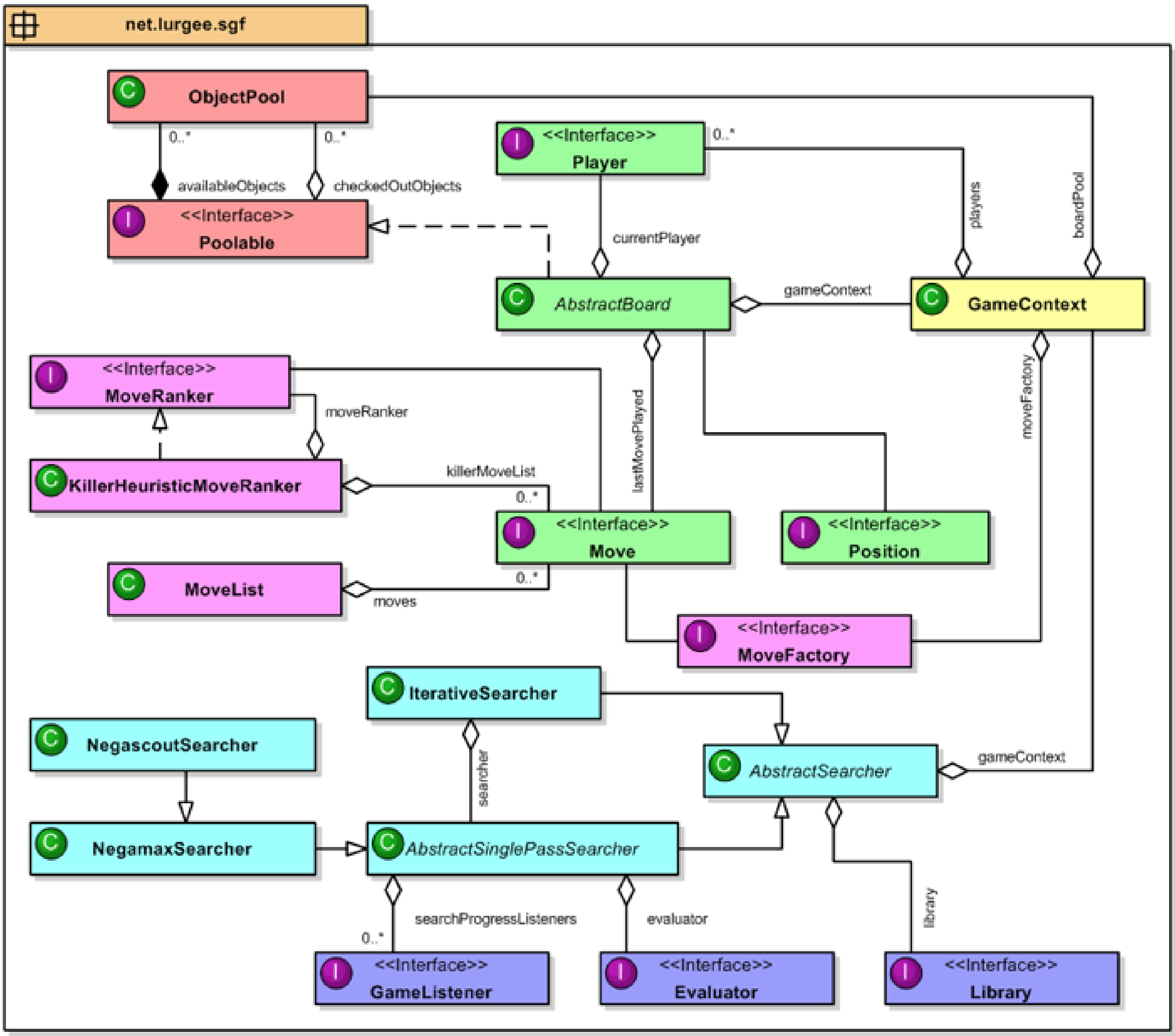
As mentioned, implementations of *`MoveRanker`* used in a search are provided with feedback regarding the search process - specifically node evaluation. In addition, implementations of the *`SearchProgressListener`* interface may be provided to a searcher, which are provided with feedback too - iteration start/end, branching, node evaluation and leaf evaluation.

Other Artifacts

The `GameContext` class encapsulates global game information such as the list of players taking part in the game, and provides factories/pools for obtaining game entities.

Class Diagram

The UML sketch class diagram below shows the key classes that comprise the strategy game framework.



Common Classes

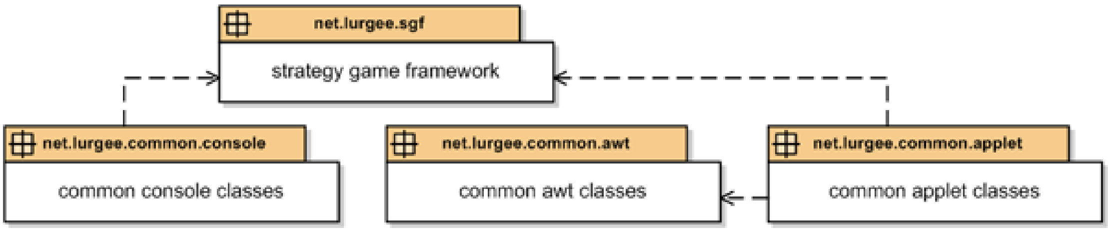
None of these classes are required by the strategy game framework itself, but were generated simply to ease the future development of applications. Together these classes provide a wider framework for quickly implementing console applications and applets using the strategy game framework.

The common classes may be found in the `common` directory in the source code. The structure of the `common` directory is as follows:

- `src` - Java source for the common classes.
- `test-src` - Unit tests for the common classes based on JUnit.
- `bin` - Target directory for classes when the code is built with Ant.
- `test-bin` - Target directory for unit tests when the code is built with Ant.
- `test-results` - Target directory for JUnit test results when the code is built with Ant.
- `doc` - Target directory for javadocs when the code is build with Ant.

Package Diagram

The package diagram below shows the package dependencies for the common classes.



Common AWT Classes

The common AWT classes provide a small set of GUI components for use in AWT applications or applets and reside in the `net.lurgee.common.awt` package. These classes are completely independent of the strategy game framework.

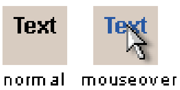
Widget

The *`Widget`* abstract class is a general-purpose GUI component, extended from `java.awt.Container`. It provides methods to load and unload images and to generate custom widget events. The class includes mouse event support by implementing the *`java.awt.event.MouseListener`* and *`java.awt.event.MouseMotionListener`* interfaces. As these interfaces are implemented by this base class, subclasses only need to override the mouse event methods relevant to them. All the common AWT classes are subclasses of *`Widget`*.

Text Widget

The *`TextWidget`* abstract class is a simple element that displays a text string whose colour is changed when the mouse cursor passes over it. The `selectAction` abstract method needs to be defined by concrete subclasses to provide the actions that are performed when a mouse button is pressed on the widget. The dimensions of the text widget and its position relative to the parent widget are specified on construction.

For example:

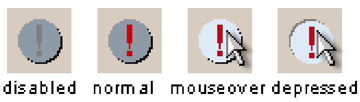


Button Widgets

The *`ButtonWidget`* abstract class is a generic button whose display can be changed if it is disabled, if the mouse cursor is over it, or if it has been pressed. The `paint` and `selectAction` abstract methods needs to be defined by concrete subclasses to paint the button appropriately and to provide the actions that are performed when a mouse button is pressed on the widget respectively. The dimensions of the button widget and its position relative to the parent widget, as well as the colours and the font to use are specified on construction.

The *`IconWidget`* abstract class extends *`ButtonWidget`* to provide a graphical icon-type button. Images are specified for the icon and optionally for its background. The icon image can be changed when the widget is disabled and the background image can be changed when the mouse cursor is over the icon (although only when the widget is enabled). Of the super class abstract methods, only the `paint` method is defined, so the `selectAction` abstract method needs to be defined by concrete subclasses to provide the actions that are performed when a mouse button is pressed on the widget. The dimensions of the icon widget and its position relative to the parent widget, as well as the filenames for each of the images are specified on construction.

For example:

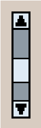


The *`StatefulIconWidget`* abstract class extends *`IconWidget`* to provide an icon whose state changes every time it is clicked. An array of images is provided to it, each of which represents an icon state. The `selectAction` abstract method needs to be defined by concrete subclasses to provide the actions that are performed when a mouse button is pressed on the widget. The dimensions of the stateful icon widget and its position relative to the parent widget, as well as the filenames for each of the images are specified on construction.

Scrollbar Widget

The `ScrollbarWidget` class extends *`Widget`* to provide a graphical vertical scrollbar. The minimum and maximum values represented by the scrollbar are specified, as well as the current value. The position and size of the scroller thumb in the scrollbar are determined appropriately. The value represented by the scrollbar may be changed by dragging the scroller thumb with the mouse, by clicking the up and down buttons, or by clicking in the scrollbar above or below the scroller thumb. The up and down buttons, and the scroller thumb are implemented as inner classes that extend *`ButtonWidget`*. The dimensions of the scrollbar widget and its position relative to the parent widget, as well as the range of values represented by the scrollbar and the colours to use are specified on construction.

For example:



Window

The *`Window`* abstract class is a general-purpose modal window for use in applets, extended from *`Widget`*. All the windows in the common AWT classes are subclasses of *`Window`*. It provides methods to open and close the window, which also sets the properties of the parent widget appropriately so that the window is modal (i.e. so the parent widget does not respond to certain mouse events while the child window is open).

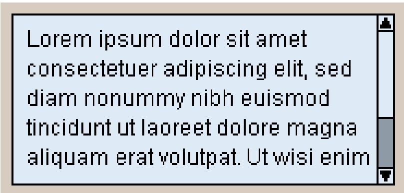
Main Window

The *`MainWindow`* abstract class extends *`Window`* to provide event dispatching functionality of custom widget events. This allows applets to be built more completely in an event-driven way. It also provides functionality for playing audio clips. Generally an applet should have one main window, which is the lowest level GUI component, and all other components should exist as children (or children of children, and so on) of the main window. This layout is needed to ensure that dispatching of widget events operates as intended.

Text Window

The `TextWindow` class extends *`Window`* to provide a window for displaying text. The text is wrapped to fit into the window horizontally and a vertical scrollbar is shown if required. The dimensions of the window are specified, but not the position as the window is always centred in the parent widget or the applet if there is no parent widget. The dimensions of the text window and the text to display, as well as the colours and the font to use are specified on construction.

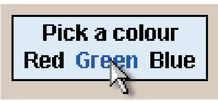
For example:



Option Window

The *`OptionWindow`* abstract class extends *`Window`* to provide a window which presents the user with a list of text options that they can select from. The options are implemented by anonymous classes that extend *`TextWidget`*. The dimensions of the window are not specified, but are rather determined automatically to fit the text and the options. The window is always centred in the parent widget or the applet if there is no parent widget. The `selectAction` abstract method needs to be defined by concrete subclasses to provide the actions that are performed when a mouse button is pressed on one of the options. The text to display and the options the user can select from, as well as the colours and the font to use are specified on construction.

For example:

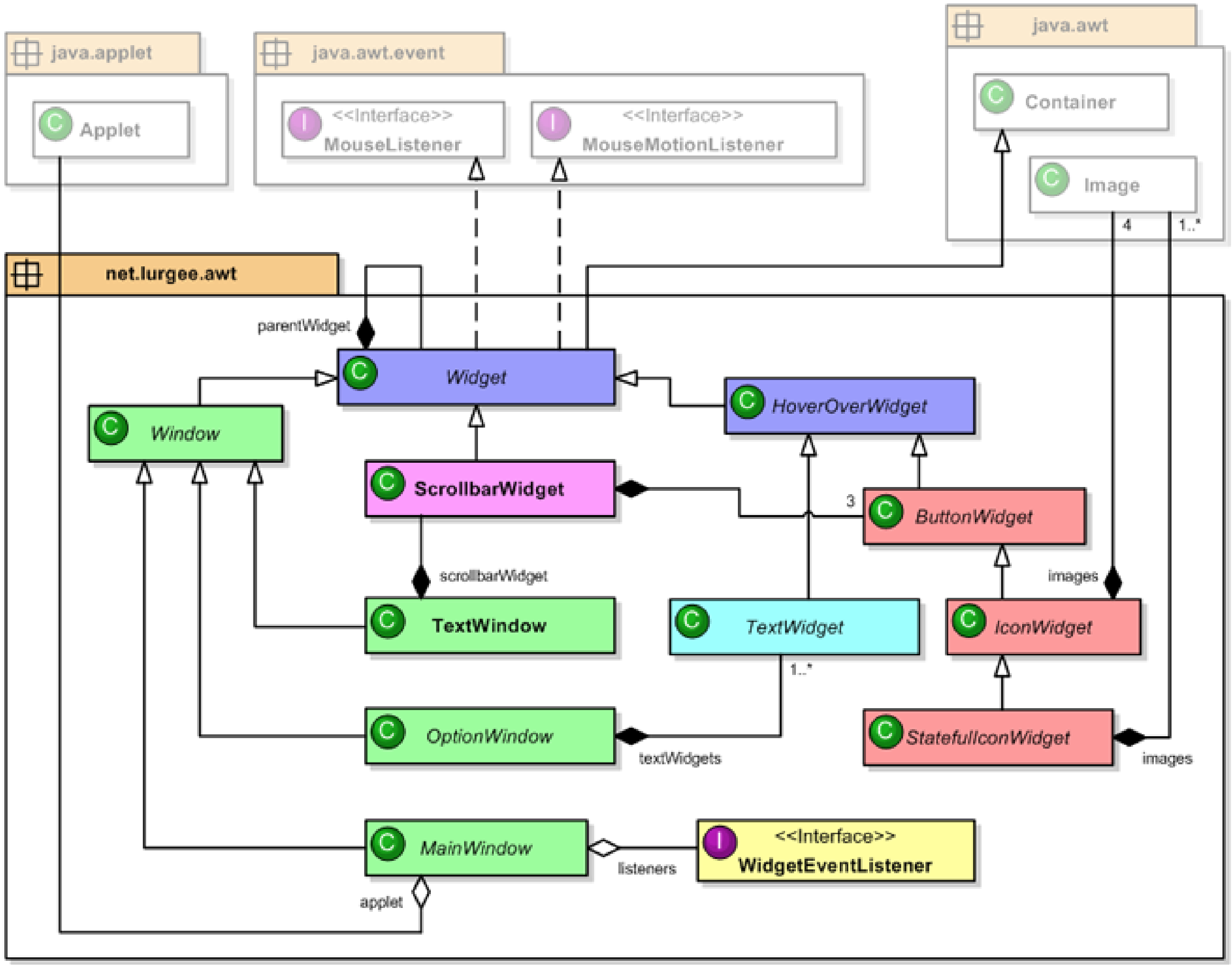


Widget Events

The `WidgetEvent` class extends the Java `AWTEvent` class and represents a generic event that can be dispatched to widgets. Events are posted through a method on any *`Widget`* class and are passed up to the main window, which posts them to the AWT system event queue. When an event fires, children widgets of the main window (or children of children, and so on) that are registered as widget event listeners are informed of the event. This allows events to be globally sent to all interested widgets, easing development and facilitating looser coupling of classes. Data may be attached to events too.

Class Diagram

The UML sketch class diagram below shows the key classes that comprise the common AWT classes.



Common Applet Classes

The common applet classes provide abstracted functionality for creating an applet that uses the strategy game framework. The classes use both the strategy game framework classes and the common AWT classes.

Game

The *AbstractGame* abstract class is simply a container for wrapping all the game-related entities, such as a searcher and a board. It keeps a history of the game played by stored the board states after every move played and facilitates 'undo' operations. It also keeps the game state and a message regarding the current game state. Abstract methods are defined that concrete classes need to define appropriately for the game being implemented.

The `Settings` class keeps track of which player is being controlled by the user, the level that the computer is playing at and whether sound effects are on or off. The *AbstractGame* abstract class has an instance of the `Settings` class.

Game Applet

The *AbstractGameApplet* abstract class extends the `java.applet.Applet` class and provides common functionality that all applets require. In particular, it over-rides the default painting mechanism to provide support for double-buffering for a smoother user interface without flicker.

Game Window

The *AbstractGameWindow* abstract class extends the common AWT *MainWindow* class. It it an abstraction of a general game window and performs operations that are common to all games, including flow control and icon management. Concrete classes that extend this class need to do very little more than setup the components and paint the window to implement a functional game.

Board Widget

The *AbstractBoardWidget* abstract class extends the common AWT *Widget* class. It it an abstraction of a general widget for game board and performs operations that are common to all games, including animation control and the dispatching of common events. This class should be extended appropriately for the game being implemented.

Thinker

The `Thinker` class implements the *java.lang.Runnable* interface to allow the computer's search for a move to be done in a thread.

Animator

The `Animator` class implements the *`java.lang.Runnable`* interface to allow animations (such as flipping pieces in reversi, or sliding pieces into place in connect-four) to be done in a thread. An animator is able to animate any class that implements the *`Animatable`* interface, by repeatedly calling the `animate` method on this class until the thread is terminated or the animatable class indicates that the animation is complete.

Events

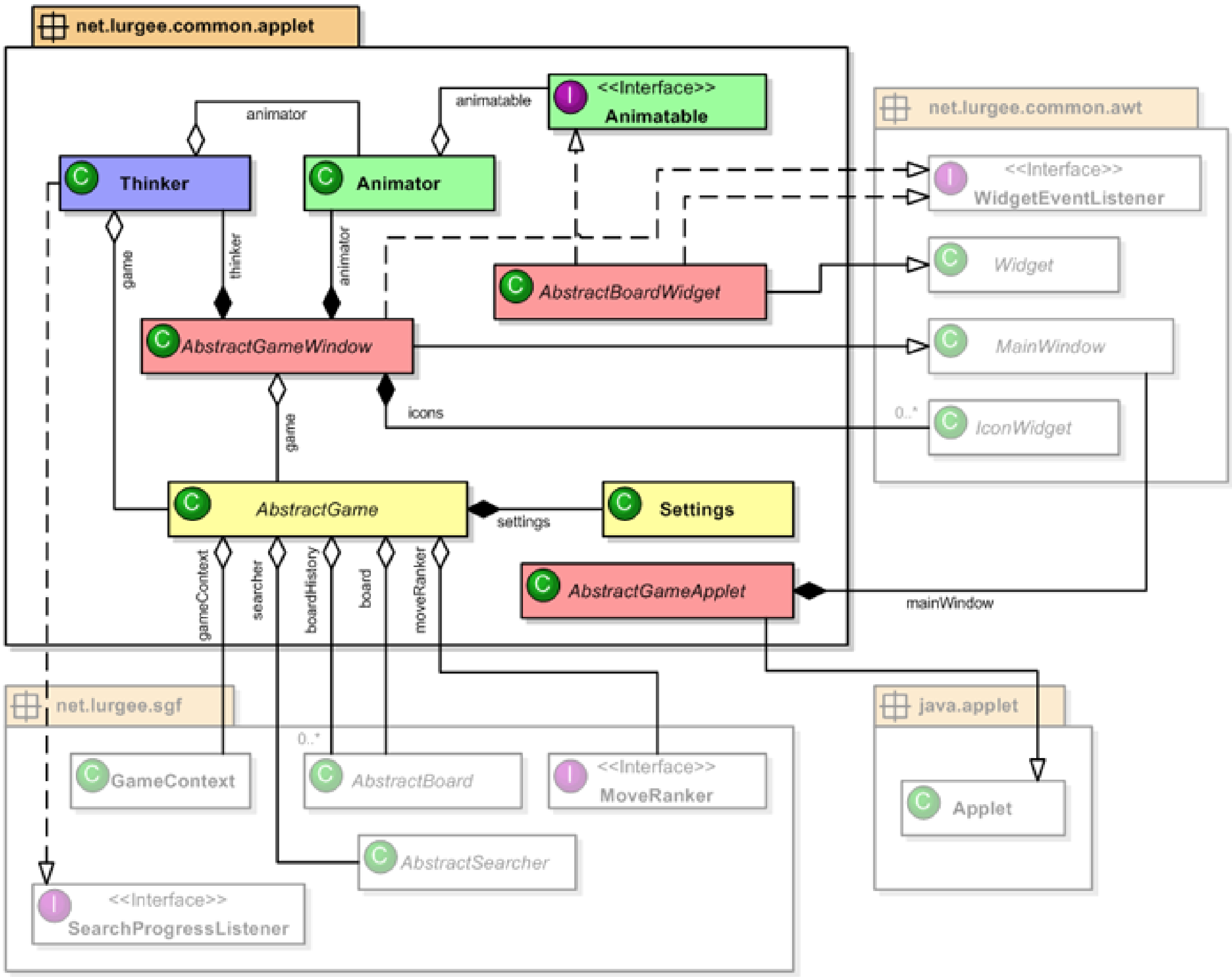
Various widget events are posted and handled during the execution of a game. The flow control is almost entirely based on these events. The event ids are defined in the `Event` class as follows:

Event id	Description	Data
`PLAY_MOVE`	Play a move	`Move` representing the move to be played
`DONE_PLAYING`	A move has been played or aborted	`Boolean` indicating if move was aborted
`REFRESH_BOARD`	Repaint the game board	none
`ENABLE_BOARD`	Enable the game board for user input	none
`REFRESH_STATUS`	Repaint the game status indicator(s)	none
`OPEN_HELP`	Open a help window	none
`CLOSE_HELP`	Close a help window	none
`NEW_GAME`	New game selected	none

The `PLAY_MOVE` and `DONE_PLAYING` events are handled by the *`AbstractGameWindow`* class and other events should be handled appropriately by the game being implemented, if relevant. Most of these events are posted by the common applet classes, but may be posted by the game being implemented as required.

Class Diagram

The UML sketch class diagram below shows the key classes that comprise the common applet classes.



Common Console Classes

The common console classes provide abstracted functionality for creating a console-based application that uses the strategy game framework.

Game

The *AbstractGame* abstract class is simply a container for wrapping all the game-related entities, such as a searcher and the competitors (which in turn may contain other game-related entities). Abstract methods are defined that concrete classes need to define appropriately for the game being implemented.

Competitors

The *AbstractCompetitor* abstract class wraps a *Player* object and adds abstracted operations for determining a move. Two concrete subclasses are defined. `HumanCompetitor` represents a user-controlled player in the game and determines the move to play via user input. `ComputerCompetitor` represents a computer-controlled player in the game and determines the move to play via a searcher in the strategy game framework.

Thinker

The `Thinker` class implements the *java.lang.Runnable* interface to allow the computer's search for a move to be done in a thread.

Statistics

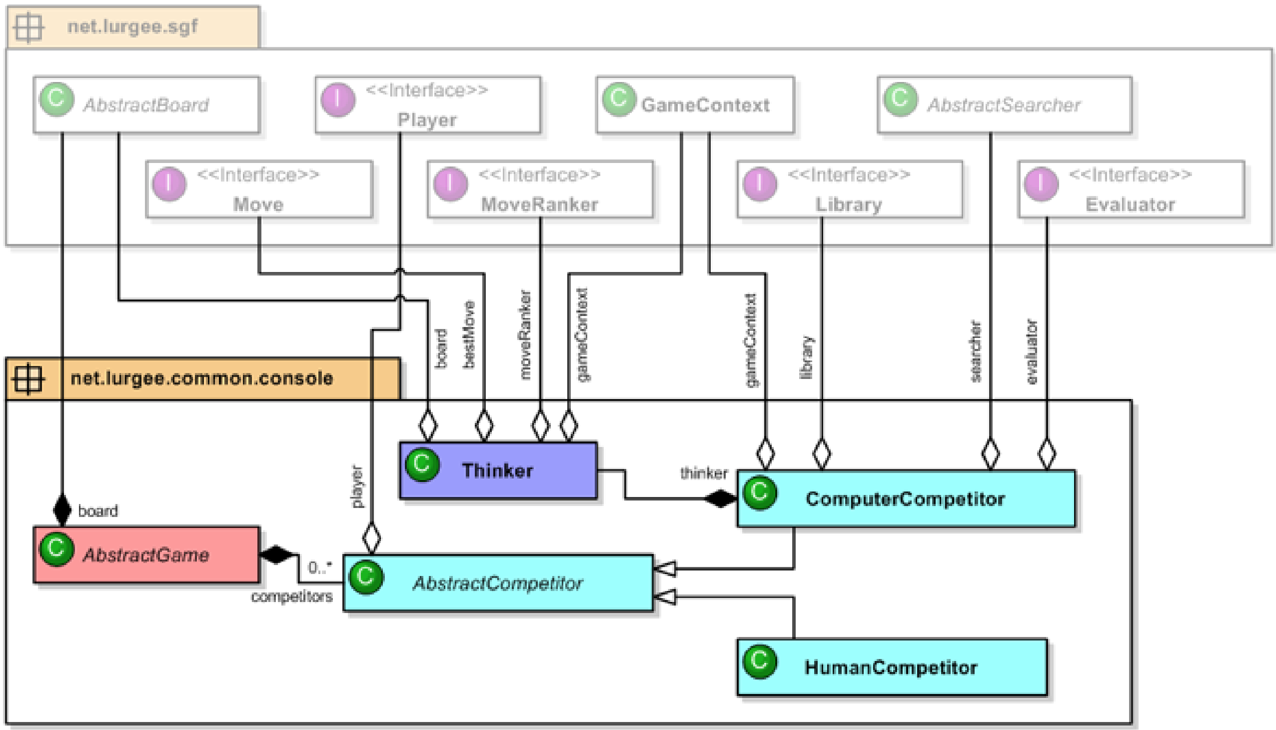
The `GameStats` class can be used to collate and report back on statistics regarding games played. It uses two related classes, `PlayerStats` to store the statistics for a single player and `ResultStats` to store the result of a single game. `PlayerStats` implements the sgf `SearchProgressListener` interface and obtains feedback regarding the search and evaluation process directly from the strategy game framework.

Input and Output

All input and output operations are done through the `Input` and `Output` classes. This allows common functionality to be changed easily and facilitates testing by replacing the input or output classes with test versions.

Class Diagram

The UML sketch class diagram below shows the key classes that comprise the common console classes.



Reversi

Rules of the Game

Reversi (also known as *Othello*) is a classic board game which is played by two players on an 8x8 grid with pieces that have two distinct sides (typically a dark side and a light side, or black and white).

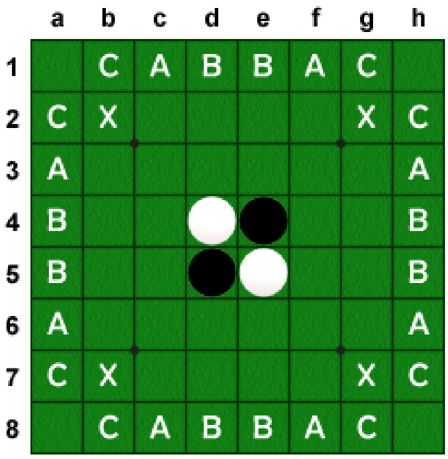
Each player places a piece on the board in turn, with their colour facing upwards, which designates it as their piece. Opponent pieces that are situated contiguously between the played piece and another of the player's pieces, orthogonally or diagonally, are flipped over (so they become the player's pieces).

Every move must result in at least one of the opponent's pieces being flipped. If a player has no valid moves, the other player plays again. The game ends when the board is full or when neither player can make a valid move. The winner is the player with the most pieces on the board. A game can be drawn if each player has an equal number of pieces on the board.

The game starts with two of each player's pieces in the centre four squares of the board, as shown below. Black always plays first.

Conventions

Squares on the board are referenced by a letter from `a` to `h`, and a number from `1` to `8` as shown in the board below. The starting state of the board is shown, with white pieces at `d4` and `e5`, and black pieces at `e4` and `d5`. Some squares are given special names: the squares diagonally in from each corner are known as *X-squares* (`b2`, `g2`, `b7` and `g7`); the edge squares adjacent to the corners are known as *C-squares* (`b1`, `a2`, `g1`, `h2`, `a7`, `b8`, `h7` and `g8`); the two edge squares in the middle of each edge are known as *B-squares* (`d1`, `e1`, `a4`, `a5`, `h4`, `h5`, `d8`, `e8`) and the edge squares between the B-squares and C-squares are known as *A-squares* (`c1`, `f1`, `a3`, `a6`, `h3`, `h6`, `c8`, `f8`).



Game Complexity

Reversi has a state-space complexity (the number of nodes in the game tree for a fully evaluated game) in the order of 10^{28}.

Strategy

Some of the common strategies used when playing reversi are discussed here. The rules of the game page published by the French Othello Federation is a good reference more detailed descriptions of the common strategies and for more advanced strategies.

Piece Differential

Although the aim of the game is to end up with more pieces on the board than the opponent, it is generally not advantageous to gain a majority too early in the game as this limits *mobility*.

Mobility

Mobility is a measure of the number of possible moves that a player has. When a player's mobility becomes low, they may be forced into making undesirable moves. It is therefore advantageous to play moves that improve mobility and/or decrease the opponent's mobility. Mobility is often the key factor in winning a game.

Frontier Pieces

Frontier pieces are those that are next to empty squares on the board. The fewer frontier pieces a player has, the lower the opponent's mobility becomes. A *quiet move* is a move that flips no frontier pieces and is often a good move. A corollary measure of frontier pieces is *potential mobility*, which is the number of squares that are potentially moves (open squares next to opponent frontier pieces). Improving potential mobility increases the chances of improving mobility.

Corners

Once a piece is placed in a corner it can never be flipped, so corners can be used to anchor other *stable pieces* on the edges.

Stable Pieces

Pieces that cannot be flanked by an opponent's piece cannot be flipped and are referred to as *stable*. Corners are always stable, as well as neighbouring pieces of the same colour. Establishing several stable pieces during the game guarantees an end score for the player of at least the number of stable pieces.

Edges

Playing pieces on edges can have mixed results. Playing edges too early in a game should be avoided. Generally, A-squares are stronger positions that B-squares.

C-Squares

Playing pieces on C-squares are often weak moves that open the player up to tactical traps which result in the opponent winning the corner. C-squares should thus be played with care.

X-Squares

Playing pieces on X-squares adjacent to open corners often result in the opponent winning the corner. X-squares should thus be played with caution.

Parity

As there are an even number of squares on the board and black always plays first, white always plays last, which gives white a slight advantage. More generally, the player making the last move in a particular closed-off region of the board has a slight advantage. Establishing parity involves leaving an even number of empty squares in each region in which the opponent can play.

Stages of the Game

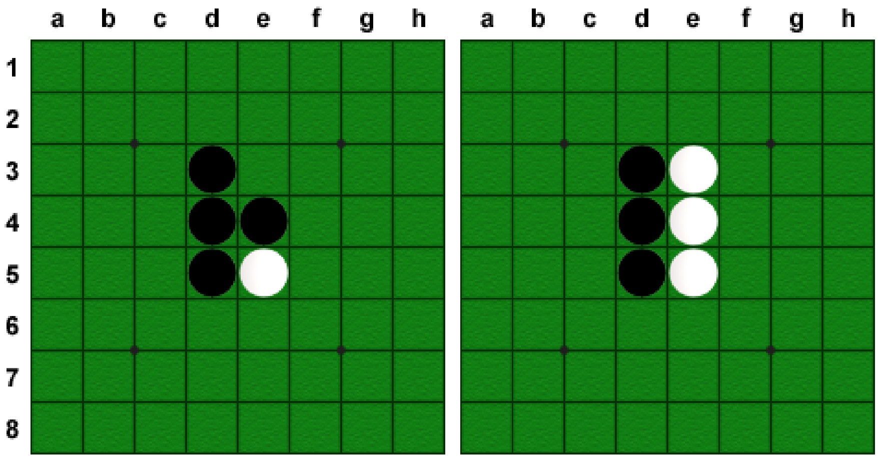
As the game progresses, a player's strategy needs to change in response to the opponent's moves and to the state of the board. A player's strategy through the early and middle stages of the game should lay the foundation for gaining a majority of pieces on the board in the endgame.

Opening Moves

There are numerous documented opening moves to a game of reversi - some covering as many as the first sixteen moves. The three possibilities for the first two moves are shown here. All of them are illustrated with `d3` as the first move; however the same openings apply for the other three possible first moves, `c4`, `f5` and `e6`, just transposed on the board.

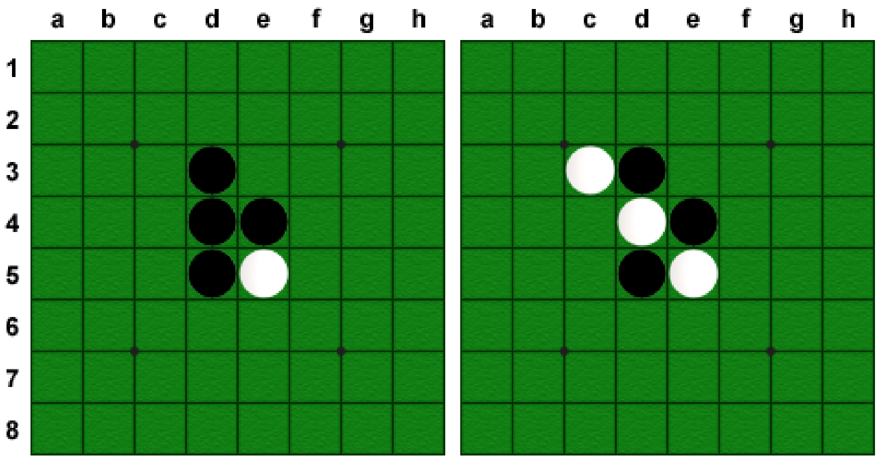
Parallel Opening

The parallel opening is `d3 e3` or `c4 c5` or `f5 f4` or `e6 d6`. It is generally considered to be the weakest opening by white.



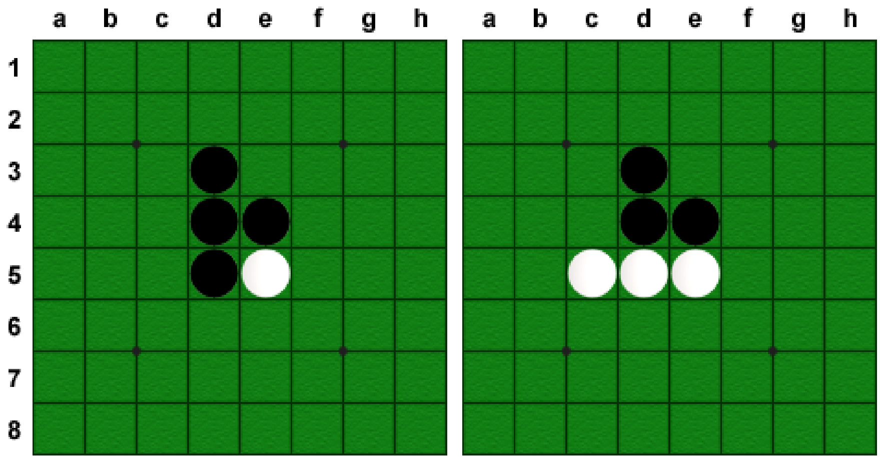
Diagonal Opening

The diagonal opening is `d3 c3` or `c4 c3` or `f5 f6` or `e6 f6`.



Perpendicular Opening

The perpendicular opening is `d3 c5` or `c4 e3` or `f5 d6` or `e6 f4`.



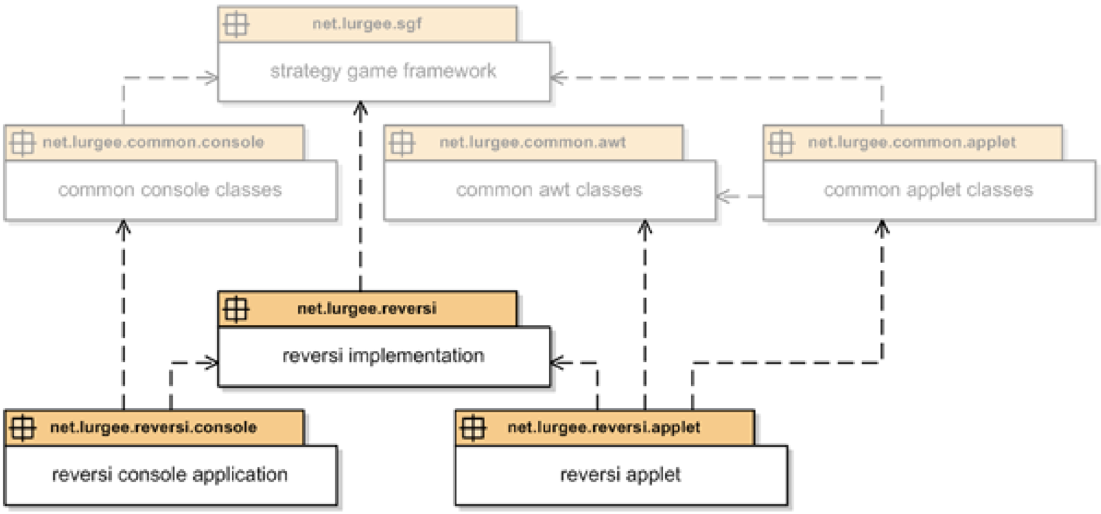
Reversi Code

The code may be found in the `reversi` directory in the source code. The structure of the `reversi` directory is as follows:

- `src` - Java source for the reversi classes, the console application and the applet.
- `test-src` - Unit tests for the reversi classes based on JUnit.
- `bin` - Target directory for classes when the code is built with Ant.
- `test-bin` - Target directory for unit tests when the code is built with Ant.
- `test-results` - Target directory for JUnit test results when the code is built with Ant.
- `doc` - Target directory for javadocs when the code is build with Ant.
- `res` - Image and audio resources used by the applet.
- `www` - Test web page for the applet.
- `dist` - Target directory for the applet jar when the code is built with Ant.

Package Diagram

The package diagram below shows the package dependencies for the reversi code.



Reversi Implementation

Features

The features of the reversi implementation include:

- At lower levels, game play is of a decent level, but can be beaten by novice to good players.
- At higher levels, game play is good enough to beat most players most of the time.
- Performance is good enough for the classes to be used on a mobile device.
- The computer does not always make the same moves, so game play does not feel 'mechanised'.

Improvements

Improvements that could be made include:

- The evaluation function could be made stronger; in particular, support for stable pieces and parity could be introduced.
- A more fully implemented opening-move library.

Constants

The `Colour` class defines constants for black and white, as well as constants to indicate 'any piece' (black or white) and 'no piece'.

The `Direction` class defines the eight directions in which a square on the board may be adjacent to another square on the board. These are the usual orthogonal directions: up, down, left, right; as well as the diagonal directions: up-left, up-right, down-left and down-right.

Board

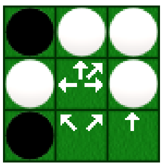
The `ReversiBoard` class extends the *`AbstractBoard`* class from the strategy game framework to provide the specifics of the board in reversi. Squares on the board are identified by x and y co-ordinates, which are one-based (so each has the range 1 to 8).

To allow entire rows on the board to be checked quickly and to facilitate bitwise-operations on boards, the 64 squares on the board are represented by a single 8-element array, each element of which represents a horizontal row of 8 squares. Each set of 2 bits, starting with the 2 least significant bits, represents an individual square in the row. Each square on the board may be empty, or may contain a black piece or a white piece. The values are:

- `Colour.NONE` (0) - The square is empty.
- `Colour.BLACK` (1) - The square contains a black piece.
- `Colour.WHITE` (2) - The square contains a white piece.

For example, if the value of an element in the array (which represents one of the rows on the board) is `0x4268` (`0100001001101000` in binary), the row is as follows:

To allow valid moves to be determined quickly, the empty squares that are adjacent to black or white pieces on the board are maintained in a 2x8 array. *Adjacent* means in a square next to that piece, orthogonally or diagonally. The first array is for the black pieces on the board and the second for the white. Each element in each of the two arrays represents a row on the board, with each group of eight bits representing a square in that row, starting with the 8 least significant bits (for the first column). Each set of 8 bits represents a bit mask containing the directions that the square is in with respect to the adjacent black or white piece using the 8 directions defined in the `Direction` class. For example, the adjacents for white are shown on the section of board below:



The values in the adjacent array for white for these three squares are:

- `Direction.RIGHT | Direction.LEFT | Direction.UP | Direction.UP_RIGHT`
- `Direction.UP_LEFT | Direction.UP_RIGHT`
- `Direction.UP`

The *`ReversiDifferenceBoard`* interface specifies only the methods that are applicable when two boards are compared with the `compare` method in `ReversiBoard`.

Player

The `ReversiPlayer` class implements the *`Player`* interface from the strategy game framework to provide the specifics of a player in reversi. Each player is identified by the colour that they are playing (black or white). This class includes static instances for each of the two players and a static accessor method to obtain references to them.

Move

The `ReversiMove` class implements the *`Move`* marker interface from the strategy game framework to provide the specifics of a move in reversi. As playing a move in reversi simply involves a player placing a piece on the board, the `ReversiMove` class represents a move by the board co-ordinates (x and y) that a piece is being placed on. Internally, this is stored as an instance of `ReversiPosition`. The colour is not associated to the move as a move is always associated to a player, so it is implicit. The `ReversiMove` class is immutable.

The `ReversiMoveFactory` class implements the *`MoveFactory`* interface from the strategy game framework to provide methods that return instances of `ReversiMove` given the x and y co-ordinates, or a string in the usual reversi notation (such as 'a8'). As the `ReversiMove` class is immutable and there are only 60 possible moves in reversi, immutable instances are created once by the factory and re-used. As thousands of moves are obtained and used during the search process, using immutable instances rather than instantiating moves whenever they are used has a substantial positive impact on performance.

Ranks for moves are determined by the `ReversiMoveRanker` class, which selects a rank for a particular move from a static table based on the position on the board. Moves that are likely to be good moves (without considering the state of the board, as it's a static table) are given higher ranks. Corners are given the highest rank, X-squares the lowest rank, and so on. As lists of moves are sorted according to rank, this approximation helps to increase the likelihood of alpha-beta cut-offs during a tree search.

Evaluator

The `ReversiEvaluator` class implements the *`Evaluator`* interface from the strategy game framework to provide the means for scoring the state of the board, which is used when searching a game tree to determine the best move. Evaluation is based on some of the common reversi strategies. The overall score is determined by summing the scores for each of the implemented strategies. Some strategies are applied only to pieces that have changed (placed on the board or flipped) from the original state of the board (the board at the foot of the game tree), some only to placed pieces, and some to all pieces on the board.

- **Piece differential** - a count of the number of pieces the player has gained minus the same count for the opponent; only pieces that have changed are considered.
- **Mobility** - a count of the number of valid moves the player has minus the same count for the opponent; all pieces on the board are considered.
- **Potential mobility** - a count of the number of potential moves (open squares that are adjacent to an opponent's piece) that the player has minus the same count for the opponent; all pieces on the board are considered.
- **Corners** - a count of the number of corners the player has gained minus the same count for the opponent; only pieces that have changed are considered.
- **Edges** - a count of the number of A-squares and B-squares the player has gained minus the same count for the opponent; only pieces that have changed are considered.
- **X-squares** - a count of the number of X-squares adjacent to open corners the player has gained minus the same count for the opponent; only placed pieces (not flipped pieces) are considered.
- **Wipe-out prevention** - a move that results in a wipe-out for the player (a player losing all their pieces) is given a score of -1000.
- **Stage of the game** - different weights are applied to each of the calculated scores for the various strategies. As indicated in the table below, these weights are changed depending on the stage of the game, which is represented by the number of pieces on the board. Some strategies are not relevant in certain stages of the game - the weight is set to 0 at these times.

<table>
<tr><th></th><th>Up to 16</th><th>17 to 32</th><th>33 to 48</th><th>49 to (64 - tree depth)</th><th>(64 - tree depth) or more</th></tr>
<tr><td>Piece differential</td><td colspan="3">0</td><td>4</td><td>1</td></tr>
<tr><td>Potential mobility</td><td>5</td><td>4</td><td>3</td><td>2</td><td>0</td></tr>
<tr><td>Mobility</td><td>7</td><td>6</td><td>5</td><td>4</td><td>0</td></tr>
<tr><td>Edges</td><td>0</td><td>3</td><td>4</td><td>5</td><td>0</td></tr>
<tr><td>Corners</td><td colspan="4">35</td><td>0</td></tr>
<tr><td>X-Squares</td><td colspan="4">-8</td><td>0</td></tr>
</table>

Library

The `ReversiLibrary` class implements the *`Library`* interface from the strategy game framework to provide a simple move library for selecting the second move of the game only. One of the opening moves is randomly selected, weighted by the effectiveness of the above evaluation scheme with each opening, as follows:

If the tree depth is 3 or less (which implies the computer level is low):

> The **parallel opening** is selected 11% of the time,
>
> the **diagonal opening** is selected 33% of the time,
>
> the **perpendicular opening** is selected 56% of the time.

If the tree depth is greater than 3:

> The **diagonal opening** is selected 33% of the time,
>
> the **perpendicular opening** is selected 67% of the time.

Reversi Console Application

The reversi console application uses the classes described above, as well as the common console classes and the strategy game framework. This is a full implementation of the game of reversi, in which each player can be played by a user or by the computer at varying skill levels. The console application looks something like this:

```
White (X) plays h1
7375ms; Score: Black (O): 50, White (X): 14
  a b c d e f g h
1 O O O O O O O X
2 O O O O O O X O
3 O O O X X X O O
4 O X X O X O O O
5 O X O X O X O O
6 O X O O X O O O
7 O O O O O X O O
8 O O O O O O O O
Black (O) wins

STATS                   Black (O)             White (X)
Total wins              4 (80%)               1 (20%)
Wins when starting      3 (60%)               1 (20%)
Early wins              1                     0
Average score           46.2                  14.0
Parallel openings       0                     0
Diagonal openings       2                     0
Perpendicular openings  2                     5
Moves considered        334772                45913
Moves discarded         0                     31872
Moves played            144                   137
Evaluations done        251014                32029
Evaluations discarded   0                     22275
Search depths           1:2 2:2 3:2 4:2 5:134 1:2 2:2 3:44 4:53 5:33
Searcher                Iterative Negascout   Iterative Negamax
Evaluator               Standard              Alternative
Tree depth              5                     5
Games played            5
Time taken              7375 ms
Scores                  50-14 45-0 56-8 30-34 50-14
```

Purpose

The user interface of the reversi console application is less than elegant and is not conducive to good game play. Its primary purpose is to assess the reversi implementation. Playing the computer against itself is a particularly good way of assessing the effectiveness of various strategies. A second reversi evaluator class, `AlternativeEvaluator`, is included in the application to allow strategies to be changed and tried against the default evaluator in the reversi implementation. Assessing strategies should involve many games being played (at least 20, but more is better) to even out the results. Various tree depths (usually 5 or more) should be used for both players, unless the strategy being assessed is specifically to improve shallow searches. Also, various combinations of each evaluator being used for the starting player and the second player should be assessed to ensure that it is not too effective for only the one (for example, this would be important for parity).

Alternative Evaluator

The `AlternativeEvaluator` class implements the *`Evaluator`* interface from the strategy game framework and is a copy of the `ReversiEvaluator` class from the reversi implementation. It is included in the console application to allow the effectiveness of various strategies to be assessed, as described above in the purpose section.

For example, changing the weights for all the strategies to 0 except for piece differentials makes the evaluator operate on the *greed* principle, which is it always plays the move that results in the most pieces being won. This is a poor strategy and if played against the default evaluator in the reversi implementation, it should lose most of the time with a tree depth of 1 for both competitors, and should always lose with a higher tree depth.

Statistics

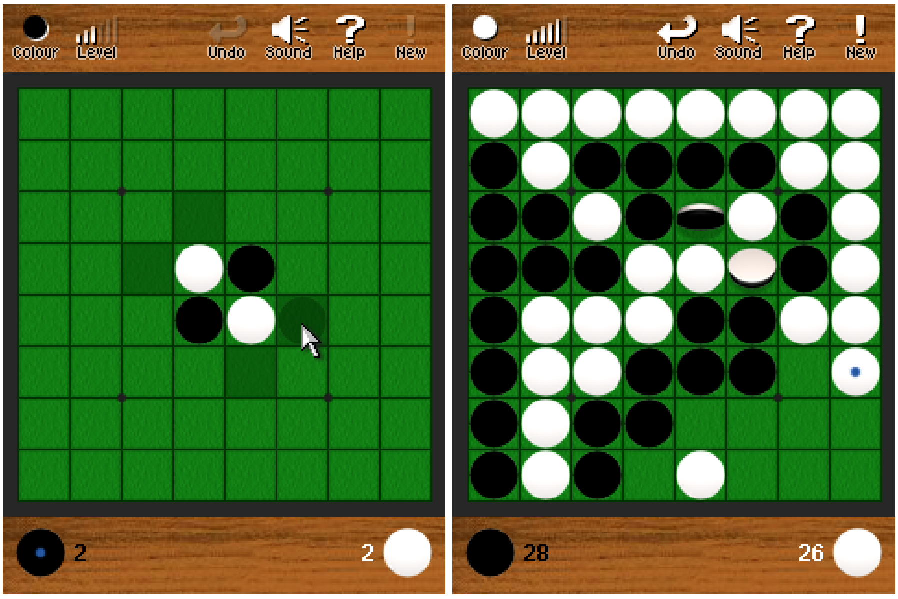
The `ReversiGameStats`, `ReversiPlayerStats` and `ReversiResultStats` classes extend the common `GameStats`, `PlayerStats` and `ResultStats` classes to add additional stats specific to reversi.

Game

The `ReversiGame` class extends the common console *`AbstractGame`* class and contains the static `main` method, which is the entry point for the application. The main method creates a singleton instance of the `ReversiGame` class and calls its `run` method, which initiates the application.

Reversi Applet

The reversi applet uses the classes described above, as well as the common applet classes, the common AWT classes and the strategy game framework. Below are two screenshots of the applet in action:



Features

The features of the reversi applet include:

- Colourful graphical user interface.
- Simple layout, with all options available directly on the main screen (rather than popup menus).
- Graphical indication of the valid moves on a player's turn.
- Six difficulty levels ranging from easy to difficult (can be changed during a game).
- User can play black or white (can be changed during a game).
- Unlimited undos.
- Animated moves.
- Audio (which can be disabled).

Game Play

The reversi applet uses iterative deepening with a negascout searcher. The difficulty levels 1 to 6 map to tree depths and evaluation thresholds as follows:

Level	Maximum Search Depth	Evaluation Threshold
1	5	250
2	7	1000
3	8	2500
4	9	7500
5	10	25000
6	16	50000

GUI Design

The applet is fairly small (240x320 pixels). This size was specifically selected as it is a common size for PDA devices and a PDA version of reversi is planned. So, rather than designing the user interface twice, this same interface will be used for the PDA version.

Constants

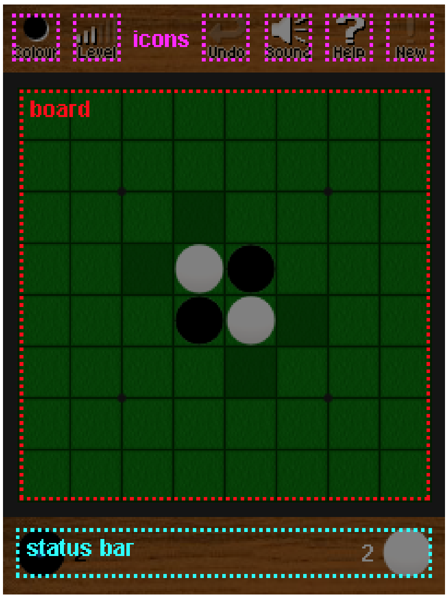
The `AppletConsts` class contains common constants that are used throughout the applet.

Applet

The `ReversiApplet` class extends the common applet *`AbstractGameApplet`* class and creates the appropriate main window for the game.

Main Window

The `ReversiMainWindow` class extends the common applet *`AbstractGameWindow`* class and contains (and creates) several widgets, as shown below, each of which is responsible for painting an area of the window and managing mouse events generated within them. The `ReversiMainWindow` class overrides the `paint` method to appropriately paint the display. The widgets contained in the window paint themselves.



The *undo*, *help* and *new game* icons are not stateful, whilst the *colour*, *level* and *sound* are. Only the *undo* and *new game* icons are ever disabled during the game, so only these two are provided with disabled images on construction.

The main window also contains two other windows: the help window and the new game option window. Both are created hidden and only made visible when they are required. The help window is an instance of the common AWT `TextWindow` class and the new game option window is an anonymous inner class that extends the common AWT *`OptionWindow`* class.

Board

The *`ReversiBoardWidget`* class extends the common applet *`AbstractBoardWidget`* class and overrides the `paint` method to paint the reversi board and the pieces on it. It implements the common applet *`WidgetEventListener`* and *`Animatable`* interfaces and is registered as a listener for widget events. This widget deals with mouse events relating to the user moving the mouse over the board, and clicking on a square (making a move).

Status Bar

The `ReverisStatusWidget` class extends the common AWT *`Widget`* class and overrides the `paint` method to paint the status bar, which contains a black piece and a white piece with an indication of whose turn it is; the score for each player; and a status message (which is often blank). It implements the common applet *`WidgetEventListener`* interface and is registered as a listener for widget events.

Game

The `ReversiGame` class extends the common applet *`AbstractGame`* class and provides implementations of the abstract methods in this class appropriate to the game of reversi.

Connect-Four

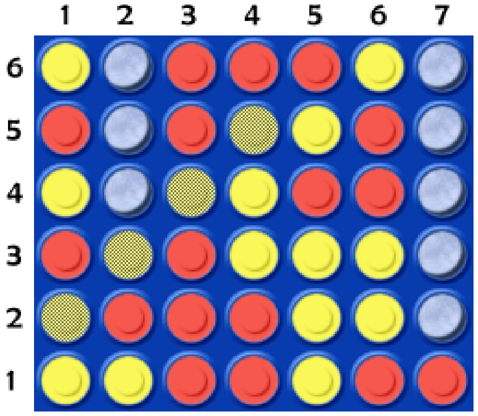
Rules of the Game

Connect-four (also known as *Four-in-a-row*) is a board game which is played by two players on an 7x6 vertical grid with pieces that are dropped into the top of the grid to slide down to the lowest available position in that column. The objective is to get four of one's own pieces in a line horizontally, vertically or diagonally.

Each player drops a piece into the grid in turn. The game ends when one of the players achieves a line of four (or more) pieces, or when the grid is full. If neither player has managed to achieve a line of four then the game is drawn. The game starts with an empty grid and red plays first.

Conventions

Squares on the grid are referenced by a number from `1` to `7` representing the column and a number from `1` to `6` representing the row, as shown in the board below, which has yellow winning the game with a diagonal line `(1,2)`, `(2,3)`, `(3,4)` and `(4,5)`.



Game Complexity

Connect-four has a state-space complexity (the number of nodes in the game tree for a fully evaluated game) in the order of 10^{14}.

Strategy

Some of the common strategies used when playing connect-four are discussed here.

Groups

A *group* (or *line*) is a collection of two or more pieces in an unbroken line horizontally, vertically or diagonally. A group of four represents a winning situation. Simply attempting to create a group of four early in a game will be blocked by an opponent, so usually several lines need to be built up simultaneously.

Possible Winning Groups

On an empty board, there are 69 possible ways to create a group of four. As the game progresses, these possibilities decrease as the opponent places pieces within these groups. The greater the number of groups that have not been broken by an opponent piece, the higher the chance of winning. Additionally, the more pieces that a player has in these unbroken groups, the more useful these groups become. The diagram below shows all the winning groups, numbered 1 to 69.

	1	2	3	4	5	6	7
6	21 27 61	21 22 30 64	21 22 23 33 67	21 22 23 24 36 46 69	22 23 24 39 48	23 24 42 51	24 45 54
5	17 26 27 59	17 18 29 30 61 62	17 18 19 32 33 46 64 65	17 18 19 20 35 36 47 48 67 68	18 19 20 38 39 50 51 69	19 20 41 42 53 54	20 44 45 56
4	13 25 26 27 58	13 14 28 29 30 46 59 60	13 14 15 31 32 33 47 48 61 62 63	13 14 15 16 34 35 36 49 50 51 64 65 66	14 15 16 37 38 39 52 53 54 67 68	15 16 40 41 42 55 56 69	16 43 44 45 57
3	09 25 26 27 46	09 10 28 29 30 47 48 58	09 10 11 31 32 33 49 50 51 59 60	09 10 11 12 34 35 36 52 53 54 61 62 63	10 11 12 37 38 39 55 56 64 65 66	11 12 40 41 42 57 67 68	12 43 44 45 69
2	05 25 26 47	05 06 28 29 49 50	05 06 07 31 32 52 53 58	05 06 07 08 34 35 55 56 59 60	06 07 08 37 38 57 62 63	07 08 40 41 65 66	08 43 44 68
1	01 25 49	01 02 28 52	01 02 03 31 55	01 02 03 04 34 57 58	02 03 04 37 60	03 04 40 63	04 43 66

→ Horizontal groups
↑ Vertical groups
↖ Diagonal groups
↗ Diagonal groups

It is evident from the above diagram that some squares on the board participate in more possible winning groups than others. As a result, these squares tend to be slightly more valuable. The diagram below shows the distribution of the number of winning groups that each square participates in.

	1	2	3	4	5	6	7
6	3	4	5	7	5	4	3
5	4	6	8	10	8	6	4
4	5	8	11	13	11	8	5
3	5	8	11	13	11	8	5
2	4	6	8	10	8	6	4
1	3	4	5	7	5	4	3

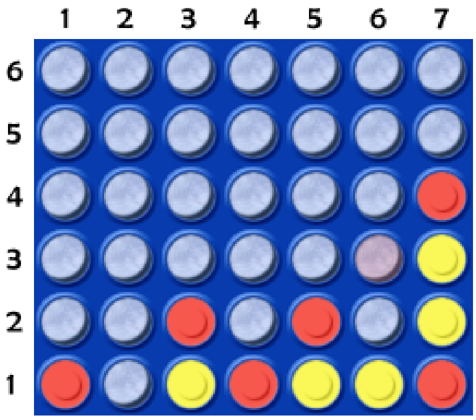
Threats

A *threat* exists when three of the pieces required for creating a group four are in place and the fourth square is open.

An *immediate threat* (or *atari*) is a threat in which the fourth square does not have an open square below it; which means that unless the opponent can win on the next move, they must play in that column. Creating an immediate threat is a way of forcing the opponent to make a particular move. Creating a double immediate threat results in victory if the opponent cannot create a group of four themselves.

An *active threat* is a threat in which the fourth square has an open square below it, but not below that; which means that if the opponent plays in this column the player will be able to complete the group. Active threats are therefore useful in limiting where the opponent can play.

Threats are called *odd* or *even*, depending on the row on which they occur. For example, in the board below, red has created an odd active threat in column 6.



A threat is called *shared* if the opponent has an equivalent threat in the same column. So, a *shared odd* threat requires an odd threat for each player in the same column and a *shared even* threat requires an even threat for each player in the same column.

As red plays first, after every move by red an odd number of open squares remain on the board and after every move by yellow an even number of open squares remain on the board. Given this, if the board is completely full except for a single line, red will win if there is an odd red threat in the column and no even yellow threats, or only even yellow threats above the red threat. Conversely yellow will win if there is an even yellow threat in the column and no odd red threats, or only odd red threats above the yellow threat. Logically, one would think then that the strategy for red is to create odd threats and for yellow to create even threats. This situation however only holds for a single column and is more complicated when the whole board is considered. Generally, the rules below may be applied.

For red to win, red needs:

- more odd threats (shared or unshared) than yellow

For yellow to win, yellow needs:

- more odd threats than red if one or more threats are shared
- an even number of odd threats which is the same number of odd threats than red, if one or more threats are shared
- the same number of odd threats (which can be zero) as red plus at least one even threat

Opening Moves

It can be shown that with perfect play, red can force a win by starting in the middle column (`4`). Yellow can force a win by starting in one of the four outer columns (`1`, `2`, `6` or `7`), or can obtain a draw starting with one of the columns adjacent to the middle column (`3` or `5`).

Connect-Four Code

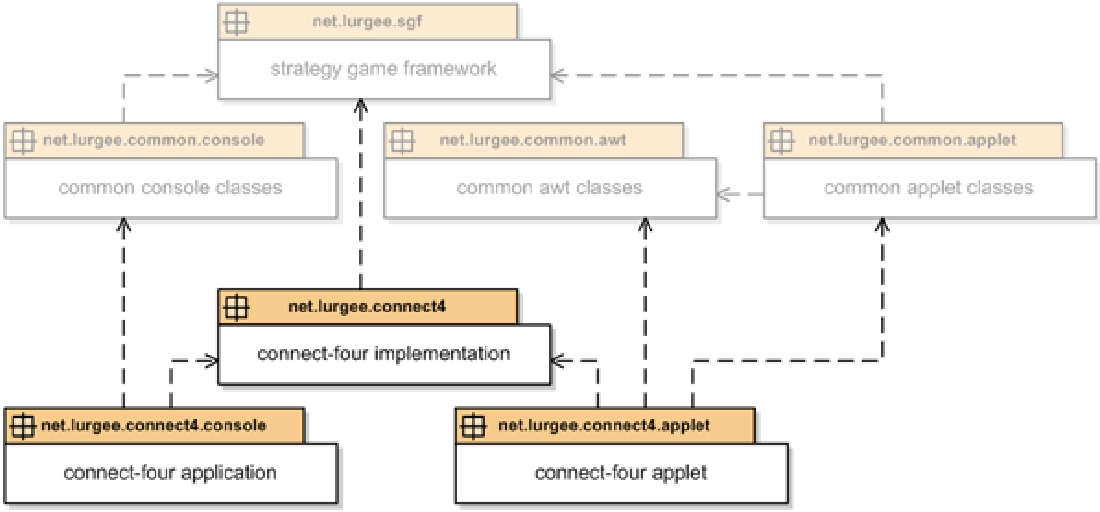
The code may be found in the `connect4` directory in the source code. The structure of the `connect4` directory is as follows:

- `src` - Java source for the connect-four classes, the console application and the applet.
- `test-src` - Unit tests for the connect-four classes based on JUnit.
- `bin` - Target directory for classes when the code is built with Ant.
- `test-bin` - Target directory for unit tests when the code is built with Ant.
- `test-results` - Target directory for JUnit test results when the code is built with Ant.
- `doc` - Target directory for javadocs when the code is build with Ant.

- res - Image and audio resources used by the applet.
- www - Test web page for the applet.
- dist - Target directory for the applet jar when the code is built with Ant.

Package Diagram

The package diagram below shows the package dependencies for the connect-four code.



Connect-Four Implementation

Features

The features of the connect-four implementation include:

- At the lowest level, game play is of a decent level, but can be beaten by novice to good players.
- At higher levels, game play is good enough to all (human) opponents.
- Performance is good enough for the classes to be used on a mobile device.
- The computer does not always make the same moves, so game play does not feel 'mechanised'.

Constants

The Colour class defines constants for red and yellow, as well as constants to indicate 'any piece' (red or yellow) and 'no piece'.

Board

The Connect4Board class extends the *AbstractBoard* class from the strategy game framework to provide the specifics of the board in connect-four. Squares on the board are identified by x and y co-ordinates, which are one-based (so, 1 to 7 horizontally and 1 to 6 vertically).

To allow entire rows on the board to be checked quickly and to facilitate bitwise-operations on boards, the 42 squares on the board are represented by a single 6-element array, each element of which represents a horizontal row of 7 squares. Each set of 2 bits, starting with the 2 least significant bits, represent an individual square in the row. Each square on the board may be empty, or may contain a red piece or a yellow piece. The values are:

- Colour.NONE (0) - The square is empty.
- Colour.RED (1) - The square contains a red piece.
- Colour.YELLOW (2) - The square contains a yellow piece.

For example, if the value of an element in the array (which represents one of the rows on the board) is 0x0A51 (00101001010001 in binary), the row is as follows:

To allow possible winning groups to be examined quickly, a static map of squares to winning groups, numbered 1 to 69 (in the same way as in the possible winning groups diagram) is held, as well as the reverse (winning group to square), which is generated initially. A count of the number of squares that a player has filled in each possible winning group, as well as whether or not the group has been broken by an opponent piece, is maintained to assist the evaluation process.

Player

The Connect4Player class implements the *Player* interface from the strategy game framework to provide the specifics of a player in connect-four. Each player is identified by the colour that they are playing (red or yellow). This class includes static instances for each of the two players and a static accessor method to obtain references to them.

Move

The `Connect4Move` class implements the *`Move`* marker interface from the strategy game framework to provide the specifics of a move in connect-four. As playing a move in connect-four simply involves a player selecting a column to drop a piece into, the `Connect4Move` class represents a move by the column on the board (the x co-ordinate). The colour is not associated to the move as a move is always associated to a player, so it is implicit. The `Connect4Move` class is immutable.

The `Connect4MoveFactory` class implements the *`MoveFactory`* interface from the strategy game framework to provide methods that return instances of `Connect4Move` given the x co-ordinate. As the `Connect4Move` class is immutable and there are only 7 possible moves in connect-four, immutable instances are created once by the factory and re-used. As thousands of moves are obtained and used during the search process, using immutable instances rather than instantiating moves whenever they are used has a substantial positive impact on performance.

Ranks for moves are determined by the `Connect4MoveRanker` class, which selects a rank for a particular move from a static table based on the position on the board. Moves that are likely to be good moves (without considering the state of the board, as it's a static table) are given higher ranks. The ranking table is therefore based on the diagram showing the distribution of possible winning groups to squares. As lists of moves are sorted according to rank, this approximation helps to increase the likelihood of alpha-beta cut-offs during a tree search.

Evaluator

The `Connect4Evaluator` class implements the *`Evaluator`* interface from the strategy game framework to provide the means for scoring the state of the board, which is used when searching a game tree to determine the best move. Evaluation is based purely on maximising the number of pieces a player has in unbroken possible winning groups, whilst minimising the same count for the opponent. Different weights are given to unbroken possible winning groups depending on the number of the player's pieces in the group, as follows:

- If any group has all **four** squares filled with the player's pieces, the board is given a constant score of 5000.
- Otherwise, the score is `(4 * 2 ^ n3) + (2 * 2 ^ n2) + (2 * n1)`, where `n3` is a count of the number of unbroken possible winning groups that have **three** squares filled with the player's pieces (and therefore one open), `n2` is a count of the number of unbroken possible winning groups that have **two** squares filled with the player's pieces (and therefore two open) and `n1` is a count of the number of unbroken possible winning groups that have **one** square filled with the player's pieces (and therefore three open).

This strategy is based on the premise that unbroken groups containing increasing numbers of the player's pieces become exponentially more valuable. The above calculation is done for the player and for the opponent and the final score for the board is the player's score minus the opponent's score.

Library

The `Connect4Library` class implements the *`Library`* interface from the strategy game framework to provide a simple move library for selecting the first and second moves of the game only, based on some simple rules for opening moves - it is best for the starting player to play in the middle column (`4`) and for the second player to play in one of the four outer columns (`1`, `2`, `6` or `7`).

For the starting player:

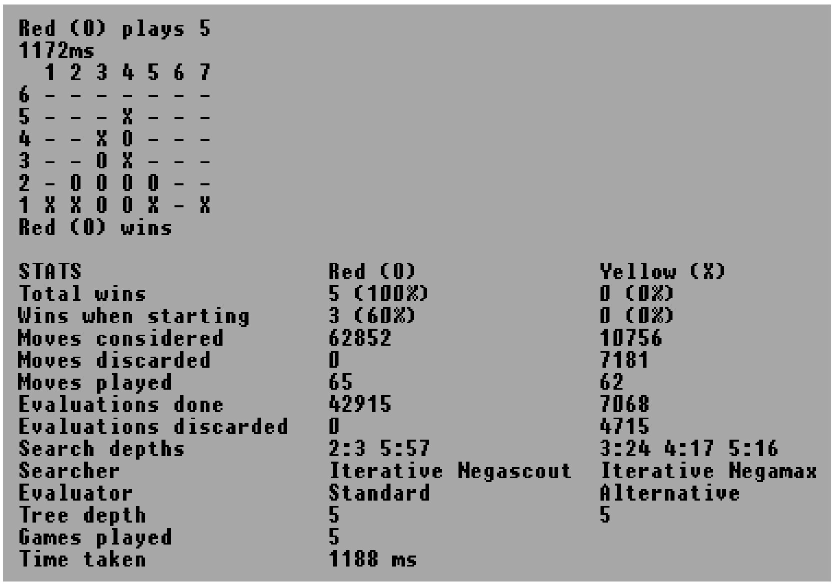
- Always play in column `4`.

For the second player:

- Randomly select one of the four outer columns, if there is already a piece in that column (which means the starting player played there), select another.

Connect-Four Console Application

The connect-four console application uses the classes described above, as well as the common console classes and the strategy game framework. This is a full implementation of the game of connect-four, in which each player can be played by a user or by the computer at varying skill levels. The console application looks something like this:



```
Red (O) plays 5
1172ms
  1 2 3 4 5 6 7
6 - - - - - - -
5 - - - X - - -
4 - - X O - - -
3 - - O X - - -
2 - O O O O - -
1 X X O O X - X
Red (O) wins

STATS                   Red (O)            Yellow (X)
Total wins              5 (100%)           0 (0%)
Wins when starting      3 (60%)            0 (0%)
Moves considered        62852              10756
Moves discarded         0                  7181
Moves played            65                 62
Evaluations done        42915              7068
Evaluations discarded   0                  4715
Search depths           2:3 5:57           3:24 4:17 5:16
Searcher                Iterative Negascout Iterative Negamax
Evaluator               Standard           Alternative
Tree depth              5                  5
Games played            5
Time taken              1188 ms
```

Purpose

The user interface of the connect-four console application is less than elegant and is not conducive to good game play. Its primary purpose is to assess the connect-four implementation. Playing the computer against itself is a particularly good way of assessing the effectiveness of various strategies. A second connect-four evaluator class, `AlternativeEvaluator`, is included in the application to allow strategies to be changed and tried against the default evaluator in the connect-four implementation.

Alternative Evaluator

The `AlternativeEvaluator` class implements the *`Evaluator`* interface from the strategy game framework and is a copy of the `Connect4Evaluator` class from the connect-four implementation. It is included in the console application to allow the effectiveness of various strategies to be assessed, as described above in the purpose section.

Statistics

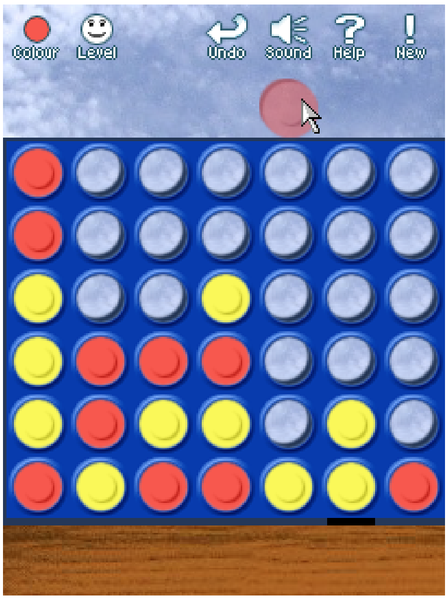
The common `GameStats`, `PlayerStats` and `ResultStats` classes are used to store statistics for the games played.

Game

The `Connect4Game` class extends the common console *`AbstractGame`* class and contains the static `main` method, which is the entry point for the application. The main method creates a singleton instance of the `Connect4Game` class and calls its `run` method, which initiates the application.

Connect-Four Applet

The connect-four applet uses the classes described above, as well as the common applet classes, the common AWT classes and the strategy game framework. Below is a screenshot of the applet in action:



Features

The features of the connect-four applet include:

- Colourful graphical user interface.
- Simple layout, with all options available directly on the main screen (rather than popup menus).
- Four difficulty levels ranging from easy to difficult (can be changed during a game).
- User can play red or yellow (can be changed during a game).
- Unlimited undos.
- Animated moves.
- Audio (which can be disabled).
- Visual indication of winning group of pieces.

Game Play

The connect-four applet uses iterative deepening with a negascout searcher. The difficulty levels 1 to 4 (smiley face to worried face on the GUI) map to tree depths and evaluation thresholds as follows:

Level	Maximum Search Depth	Evaluation Threshold
1	4	250
2	10	5000
3	12	50000
4	16	100000

GUI Design

The applet is fairly small (240x320 pixels). This size was specifically selected as it is a common size for PDA devices and a PDA version of connect-four is planned. So, rather than designing the user interface twice, this same interface will be used for the PDA version.

Constants

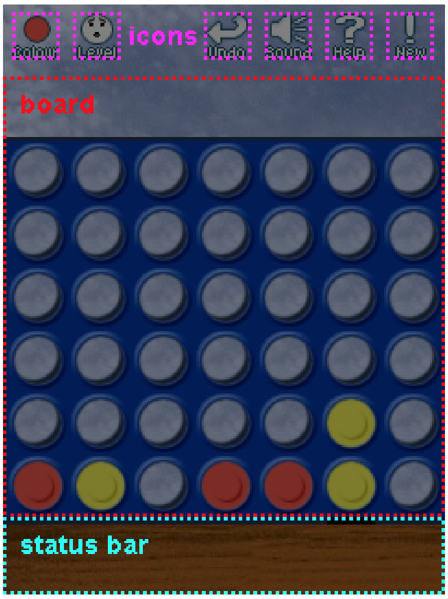
The `AppletConsts` class contains common constants that are used throughout the applet.

Applet

The `Connect4Applet` class extends the common applet *`AbstractGameApplet`* class and creates the appropriate main window for the game.

Main Window

The `Connect4MainWindow` class extends the common applet *`AbstractGameWindow`* class and contains (and creates) several widgets, as shown below, each of which is responsible for painting an area of the window and managing mouse events generated within them. The `Connect4MainWindow` class overrides the `paint` method to appropriately paint the display. The widgets contained in the window paint themselves.



The *undo*, *help* and *new game* icons are not stateful, whilst the *colour*, *level* and *sound* are. Only the *undo* and *new game* icons are ever disabled during the game, so only these two are provided with disabled images on construction.

The main window also contains two other windows: the help window and the new game option window. Both are created hidden and only made visible when they are required. The help window is an instance of the common AWT `TextWindow` class and the new game option window is an anonymous inner class that extends the common AWT *`OptionWindow`* class.

Board

The *Connect4BoardWidget* class extends the common applet *AbstractBoardWidget* class and overrides the `paint` method to paint the connect-four board and the pieces on it. It implements the common applet *WidgetEventListener* and *Animatable* interfaces and is registered as a listener for widget events. This widget deals with mouse events relating to the user moving the mouse over the board, and clicking on a column (making a move).

Status Bar

The `Connect4StatusWidget` class extends the common AWT *Widget* class and overrides the `paint` method to paint the status bar, which contains an indication of the last move played and a status message (which is often blank). It implements the common applet *WidgetEventListener* interface and is registered as a listener for widget events.

Game

The `Connect4Game` class extends the common applet *AbstractGame* class and provides implementations of the abstract methods in this class appropriate to the game of connect-four.

Source Code

The source code is available on github at http://github.com/mpatric/lurgee

For build instructions refer to the FAQ later in this document.

Release History

Strategy game source code archive 2.1

Released: 21 December 2007

Contents:

Strategy game framework

Common AWT, Applet and Console classes

Reversi console application and applet

Connect-four console application and applet

Release notes:

This release introduces iterative deepening searches and the killer heuristic for move ranking and associated changes to accommodate these. Details follow.

- The code is now Java 5 compliant.
- Renamed some classes to correctly reflect their function.
- Introduced the `Position` marker interface and changed the `Move` interface to just be a marker interface.
- Introduced the `GameContext` class as a single point for maintaining the game context, including required factories for generating game entities.
- Ranking of moves now does not need to be done explicitly by specialisations of the `AbstractBoard` class, but is handled by the `MoveList` class provided by the `GameContext`. This also handles 'randomness' in the selection of equally-good moves, rather than relying on the move rankers to do this artificially through the addition of a small random amount.
- Added some standardised tests for assessing relative performance.
- Improved the reversi evaluator.
- Completely changed the connect-four evaluator, with associated changes to `Connect4Board`. Now using a more quantitative approach to scoring the board rather than relying on some rather shaky rules.
- Added the `IterativeSearcher` and `KillerHeuristicMoveRanker` classes, which when used together allow for tree searches with fewer evaluations. Searches with `IterativeSearcher` can be ended after a specified number of evaluations rather than searching to a fixed depth.
- Reversi and connect-four applet now use negascout with iterative deepening rather than negascout with a fixed depth.
- Multiple listeners can be added to the searcher classes rather than just one.
- Applets log some interesting information to the Java console.
- Code cleanup and refactoring.

Known issues:

Audio does not work in Opera - this may be an Opera issue.

Strategy game source code archive 2.0

Released: 29 July 2007

Release notes:

This release is the result of a major refactor. Details follow.

- Added connect-four implementation, with console application and applet.
- Some changes to the strategy game framework to address issues shown up by the connect-four implementation and to tighten up the domain model - in particular, the board now has knowledge of the player to go, so the player is not passed around as an argument quite so much.
- Added further tests across the code base to tighten up regression tests.
- Created two new packages: `net.lurgee.common.console` and `net.lurgee.common.applet`, which contain generic abstracted functionality (originally in the reversi console application and applet) for implementing console applications and applets using the strategy game framework.
- Updated reversi applet GUI to add indicator of last piece played and to use new event-driven approach built into the common applet classes.

Strategy game source code archive 1.2

Released: 25 February 2007

Release notes:

This release includes some bugfixes, added functionality and some refinements. Details follow.

Strategy game framework:

- Added `MoveRanker` to delegate ranking of moves to.
- Added `NegascoutSearcher` to do negascout tree searches.
- Refactored `NegamaxSearcher` to allow `NegascoutSearcher` to extend it.

Reversi common module:

- Removed `ReversiConsts` class and created `Colour` and `Direction` classes.
- Added `ReversiMoveRanker` for ranking moves - this was previously done directly in `ReversiBoard`.
- Fixed a fairly serious bug in `compare` method in `ReversiBoard`, where counts were not being reset, which was affecting evaluation of board positions.
- Fixed a bug in `ReversiEvaluator`, which was using the wrong count for short-cutting some evaluations.
- Added an `equals` method to `ReversiBoard`, needed by unit tests.
- Added real mobility differential to `ReversiEvaluator`.
- Tweaked factors in `ReversiEvaluator` to make improve game play.

Reversi console application:

- Added option to select negamax or negascout searcher for each player.
- Updated `AlternativeEvaluator` with same changes as `ReversiEvaluator.`
- Refactored and tidied up `Reversi` class.

Reversi applet:

- Changed search depths for levels 1 - 6 from (2, 4, 5, 6, 7, 8) to (2, 3, 4, 5, 6, 7).
- Changed searcher to negascout rather than negamax.

Strategy game source code archive 1.1

Released: 15 April 2006

Release notes:

Strategy game framework:

- Changed the way in which no-move situations are managed - there is now a `byeAllowed` flag on the searcher, which indicates that if a user cannot move play reverts to the other player.

Strategy game source code archive 1.0

Released: 09 April 2006

Release notes:

Initial public release.

FAQ

Is the original reversi midlet available?

No, it is not currently available and is not likely to ever see the light of day again.

Will you mavenize the project?

No, but feel free to fork the project in github and do it, then submit a pull request.

How do I build the code?

To build the code you require:

- J2SE 1.5.0 or higher - the Java 2 platform, standard edition.
- Ant 1.6 or higher - the de facto build tool for Java projects.

To run the unit tests you require:

- JUnit 3.8.1 or higher - a popular, simple framework to write repeatable tests.

To optionally shrink/optimise/obfuscate the compiled code (which is particularly useful for applets and midlets) you require:

- ProGuard 3.0 or higher - an excellent Java shrinker/optimiser/obfuscator which operates on the byte code (not the source code).

Unpack the source code into a directory then edit `ant.properties` in the root directory making sure that the `junit-jar` and `proguard-jar` properties point to the correct locations for the JUnit and ProGuard jars.

Each project directory has its own build file and can be built separately, or the root build file can be used to build the lot. To build the complete source, execute:

```
ant build
```

or just simply:

```
ant
```

from the root of the directory where the source was unpacked, which calls the build target for each project in the correct order.

The useful Ant targets defined in the root build file are:

- `clean`: removes all the directories created by other targets.
- `build`: compiles the code in each subdirectory in the correct order.
- `doc`: generates javadocs for the complete source excluding the unit tests.
- `alldoc`: generates javadocs for the complete source including the unit tests.

The useful Ant targets defined in each of the project build files are:

- `clean`: removes all the directories created by other targets.
- `build`: builds the console app and the applet.
- `build-console-app`: compiles the code for the console application.
- `run-console-app`: launches the console application from Ant.
- `build-applet`: compiles the code and builds a distribution jar file for the applet.
- `doc`: generates javadocs for the project excluding the unit tests.
- `alldoc`: generates javadocs for the project including the unit tests.
- `test`: runs the unit tests for the project.

The reversi and connect4 projects include an obfuscate target, which slims down the applet distribution, minimising the size of the binary:

- `obfuscate`: shrink and obfuscate the distribution