Crossroads
The ACM Student Magazine

The Development of a Game Playing Framework Using Interface-based Programming

by ***Mark A. Cohen***

Introduction

The Java programming language contains object-oriented features enablingthe construction of interface-based application frameworks. Interfacesseparate module implementation from core implementation, thussimplifying module development. The following article demonstrates howto take advantage of Java interfaces by designing and implementing agame playing application framework.

Discovering the Interfaces

Gamma et al. suggest developers "Program to an interface, not animplementation" [2]. When using interfaces, theactual implementation is not a factor in the design. As a result, thefinished application has a much better chance of surviving theimplementation changes it will inevitably be asked to support.

An interface can be defined as a contract that outlines the terms of usefor a class of objects. When a class implements an interface, the classis agreeing to the contract outlined by the interface. In other words,the class is agreeing to provide the logic for all the methods listed inthe interface. The single most important attribute of an interface isits implementation independence. The contract enforced by an interfacedoes not make any assumptions about how the methods are implemented; theimplementation strategy is left to the implementing class. By enforcingwhat a class of objects must do, without placing restrictions on*how* it is done, interfaces play a key role in making thebehavior of objects more flexible.

The first step in the design process is discovering the necessaryinterfaces. A good way to start is to develop a rough sketch of themain program; the main program should not change whenever animplementation is revised. A sketch of the main program for the gameplaying framework is shown in **Listing 1**.

```
IFactory factory = new Factory();IBoard board = factory.getBoardInstance(args[0]);
boolean bContinue = true;while (bContinue) {    board.resetState();     System.out.
println("\nStarting Board:\n" + board);         for (Iterator i = board.playerIterator();
i.hasNext(); ) {                IPlayer player = (IPlayer)i.next();             System.
out.println(player + ": ");             player.takeTurn(board);         System.out.
```

```
println("\n" + board);  }                       if (board.getWinner() !=
null) {         System.out.println("Winner: " + board.getWinner() + "!");       }
else {          System.out.println("Tie!");             bContinue = playAgain();  }}
```

Listing 1: A rough sketch of the main program.

The code in **Listing 1** can play many differentgames, using a variety of strategies. However, the implementationdetails, such as the type of game and the players that are playing, arenot obvious from looking at the code. Hiding the implementation detailswas accomplished by representing the game and the players involved, onan abstract level using interfaces.

In any object-oriented program, the only time a specific type needs tobe revealed is during instantiation [**1**]. Thisproperty is exploited in **Listing 1** by representingthe current game using the `IBoard` interface. Observe howthe type of game is "hidden" behind this interface and is determined bya "factory" responsible for creating the game. The factory isrepresented using the `IFactory` interface shown in **Listing 2**. As will be shown later, this is an exampleof the **builder design pattern** [**2**]because the factory constructs the board object out of other objects. The use of **creational design patterns** [**2**] is a key component to interface-based designbecause they make it so easy to change the concrete components used byan application.

```
public interface IFactory{      public IBoard getBoardInstance(String strConfigFile);}
```

Listing 2: The IFactory interface.

In addition to creating an abstraction for the type of game, it is alsoimportant to create an interface to represent the players. Theinterface will support the need to easily change a player's strategy. The types of players are hidden from the main program by way of the`IPlayer` interface. Instead of being directly involved inthe creation of the players, the main program consults the board for theplayers and the associated turn sequence. The players and the turnsequence are abstracted by taking advantage of the **iteratordesign pattern**. Iterators allow a sequence of objects to beenumerated without any knowledge of how these objects are contained orthe concrete type of objects being enumerated [**2**]. For example, the board object in **Listing 1** providesthe main program with a player iterator that makes it possible toenumerate the set of players. The responsibilities and behavior of anyplayer that is capable of playing a game is summarized in the`IPlayer` interface (**Listing 3**).

```
public interface IPlayer{       public String getDescription(); public
String getShortDescription();   public void takeTurn(IBoard board);     public
void setEvaluator(IEvaluator evaluator);}
```

Listing 3: The `IPlayer` interface.

The two most important methods in the `IPlayer` interface are`takeTurn()` and `setEvaluator()`. The`takeTurn()` method asks a player to take a turn using thespecified board. The player's `setEvaluator()` methodaccepts any object that supports the `IEvaluator` interface(**Listing 4**). The `setEvaluator()` method makesit possible to configure any player with a specific, and more effective,strategy for evaluating the current state of the game board. Theability to change evaluation strategies is an example of the**strategy design pattern** [**2**] and isanother example of how interfaces can increase flexibility.

```
public interface IEvaluator{    public int evaluate(IBoard board, IPlayer player);}
```

Listing 4: The `IEvaluator` interface.

Up to this point, the use of interfaces has helped create a main programwith no dependencies on the game, the types of players, or the strategiesbeing employed. For a better understanding of how this is possible, itwill help to examine the IBoard interface (**Listing 5**).

```
public interface IBoard{        public void pushMove(IMove move);       public
IMove peekMove();       public IMove popMove(); public void setPlayers(IPlayer
[] players);    public Iterator playerIterator();       public void moves
(Collection col);       public IPlayer getWinner();     public void resetState();}
```

Listing 5: The IBoard interface.

As shown in **Listing 5**, all boards contain a stackof moves. A turn can be taken by pushing a move onto this stack usingthe pushMove() method, and the most recent move can beobtained, or undone, using the peekMove() andpopMove() methods. In addition to moves, each board mustknow the players that are permitted to make moves, and the order inwhich these players can take their turn. The IBoardinterface provides this functionality with the setPlayers()and playerIterator() methods. When the game has ended, thegetWinner() method can be used to obtain a reference to thewinner of the game; null will be returned if the game endedin a tie. All boards must also provide a collection of the currentlyavailable moves using the moves() method. Finally, theresetState() method can be invoked to prepare the board fora new game. Without making any assumptions about a specific game type, the IBoard interface provides a working abstraction formany different types of games.

Since different games support different types of moves, theIMove interface is used to define these move types. Inaddition, an interface named IPiece is needed to representthe pieces associated with a move (**Listing 6**).

```
public interface IMove{ public IPiece getPiece();}public interface IPiece{}
```

Listing 6: The IMove and IPiece interfaces.

A striking property of the IPiece interface is that it doesnot contain any methods. It may seem strange to define an interfacewithout any methods, especially given the fact that most games can beimplemented using pieces that require only the methods provided byJava's common base class: Object [**5**]. Although not immediately necessary, IPiece was created inorder to keep with the spirit of programming to an interface. As avariety of games are implemented, this interface may prove to be moreuseful.

Figure 1 summarizes the edu.lhup.aipackage, and the interfaces that have been uncovered in order toimplement the generic game playing framework.

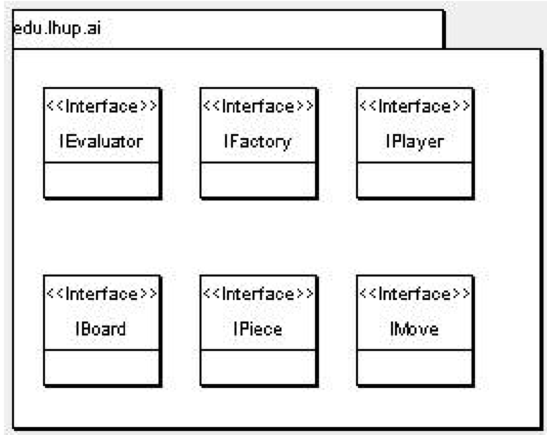

Figure 1: The `edu.lhup.ai` package.

Building Generic Players

Using the interfaces designed earlier, it is possible to implementplayers capable of playing a game with absolutely no detailed knowledgeof the game they are playing. The players can play any game using onlythe abstract view provided by the `IBoard` interface. Thetwo players that will be implemented here are the`RandomPlayer` and the `MinMaxAplhaBetaPlayer`.

The `RandomPlayer` is capable of playing any game byselecting moves randomly. Naturally, this particular participant willnot perform well at most games. However, it is capable of playing anygame, and looking at its implementation does start to demonstrate thepower of the current interface design. Since the`RandomPlayer` does not use any real strategy in order tochoose its moves, it does not require an evaluator. As a result, theonly method that needs to be implemented is `takeTurn()`.

The `RandomPlayer` works with the current game using the`IBoard` interface (**Listing 7**). The useof the `IBoard` interface makes it possible to query the gamefor all of the possible moves. Once the player has a list of thepossible moves, the player randomly selects a move from the availablechoices, and pushes the move onto the specified game's stack. Noknowledge beyond the abstract representations of the game and theallowed moves is required in order to play a game in a random fashion.

```
public void takeTurn(IBoard board){     LinkedList moves = new LinkedList();    board.
moves(moves);   int i = m_random.nextInt(moves.size()); board.pushMove((IMove)moves.
get(i));}
```

Listing 7: The `takeTurn` method for the `RandomPlayer`.

The random player is interesting, but not useful, except for randomlytesting new games. Perhaps it would be more useful to create a playerthat can play with more intelligence, and still have no knowledge of thegame it is playing. To accomplish this, a little background in gametheory is required.

Minimax Search and Alpha-Beta Pruning

In order to implement an intelligent player, a more sophisticatedalgorithm must be used. One possible solution is the**minimax** search algorithm. In its purest form, thisalgorithm plays the game from the current state to all possible endingsby exploring the legal moves at each state. Upon reaching each endstate, an evaluation is made on the quality of the result. Using thisevaluation, each player's move leading up to the end state can bepredicted. Employing this strategy, the best possible sequenceof moves can be discovered, and the best next move determined. Readersthat desire are more detailed description of the minimax searchalgorithm should refer to [**4**].

In a perfect world, the minimax search algorithm is very effectivebecause it examines all possible moves in order to find the path thatwill lead to the best result. However, few games reside in a perfectworld. The limitations of the basic minimax algorithm lie in the verylarge search space that must be examined, and most interesting games areinteresting because they have such a large search space [4]. Fortunately, a minor modification can be madeto the basic minimax algorithm that can decrease the number of movesevaluated. The technique used to prune a minimax search tree is called**alpha-beta pruning** [4].

Implementing the `MinMaxAlphaBetaPlayer`

The `MinMaxAlphaBetaPlayer` uses a minimax search algorithm,along with alpha-beta pruning, in order to play a game. The mostinteresting thing about this implementation is that it does this withoutany knowledge of the specific game it is playing. Once again, thesimple `IBoard` interface is all that is needed to implementthis more sophisticated game player.

The implementation chosen for the `MinMaxAlphaBetaPlayer` isbased on the pseudo code presented in [4]. **Listing 8** shows the implementation of the`takeTurn()` method that performs the minimax search. Thismethod queries the board for the current list of possible moves. Itthen examines each move and assigns it a rating. The move with the bestrating is chosen as the next move.

```
public void takeTurn(IBoard board){     int bestRating = Integer.MIN_VALUE;
IMove bestMove = null;  List moves = new LinkedList();  board.moves(moves);     for (int
i = 0; i < moves.size(); i++) {         IMove move = (IMove)moves.get
(i);            board.pushMove(move);           int currentRating
=                       getRating(board, 0, Integer.MIN_VALUE,
Integer.MAX_VALUE);             board.popMove();                if (currentRating
> bestRating) {                 bestMove = move;                        bestRating
= currentRating;                }       }       board.pushMove(bestMove);}
```

Listing 8: The `takeTurn` method for the `MinMaxAlphaBetaPlayer`.

Most of the work done by the `MinMaxAlphaBetaPlayer` playertakes place in the `getRating()` method. This method iscalled by `takeTurn()`, and recursively searches the tree ofmoves looking for an end state. An end state occurs when a cutoff limitis reached, the game ends in a tie, or one of the players wins. Thecutoff limit is used for games which contain search spaces that are toolarge to be searched in their entirety. Once an end state is found, itis evaluated using the player's evaluator object.

The implementation for `getRating()` is not shown in order toavoid getting bogged down in the details of the algorithm (see thepseudo code presented in [4]). Instead, thereader is asked to focus on the following important points. First, the`takeTurn()` and the `getRating()` methodsinteract with the game using the `IBoard` interface. Usingthe `IBoard` interface is what makes it possible for theparticular player to play any implementation of `IBoard`.Finally, rating each end state is performed by the player's evaluatorand makes it possible to inject a problem-specific evaluation into thealgorithm.

In an effort to keep the player completely general, a default evaluatorthat does not need game specific information is required. The default`Evaluator` class (**Listing 9**) can onlyevaluate terminal states, and assigns a rating of one for a victory,zero for a tie, and a minus one for states that result in defeat.

```
public class Evaluator implements IEvaluator{   public int evaluate(IBoard board, IPlayer
maxPlayer)      {               int rating = Integer.MIN_VALUE;         if (board.getWinner
() == null) {                   rating = 0;             } else if (board.getWinner() ==
maxPlayer) {                    rating = 1;             } else
```

```
{                          rating = -1;            }               return rating;  }}
```

Listing 9: The default `Evaluator` class.

As mentioned earlier, both of these implementations can be altered tocreate more effective players that take advantage of the details of aspecific game. An actual game needs to be implemented in order toillustrate the creation of such specialized players.

Tic-Tac-Toe

All of the interfaces and classes introduced reside in the`edu.lhup.ai` package, and support a variety of games. Toput these classes to work, the game of tic-tac-toe was implemented. Inorder to implement tic-tac-toe, the `IBoard`,`IMove`, and `IPiece` interfaces must beimplemented. In addition, an iterator must be created that allows theenumeration of the tic-tac-toe players. These classes have been placedin the `edu.lhup.ai.tictactoe` package.

Tic-tac-toe requires three different types of pieces:`XPiece`, `OPiece`, and `EmptyPiece`. In addition, a `Move` class was created to associate a piecewith a board location. The rules and logic of tic-tac-toe areimplemented in the `Board` class and the`PlayerIterator` class.

The `edu.lhup.ai.tictactoe` package also contains a custom`Evaluator` that demonstrates how the generic`MinMaxAlphaBetaPlayer` can be improved by taking advantageof the rules of a specific game. If you recall, a cutoff point wasadded to the `MinMaxAlphaBetaPlayer` for the case when, evenwith alpha-beta pruning, a search takes too long. However, using acutoff can lead to something called the **horizon problem**[4]. The horizon problem arises when the cutoffpoint is reached and the evaluator is asked to evaluate a game statethat is not terminal. As a result, the evaluator must be capable ofdeciding how good a state is, even if there is currently no winner.Failure by the evaluator to see what is about to happen in the game canlead to a poor evaluation. Creating a custom evaluator is a good way toalleviate the horizon problem because additional logic can be added thatuses knowledge of the game to look for trouble beyond the cutoff.

The tic-tac-toe specific evaluator examines terminal and non-terminalgame states by judging how close a particular state is from victory ordefeat; this can minimize the horizon problem. Players using thisevaluator will be able to play tic-tac-toe much quicker because they cantake advantage of a cutoff. The disadvantage is that players using thisevaluator become tic-tac-toe specialists, and can no longer play avariety of different games. Because the focus of this article is on theframework, and how it supports several different types of games, theactual tic-tac-toe classes are not covered in detail. A complete classdiagram of all of the classes in the framework, along with thetic-tac-toe specific classes, is shown in **Figure 2**. For the purposes of testing the framework, the code from **Listing 1** was used to create the `PlayGame`class.

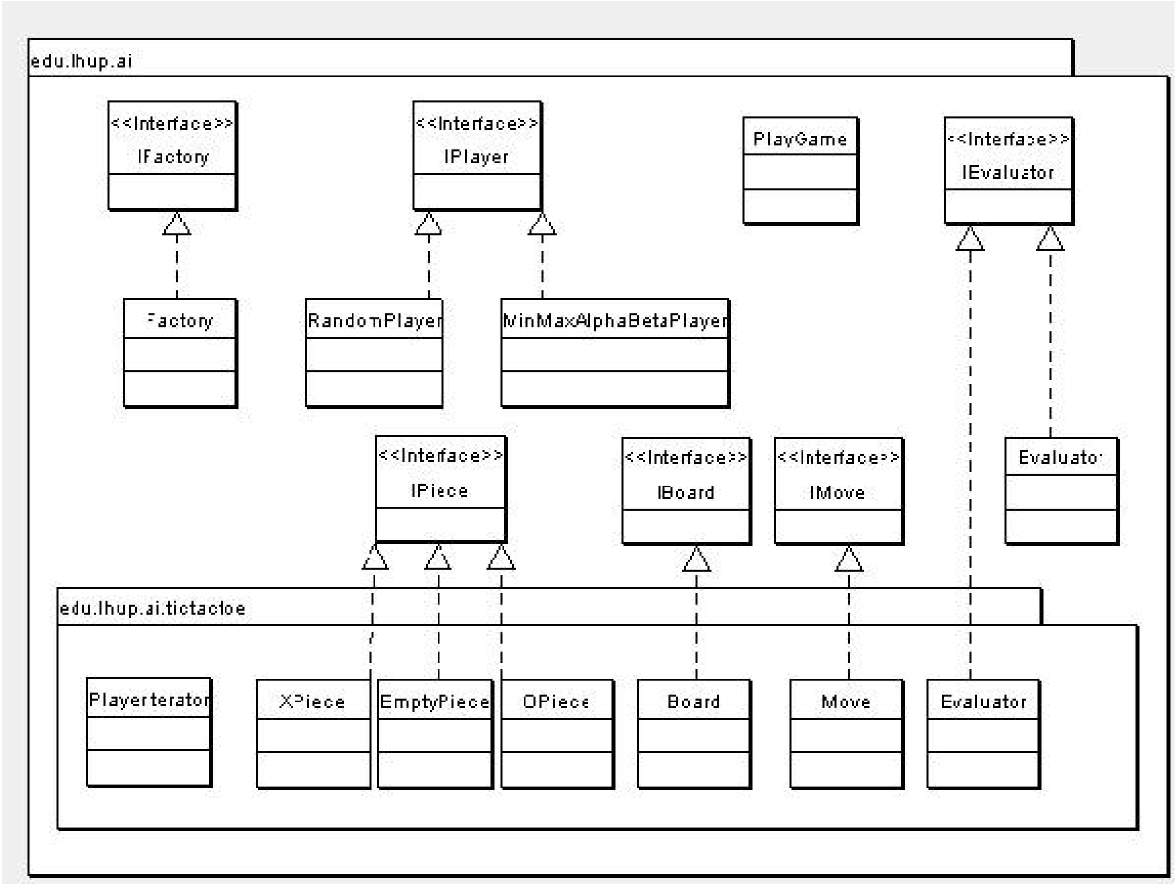

Figure 2: Framework class diagram.

Configuring the Framework

The main program sketched earlier and integrated with the`PlayGame` class relies on the `Factory` class inorder to get an instance of the current game board. The flexibility ofthe framework can be taken advantage of by placing the name of theactual game created by the `Factory` in an XML configurationfile. The XML code is parsed using the classes in the`javax.xml.parsers` package [**5**]. Thefactory builds the game board using the class names located in theconfiguration file. Because the creation of the board is rathercomplex, consisting of building the board using players and evaluators,the builder design pattern was implemented [**2**]. The creation of classes, and instances of these classes, is accomplishedby using Java's support for dynamic class loading [**3**].

An example of a configuration file that results in a game of tic-tac-toebetween a `RandomPlayer` and a`MinMaxAlphaBetaPlayer` can be found in **Listing 10**.

```
<game>  <gameFactory>           <boardClass>edu.lhup.ai.tictactoe.
Board</boardClass>              <player>                        <playerClass>edu.lhup.
ai.RandomPlayer</playerClass>           </player>
<player>                    <playerClass>edu.lhup.ai.
MinMaxAlphaBetaPlayer</playerClass>
<cutoff>1000</cutoff>
<evaluator>                                 <evaluatorClass>edu.lhup.ai.
Evaluator</evaluatorClass>                      </evaluator>            </player>
</gameFactory></game>
```

Listing 10: Tic-tac-toe configuration file.

Educational Uses

This configuration file makes it possible for new games and new playersto be implemented and plugged into the framework. Readers areencouraged to download the source code, implement a new game, and seehow the default `MinMaxAlphaBetaPlayer` performs. It wouldalso be a good exercise to create an even better player by implementingan evaluator that is specific to the new game and plugging it into the`MinMaxAlphaBetaPlayer`.

Further extending of the framework could also be assigned in anundergraduate course in object-oriented programming or artificialintelligence, and competitions can be setup to assess the effectivenessof the student's work. The combination of game programming and goodnatured competition is often a strong motivation to undergraduatestudents, and should be more than enough to inspire livelyparticipation. Interested students or instructors can obtain a digitalcopy of all of the source code, along with the Javadocs, at **http://www.lhup.edu/mcohen/oogt**.

Conclusion

The game playing framework described above demonstrates how thegenerous use of interfaces can lead to a flexible design. Thisflexibility is evident in the framework's ability to support manydifferent games using a variety of game playing strategies. Inaddition, the framework can be customized to support game specificstrategies. Most importantly, this customization can be done via aconfiguration file, and imposes minimal impact on the existing codebase. Identifying the interfaces required by an application, and thenprogramming to these interfaces, results in an application that happilyaccepts implementation changes. Instructors and students alike areencouraged to take advantage of the flexibility of the framework to helpwith the education of object-oriented programming and game theory.

References

1
Bloch, Joshua. 2001. *Effective Java.* Reading, MA: Addison-Wesley.

2
Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns.* Reading, MA: Addison-Wesley.

3
Gosling, James, Bill Joy, and Guy L., Jr. Steele. 1996. *The Java Language Specification.* Reading, MA: Addison-Wesley.

4
Russell, Stuart, and Peter Norvig. 1995. *Artificial Intelligence.* Upper Saddle River, NJ: Prentice-Hall.

5
Sun Microsystems Inc. 2002.*Java 2 Platform, Standard Edition, v 1.4.1, API Specification.* <**http://java.sun.com/j2se/1.4.1/docs/api/index.html**>

Biography

Mark Cohen (**mcohen@lhup.edu**) is aninstructor in the Business Administration, Computer Science, andInformation Technology department at Lock Haven University, and agraduate student associated with the Applied Cognitive Science Lab inthe School of Information Sciences and Technology at Pennsylvania StateUniversity. His research interests include object-oriented programmingand artificial intelligence. He received a M.S. in Computer Science fromDrexel University, a B.S. in Electrical Engineering from LafayetteCollege, and has over ten years of experience developing software for thehealth care and pharmaceutical industries.