UNBC **CPSC 101 Team Term Project** Winter 2018

Interface Specification

Contents

List of Figures

1 The Tournament

When each team has completed its implementation of Score 4, the various teams' AIs will compete in a tournament.

In order for your team to participate in the tournament, you must e-mail a `.jar` file to Dr Casperson that contains your AI program.

This document specifies how to do so.

2 Packaging

In order that your classes not collide with other teams' classes or Dr Casperson's referee classes, your code *must be* packaged in a package name that starts with `ca.unbc.cpsc.`*`team_name`*, where team name might be, say, `nescafe`.

This prevents your classes from colliding with another team's classes.

3 The General Idea

The AI that you provide to Dr Casperson functions as a stand-alone object. Dr Casperson's referee will call your AI object using the methods of an interface that your AI object must implement (see Figure 1).

Dr Casperson's referee and your AI must exchange information about locations where moves are played, and the colour that your AI is playing in a given game. When your AI returns a move location from its `.requestMoveLocation()` method, it can return any object it wants, but that object must conform to the `Location` interface specified in Figure 3. Similarly, when the tournament referee tells you an opponent's move via `.opponentPlays(Location l)`, all that you can assume about the object sent to your AI is that it conforms to Figure 3.

In the same vein, the colour that your AI receives via the `.startGameAs(Colour c)` method only guarantees the methods shown in Figure 4.

The remainder of the details of the `Player` specification are provided in Section 4.

3.1 How Dr Casperson gets an AI Object

Dr Casperson will dynamically load a class from your `.jar`-file that has a static method with signature

```
  public static
  ca.unbc.cpsc.score4.interfaces.Player getAI() ;
```

that is, your class' static `getAI()` method returns an object that conforms to the interface shown in Figure 1.

In order for this to work, you must supply Dr Casperson with

- a `.jar` file, whose name indicates the team somehow
- the complete name of a class in the jar, that is,
  - a package name (that starts with `ca.unbc.cpsc.`*`team_name`*), and
  - a simple class name.

4 The `Player` interface

The object that your `getAI()` method returns can be a member of any class that you choose, but it must implement the
`ca.unbc.cpsc.score4.interfaces.Player`
interface. Other requirements that your AI must satisfy are specified first below, and then there is a method-by-method description of the `Player` interface.

4.1 Well-behaved Players

Your AI player must be well-behaved and make minimal assumptions about its environment. That is, your AI player shouldn't call `System.exit`, use large amounts of memory, take excessively long to play, or assume anything about the surrounding file system.

4.2 Exception handling

When you implment an interface or extend a class overriding methods must not throw any more visible exceptions than the method being overridden. In order to allow for the possibility that your implementation's methods throw exceptions, all of the methods of the
`ca.unbc.cpsc.score4.interfaces.Player`

```
package ca.unbc.cpsc.score4.interfaces;

import ca.unbc.cpsc.score4.enums.GameOverStatus;
import ca.unbc.cpsc.score4.interfaces.Colour;
import ca.unbc.cpsc.score4.interfaces.Location;
import ca.unbc.cpsc.score4.exceptions.PlayerException;

public interface Player
{
  public abstract void reset()          throws PlayerException;

  public abstract void startGameAs(Colour c)
                                        throws PlayerException ;
  public abstract void noteOpponentsId(int id)
                                        throws PlayerException ;
  public abstract void opponentPlays(Location ell)
                                        throws PlayerException ;
  public abstract Location requestMoveLocation()
                                        throws PlayerException ;
  public abstract Location retry()      throws PlayerException ;

  public abstract void noteGameOver(GameOverStatus whatHappened)
                                        throws PlayerException ;

}   // end interface Player
```

Figure 1: Parts of the `Player` interface

interface state that they may throw a
    `ca.unbc.cpsc.score4.exceptions.PlayerException`
exception.

If you have a method that might otherwise throw an exception, you can wrap it in a `PlayerException` so that your method conforms to the interface. For instance you could write code like that shown in 2 on the following page to handle exceptions that your own code throws.

Try to avoid such complications if you can.

```
package ca.unbc.cpsc.flat_white.ai;

import ca.unbc.cpsc.flat_white.score4.Colour;
import ca.unbc.cpsc.score4.exceptions.PlayerException;

public class AIPlayer
    implements ca.unbc.cpsc.score4.interfaces.Player
{

  // ....

  public void reset() throws PlayerException
      {
      try
          {
          // stuff that might throw a GameStateException
          }
      catch (GameStateException gse)
          {
          // package the exception up to conform and then
          // rethrow it.
          throw new PlayerException("bad reset", gse) ;
          }
      }

  // ... lots more
}
```

Figure 2: An Example of Handling Exceptions

4.3 Methods of the `Player` interface

`void reset()` This method should cause the AI to go to its beginning of game state, ready to play either white or black. An AI should be prepared to accept a `reset()` call at any time, because the tournament referee might need to cope with errors in the opponent's program.

`void noteOpponentsId(int id)` The tournament referee will call this method of your AI after doing a `reset()` to let the AI know who it is playing against. Each opponent has a unique identification number. You may choose to ignore this information if you wish.

`void startGameAs(Colour c)` (See Section 6 on page 8 for the definition of `Colour`.) The tournament referee will call this method of your AI after doing a `reset()` to let the AI know what colour it is playing in this game. Your AI can rely on this method being called only after a `reset()` or `noteOpponentsId(int)` method call.

`Location requestMoveLocation()` (See Section 5 on the following page for the definition of `Location`'s.) The tournament referee will call this method when it is your AI's turn to play. Your AI should return a valid location (that is, the location of a non-empty peg) where it wishes to play. If the tournament referee doesn't like your choice of location, it will call the AI's `retry method`, otherwise it will silently accept your AI's move, and call one of

- `reset()`
- `noteGameOver(...)`
- `opponentPlays(...)`

`Location retry()` The tournament referee will call this method only if the previous call to `requestMoveLocation()` or `retry()` resulted in an invalid move. Note that the referee *will not have* added your previous move to its understanding of the game board.

`void opponentPlays(Location ell)` The tournament referee calls this method after your opponent has made a valid move to let your AI know what its opponent has done. Beware that the tournament referee may call `reset()` or `noteGameOver(...)` instead, depending on what happens.

`void noteGameOver(GameOverStatus ...)` When the game ends normally, either through one player making a winning move or the tournament referee noticing that neither player can win (no matter what they do), the tournament referee calls this method. The result received is from the AI's point of view (that is, `GameOverStatus.LOSS` means that the AI lost this game). Beware that

```
package ca.unbc.cpsc.score4.interfaces;

public interface Location
{
  public static final int MAX_ROW    = 3;
  public static final int MIN_ROW    = 0;
  public static final int MAX_COLUMN = 3;
  public static final int MIN_COLUMN = 0;

  public abstract int getRow()    ; // in [MIN_ROW,MAX_ROW]
  public abstract int getColumn() ; // ...
}
```

Figure 3: The `Location` interface

```
package ca.unbc.cpsc.score4.interfaces;

public interface Colour
{
  public abstract boolean isBlack() ;
  public abstract boolean isWhite() ;
}
```

Figure 4: The `Colour` interface

the tournament referee may insead call `reset()` instead, depending on what happens.

5 The `Location` interface

See Figure 3.

Locations are how moves are communicated between AI's and the tournament referee. They are two-dimensional (row and column, but not height) and zero-indexed. When your AI responds to a `requestMoveLocation` or `retry` request it can return any object that it wants provided that that object conforms to the
`ca.unbc.cpsc.score4.interfaces.Location`
interface. However, note that when the AI receives a `Location` via `opponentPlays` it cannot assume anything other than what the interface specifies.

```
package ca.unbc.cpsc.score4.exceptions;

public class PlayerException extends Exception
{
  public PlayerException() { this("Unknown Player Exception") ; }
  public PlayerException(String s)                 { super(s) ; }
  public PlayerException(Throwable t)              { super(t) ; }
  public PlayerException(String s, Throwable t)    { super(s,t) ; }
}
```

Figure 5: The `PlayerException` class

```
package ca.unbc.cpsc.score4.enums;

public enum GameOverStatus { LOSS, DRAW, WIN ; }
```

Figure 6: The `GameOverStatus` enumeration

6 The `Colour` interface

See Figure 4 on the preceding page. In the `Player` interface, `Colour`'s are only used to indicate whether the AI plays first or second.

7 The `PlayerException` Class

See Figure 5. As discussed in Section 4 the only exceptions that the AI is allowed to throw are `PlayerException`'s. These are a standard non-`RuntimeException` `Exception` class.

Note that these exceptions inherit `getCause()` and `getMessage()` from `Exception` (originally specified in `java.Lang.Throwable`).

8 The `GameOverStatus` enumeration

The `Gameoverstatus` enumeration is shown in Figure 6. Note that the tournament referee reports from the AI's point of view.

9 History

This is the second version of this specification, created on 2018-02-26. It repairs the `GameOverStatus` figure, and corrects a few typos.

Previous version

2018-01-12 This was the initial version.

Dr David Casperson 2018-01-12