**IPL Draughts Game**

**com.ipl.training.induction.draughts**

**System Design Specification**

**Project Reference 999/170**

**Document Reference SDS**

**13 June 2013**

**Issue 5**

**Copyright Notice**

This document has been prepared by IPL Information Processing Limited (“IPL”) and, subject to any existing rights of other parties, IPL is the owner of the copyright of this document. No part of this document may be copied, reproduced, stored in a retrieval system, disclosed to a third party or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of IPL.

**© IPL Information Processing Limited 2013**

**Document History**

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Issue | Author | Comments |
| 10/6/2010 | 1 | Brian Bassil |  |
| 02/07/2010 | 2 | Christine Harrison | Minor typos and clarifications |
| 21/10/2010 | 3 | Christine Harrison | Corrections after first year of course |
| 22/11/2011 | 4 | Christine Harrison | Updated for new Java Standards |
| 13/06/2013 | 5 | Christine Harrison | Minor corrections to the Data Flow section |

Contents

[1 System Identity 1](#_Toc358901536)

[2 Description 1](#_Toc358901537)

[3 Interface 1](#_Toc358901538)

[4 Structure 2](#_Toc358901539)

[5 Element Descriptions and Interfaces 3](#_Toc358901540)

[5.1 Package com.ipl.training.induction.draughts 3](#_Toc358901541)

[5.2 Package com.ipl.training.induction.draughts.controller 3](#_Toc358901542)

[5.3 Package com.ipl.training.induction.draughts.model 4](#_Toc358901543)

[5.4 Package com.ipl.training.induction.draughts.view.ui 6](#_Toc358901544)

[5.5 Package com.ipl.training.induction.draughts.view.remote 7](#_Toc358901545)

[6 Computer Players 7](#_Toc358901546)

[6.1 Game Play Sequence 8](#_Toc358901547)

[6.1.1 Construction 9](#_Toc358901548)

[6.1.2 Start playing 9](#_Toc358901549)

[6.1.3 Make move 9](#_Toc358901550)

[6.1.4 Stop Playing 10](#_Toc358901551)

[7 Model Data 10](#_Toc358901552)

[7.1 Board Layout 11](#_Toc358901553)

[7.2 Turn Data 11](#_Toc358901554)

[7.3 Game Data 11](#_Toc358901555)

[8 Data Flow 11](#_Toc358901556)

[8.1 Model to Controller 11](#_Toc358901557)

[8.1.1 ADD\_HISTORY 11](#_Toc358901558)

[8.1.2 STATUS 11](#_Toc358901559)

[8.1.3 SET\_SQUARE 12](#_Toc358901560)

[8.1.4 ERROR 12](#_Toc358901561)

[8.1.5 GAME\_DATA\_UPDATED 12](#_Toc358901562)

[8.1.6 SET\_HINTS 12](#_Toc358901563)

[8.1.7 SET\_SELECTED 12](#_Toc358901564)

[8.1.8 PLAYER 12](#_Toc358901565)

[8.1.9 SEND\_CLICK 12](#_Toc358901566)

[8.1.10 REMOVE\_HISTORY 12](#_Toc358901567)

[8.1.11 GAME\_END 12](#_Toc358901568)

[8.2 Controller to Model 12](#_Toc358901569)

[8.2.1 newGame() 12](#_Toc358901570)

[8.2.2 setClick() 13](#_Toc358901571)

[8.2.3 undo() 13](#_Toc358901572)

[8.2.4 export() 13](#_Toc358901573)

[8.3 View to Controller 13](#_Toc358901574)

[8.3.1 addView() 13](#_Toc358901575)

[8.3.2 squareClicked() 13](#_Toc358901576)

[8.3.3 newGame() 13](#_Toc358901577)

[8.3.4 resetBoard() 13](#_Toc358901578)

[8.3.5 connectionFailed() 13](#_Toc358901579)

[8.3.6 error() 13](#_Toc358901580)

[8.3.7 undo() 13](#_Toc358901581)

[8.3.8 export() 13](#_Toc358901582)

[8.4 Controller to View 13](#_Toc358901583)

[8.4.1 ADD\_HISTORY 13](#_Toc358901584)

[8.4.2 STATUS (acted on by UI) 13](#_Toc358901585)

[8.4.3 SET\_SELECTED (acted on by UI) 13](#_Toc358901586)

[8.4.4 ERROR (acted on by UI and RemoteView) 14](#_Toc358901587)

[8.4.5 SET\_SQUARE (acted on by UI) 14](#_Toc358901588)

[8.4.6 SET\_HINTS (acted on by UI) 14](#_Toc358901589)

[8.4.7 PLAYER (acted on by UI) 14](#_Toc358901590)

[8.4.8 NEW\_GAME (acted on by UI) 14](#_Toc358901591)

[8.4.9 GAME\_DATA\_UPDATED (acted on by UI) 14](#_Toc358901592)

[8.4.10 REMOVE\_HISTORY (acted on by UI) 14](#_Toc358901593)

[8.4.11 SEND\_CLICK (acted on by RemoteView) 14](#_Toc358901594)

[8.4.12 REMOTE (acted on by RemoteView) 14](#_Toc358901595)

[8.4.13 DISCONNECT (acted on by RemoteView) 14](#_Toc358901596)

[9 Supporting Documentation 14](#_Toc358901597)

# System Identity

System Name - Draughts Game  
System Identity - com.ipl.training.induction.draughts

# Description

The system is a Draughts Game.

The system provides the following functions:

* Two players can play draughts according to the ruled defined in SRS\_999170.docx.
* The players can be any combination of human and computer.
* One player may be running on a separate instance of the draughts game.

# Interface

There are two interfaces to the system

* The graphical user interface which is defined in the com.ipl.training.induction.draughts.view.ui package.
* The remote interface which is defined in the com.ipl.training.induction.draughts.view.remote package.

# Structure



Figure Package diagram of com.ipl.training.induction.draughts.



Figure Package diagram of com.ipl.training.induction.draughts.view.

The structure clearly shows how the system uses the MVC pattern. The main method that constructs the system resides outside of the pattern in a class DraughtsGame.

# Element Descriptions and Interfaces

## Package com.ipl.training.induction.draughts

The draughts package contains the main game class. This class is responsible for constructing the model, views and controller and ensuring that they are connected to each other. Once they are constructed this class performs no further function.

## Package com.ipl.training.induction.draughts.controller

This package defines the controller and all associated interfaces. The controller only communicates to the views and model through defined interfaces. This allows different concrete instances of these interfaces to be used with little change to the controller.



Figure Component diagram showing communication to the view and model from the controller.

The diagram shows that the controller only communicates with the views and model through the interfaces IDraughtsView and IDraughtsModel respectively.

The controller provides the logic that interfaces the views and the model. Responsibilities of the controller are

* Reset the board for a new game, set up the players and initialising the model;
* Providing an interface for the views and model to feed data back to the controller and trigger events.



Figure Package diagram of the controller package

The class structure shows that the controller package will also contain a few helper classes.

GameState is an enumeration describing the state of the game, e.g. in progress, won, lost etc

PlayerData is a class collating all the information about a player. It contains the name of the player and the type of that player, human, computer or remote.

GameData is a class collating all of the information about a game. It contains the 2 players, whether the game is local or remote, and any initial position.

Square is an enumeration describing what is on a particular square.

Square data contains the data for a square on the board.

DraughtsChangeEvent is a class that extends PropertyChangeEvent. It simply overrides the toString() method for debugging.

## Package com.ipl.training.induction.draughts.model

All processing related to board positions and legal moves is handled within this package along with the implementation of the computer player. This package provides support for reading and writing PDN files.



Figure Package diagram of the model package

The model contains all of the data associated with a game as well as the business logic that processes that logic.

The class DraughtsModel is the central class of the package. The model performs 2 major roles:

* Maintain a board position with all legal moves;
* Play a move based upon an algorithm.

BoardLayout, MoveData,and NextSquare are all used in the process of storing the current position and identifying legal moves. Note TurnData stores the move history which is used when undoing a move.

AbstractComputerPlayer is an abstract class that all computer playing algorithms must extend. This ensures that any algorithm plays a legal move within a set time. The model will create an instance of an AbstractComputerPlayer for each computer player. The model uses the system properties BLACK\_COMPUTER\_PLAYER and WHITE\_COMPUTER\_PLAYER to identify which concrete algorithm is used – see section 6 for further details

* FirstMovePlayer is a class that simply plays the first legal move that has been identified. It is supplied in the first group exercise.
* InductionMovePlayer is a skeleton algorithm which uses an Evaluator class to determine the best move. It is supplied in the second group exercise – your challenge will be to write a small part of this ‘intelligent’ player. There will be a competition at the end of the course between the different groups’ games.
* AnotherMovePlayer is the ‘intelligent’ algorithm produced by the authors of the course. It will be used in the competition by the presenters of the course.

## Package com.ipl.training.induction.draughts.view.ui

This package provides the user interface.



Figure Package diagram of the ui package

DraughtsView displays the state of the game based upon the data received from the controller. It also allows the user to make a move.

DraughtsView will display a NewGameDialog when requested to by the controller.

## Package com.ipl.training.induction.draughts.view.remote

This package provides communication between two separate Draughts Game instances.

RemoteView is an implementation of com.ipl.training.induction.draughts.IDraughtsView that provides support for remote players. It acts as both a client and a server using Java RMI.



Figure Package diagram of the remote package

RemoteView accepts connections from other instances of the draughts game. It will attempt to connect to a remote instance when a player is identified as being remote.

RemoteView passes all clicks from the controller over to the RemoteView of the other instance. It also passes all clicks received from the other RemoteView to the controller.

AcceptGameDialog displays a dialog when a game request is received from a remote instance. This will allow the user to enter a name and select the type of player (human or computer).

# Computer Players

Either player may be controlled by the computer. The algorithm used to control a player will be specified as a JVM argument so that different algorithms can be added later, e.g.:

* BLACK\_COMPUTER\_PLAYER=FirstMovePlayer
* WHITE\_COMPUTER\_PLAYER=FirstMovePlayer

The game has been designed so that the JVM arguments can be put into a draughts.properties file which is read when the game starts up.

The ability to specify different computer player algorithms has been implemented by having an abstract base class which a computer player must extend. The application communicates with the computer algorithm through the interface specified in AbstractComputerPlayer.



As shown, *in italics*, the method determineMoves is abstract and must be implemented by the algorithm.

AbstractComputerPlayer is responsible for telling the algorithm when it should determine the best move. It will also stop the algorithm when the allotted time has expired.

The algorithm will need to exist in the package com.ipl.training.induction.draughts.model so that it can call the following methods on the AbstractComputerPlayer

* getTimeRemaining
* stopPlaying
* storeClick

## Game Play Sequence

The draughts model is responsible for ensuring each player moves in turn. The model waits until a legal move has been made then allows the other player to move. Play starts with the black player, unless a PDN file has been loaded where the next player to move is defined in the PDN.

All players (computer, human and remote) inform the model they have moved by calling setClick on the model. When the model detects the game has ended it stops any further moves being made.

The following sequence diagram shows the entire process for a computer player.



### Construction

If the player is a computer player call loadPlayer. This will read the JVM property for that player (BLACK\_COMPUTER\_PLAYER or WHITE\_COMPUTER\_PLAYER) then use the return as a class name to be loaded. This class must extend AbstractComputerPlayer.

The Construction of the player will also construct the AbstractComputerPlayer parent class.

In this example the class loaded is InductionMovePlayer. This is a version that uses a separate class, Evaluator, to rate a particular board position.

### Start playing

AbstractComputerPlayer extends thread. We trigger the thread with start which causes the run method to be executed.

### Make move

AbstractComputerPlayer calls determineMoves on InductionMovePlayer. InductionMovePlayer then determines the best move, making use of Evaluator. Before returning it should store the move it wishes to make as a set of discrete clicks. This is performed by calling setClick for each move.

Once determineMove returns AbstractComputerPlayer makes the moves by calling setClick on the model for each click.

If InductionMovePlayer fails to make a valid move in time it will make the first valid move.

### Stop Playing

Once a move has been made the thread calls wait() so that it will idle until notified it should run again.

# Model Data

The model contains three types of data

* The board layout.
* The turn history
* The game data



Note that computerPlayers is not just data but an implementation of the AbstractComputerPlayers class. This contains processing for each computer player in the system.

## Board Layout

The board layout is stored in the class BoardLayout. The board layout contains information about

* What is on each of the 32 squares;
* Which player is to move;
* What the possible moves are for that player.

BoardLayout is immutable. This allows computation engines to store a BoardLayout and return to it later knowing it will not have been modified.

Normally BoardLayouts are constructed by taking an existing BoardLayout and applying a move to it. However at the start of a game an initial BoardLayout is created by supplying a FEN tag. If the FEN tag is empty then a default board is constructed.

The construction of a BoardLayout will cause it to calculate all of the possible moves. As the class is immutable this list will never need to be regenerated.

## Turn Data

Turn data stores a BoardLayout and a move. This is used for:

* Displaying the move history in the UI;
* Providing move history for undo support.

The BoardLayout stored is the position of the board before the move was made. This is used when a move is undone.

DraughtsModel stores a stack of TurnData objects. When a move is made this is pushed onto the stack. When a move is undone a move is popped from the stack.

## Game Data

Game data stores information that persists for the entire game in the class GameData. This is

* The name and type of each player;
* Any FEN tag used to start the game;
* The name of the remote host;
* Whether this is a network game.

# Data Flow

## Model to Controller

Data from the model to the controller is sent as property changes.

### ADD\_HISTORY

An ADD\_HISTORY property is fired when a turn has been made. This occurs in changesTurn().

### STATUS

A STATUS property is fired when the model status changes:

* Fired in checkSquare() or setClick() when a square sent from the view cannot be selected;
* Fired in chagesTurn() when the game is over (with information about the winner);
* Fired in newGame() when the model is ready to play.

A STATUS property is also fired when an illegal move is being attempted. This will cause a message to be displayed on the UI.

A STATUS property is also fired in undo() if undone is attempted under the following circumstances:

* No move has been made;
* The game is over;
* A computer move.

### SET\_SQUARE

This is fired in to indicate the model has been updated with a move from a computer player. The information will be in the format of a square ID and what is now on that square.

### ERROR

This is fired if the model detects an internal error. This will always be followed by initiating a new game.

### GAME\_DATA\_UPDATED

This is fired when the model detects that is a new game.

### SET\_HINTS

Fired in setClick() to indicate which squares a selected piece may be moved to (i.e. move hints). The UI is then free use or ignore them.

Fired in undo() to clear any hints.

### SET\_SELECTED

Fired in setClick() to select a square. This will be the initial click to select a piece.

Fired in undo() to clear any piece selection,

### PLAYER

Fired in setClick() and undo() to indicate which player is to play the mext move.

### SEND\_CLICK

Fired in setClick to move a selected piece. Note if the click has been sent from the remote system (i.e. propagate is false) then this will not be sent.

### REMOVE\_HISTORY

Fired in undo() to remove an entry from the history.

### GAME\_END

Fired in changesTurn() if the game has ended.

## Controller to Model

The controller invokes methods directly on the model to pass data to it. Each method that the controller calls is listed below with a brief description of when it occurs.

### newGame()

When the controller is informed a new game should be initiated, e.g. from the UI when New Game is clicked, it will call newGame() on the model.

### setClick()

When the controller is informed a piece has been clicked on the UI, or from a remote instance, the click shall be sent to the model using setClick().

### undo()

When the controller is informed a move should be undone, e.g. from the UI when undo is clicked, it will call undo() on the model.

### export()

When the controller is informed the curremt position should be exported, e.g. from the UI when export is clicked, it will call export() on the model.

## View to Controller

The view invokes methods directly on the controller to pass data to it.

### addView()

Called by a view when it is created to inform the controller it wishes to be notified of updates.

### squareClicked()

Called by a view to indicate a square has been clicked.

### newGame()

Called by a view to indicate a new game has been requested.

### resetBoard()

Called by a view to reset a board back to its initial position.

### connectionFailed()

Called by the remote view to indicate a connection has been lost with the remote game instance.

### error()

Called by a view to indicate an error has occurred. This will lead to a new game being initiated.

### undo()

Called by the view when a user has requested the last move is undone.

### export()

Called by the view when a user has requested the current position be exported.

## Controller to View

Data from the controller to the view(s) is sent as property changes. The controller simply passes the property events raised in the model directly to all registered views, using the IDraughtsView.modelPropertyChange() method. It also generates a NEW\_GAME property change.

### ADD\_HISTORY

Sent to the UI to display the latest move in the move history.

### STATUS (acted on by UI)

A status message that can be displayed on the UI. Note that not all statuses are actually displayed.

### SET\_SELECTED (acted on by UI)

Mark a square as selected. 0 is used to clear any selection.

### ERROR (acted on by UI and RemoteView)

Force the UI to display an error dialog.

### SET\_SQUARE (acted on by UI)

Sets the contents of a square, i.e. either a type of piece or empty.

### SET\_HINTS (acted on by UI)

Sent to indicate which square should be highlighted as possible moves for a selected piece.

### PLAYER (acted on by UI)

Sent to indicate which player is playing. Used in the UI to highlight the current player name.

### NEW\_GAME (acted on by UI)

Sent to the UI to force a new game dialog to be displayed.

### GAME\_DATA\_UPDATED (acted on by UI)

Sent to the UI to indicate there has been a change in the game data that needs to be displayed.

### REMOVE\_HISTORY (acted on by UI)

Sent to the UI to remove the display of the previous move, i.e. undo the last entry on the move history.

### SEND\_CLICK (acted on by RemoteView)

Sent to the UI to propagate a click through to the remote system.

### REMOTE (acted on by RemoteView)

Sent to the UI to ask a remote system to play a game.

### DISCONNECT (acted on by RemoteView)

Sent to the UI to disconnect from the remote system.

# Supporting Documentation

None.

**- End of Document -**