Appendix 3 – Chess board – a custom user interface

Overview

This appendix shows the details of the design and implementation of an application with a custom UI with its specialized view and controller. It introduces several new techniques for creating and controlling cursors, view updating, and other useful tasks.

A.3.1 Chess – a specification

In this and the following sections, we will develop a somewhat limited version of a chess front end, as an example of an application using a non-trivial custom widget. The specification of the problem is the subject of this section.

Specification

This program will be a pilot version of a program allowing two players to play chess. Eventually, the players may be two humans or one human and one computer program but the pilot version is limited to the design and implementation of the user interface for two human players.

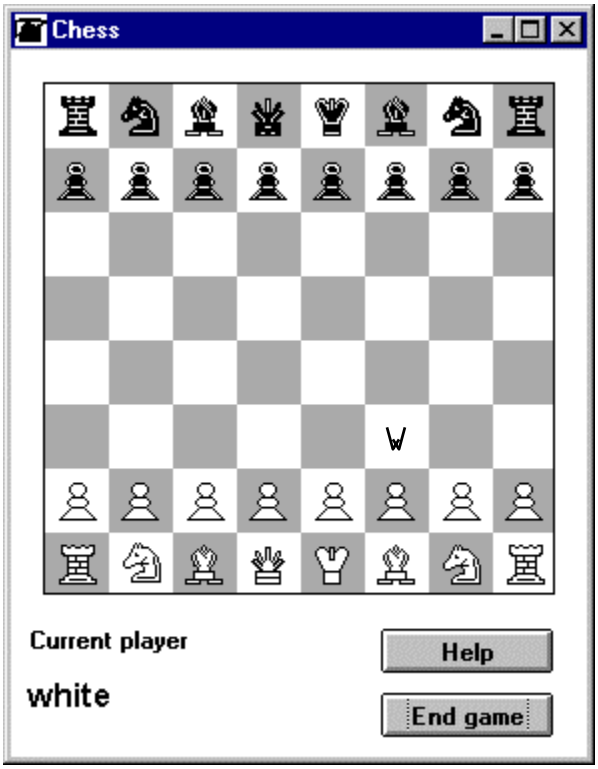

Figure A.3.1. Chess board in starting position and cursor indicating white's turn

Each game begins in the standard configuration shown in Figure A.3.1. A player makes a move by clicking the <select> mouse button over the piece to be moved and then over the desired destination square. Before the first click, the cursor has the shape of a letter indicating which player should move - W for White, B for Black. After the first click, the cursor changes to D (for Destination).

If the move specified by the player is legal, the program displays the move on the chessboard. If the first click identifies an empty square or a square occupied by an opponent's piece, the click is ignored.

If the initial click identifies a legal piece but the destination square is illegal, the attempted move is also ignored. Illegal moves include an attempt to move to a position occupied by one's own piece, moving to a position outside the legal path of the piece or identical to the starting position, and moves along an obscured path (legal knight moves are an exception). A move that exposes one's own king to check is also illegal.

More complicated moves including castling (a move involving a king and a rook), pawn capturing, and pawn replacement upon reaching the opponent's end of the board are not supported. Check mates are recognized only implicitly in that no legal move is possible from a check mate position. Draws are not recognized. When a move results in a check of the opponent's king, the program displays a warning and a red CHECK label. The game ends when a player clicks *End of game*. The *Help* buttons explains the use of the program.

Scenarios.

In keeping with the specification, the term *Player* used in the following scenarios denotes a human player.

Scenario 1: Starting a game between two human players.
1. *Player* executes a statement such as Chess open.
2. *Program* opens board in the state shown in Figure A.3.1, expecting white to make the first move. When the player moves the cursor to the chessboard, it takes the shape of a W indicating that it is the white's turn to play.

Scenario 2. Making a legal move.
1. *Player* clicks one of his or her pieces.
2. *Program* changes cursor to a D, to indicate that the player should click the destination square.
3. *Player* clicks a legal destination square.
4. *Program* redisplays the affected squares.
5. *Program* changes cursor to a W or a B depending on which player moves next.

Scenario 3. Making a legal move resulting in a check.
Same as Scenario 2 but at the end, *Program* displays a dialog warning about the check and the word CHECK in red letters in the game window. The word is removed when the player makes a move that eliminates the check.

Scenario 4. Making an illegal move: First click is an empty square or square occupied by opponent's piece.
1. *Player* clicks an empty square. The shape of the cursor does not change and the click is ignored. The next click is considered to be the first click.

Scenario 5. Making an illegal move: Desired destination is occupied by one's own piece, a piece is blocking the path, the desired move leaves player's king in check, the destination is not on a legal path or is identical to the starting position.
1. *Player* clicks one of his or her pieces.
2. *Program* changes cursor to a D.
3. *Player* clicks a square occupied by his or her own piece.
4. *Program* does not change the chess board and redisplays the initial cursor (B or a W) allowing the player to make a new move.

Scenario 6. End of game by clicking *End of game*.
1. *Player* clicks *End of game*.
2. *Program* displays multiple choice dialog with *Quit*, *New game*, and *Cancel* buttons.
3. *Player* clicks *Quit* and closes the program.

Context Diagram

The major components of the system are the chess board with its pieces and user interface, and the players. The players (including a computer program player) are outside of the scope of the task. The Context Diagram is shown in Figure A.3.2.

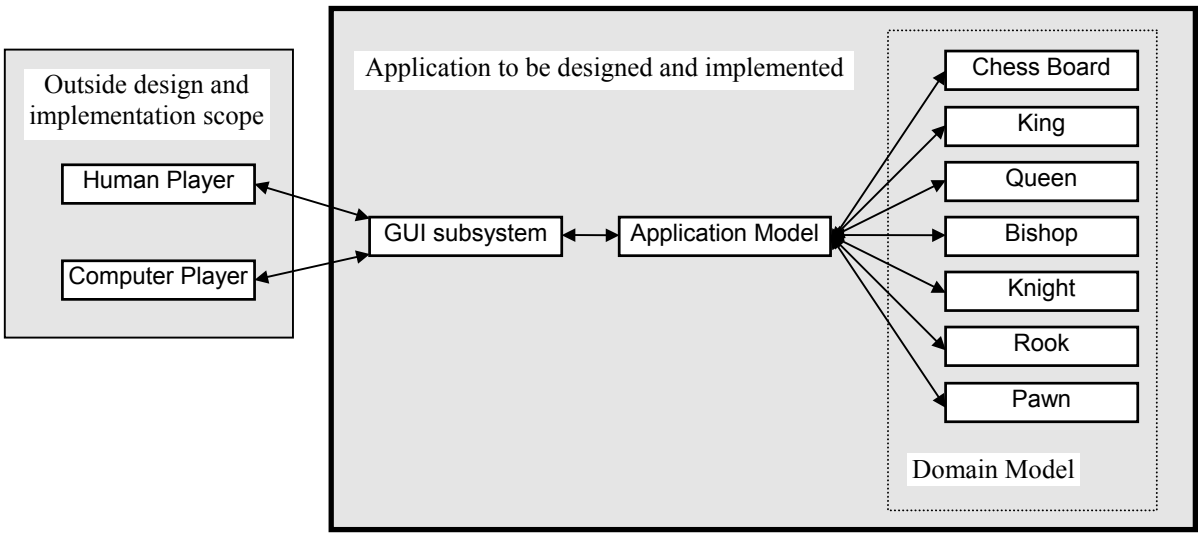

Figure A.3.2. Context Diagram showing components to be designed and components outside the scope of the task.

A.3.2 Preliminary Design

Candidate classes

From the specification, our understanding of chess, and our knowledge of the principles of VisualWorks interfaces we decide on the following candidate classes:

ChessBoard. Domain model of the chess board. Knows the occupant of each square, and which player will move next. Can determine whether a move is legal or not in collaboration with individual pieces.

King. Knows its position and color, helps to determine whether a an attempt to move it is legal. Domain class. Knows its graphical representation.

Queen, Bishop, Knight, Rook, Pawn. Same responsibilities as King but different operation. Domain classes.

ChessGame. Application model defining the window and interfacing the GUI with the domain model.

ChessView. Displays the board and knows how to redraw itself or its parts in response to a new ChessBoard state. A UI class.

ChessController. The controller of ChessView. Captures mouse clicks and converts them to appropriate actions resulting in changes of ChessBoard, and piece positions. Controls cursor image.

Class-level scenarios

After identifying the candidates, we will now examine the scenarios at class-level detail to determine class responsibilities and confirm that we have all the classes we need.

Scenario 1. Starting a game.

1. *Player* executes Chess open.
2. ChessGame initializes its instance variables, opens board in starting state, expecting the first move to be made by white. ChessController displays a W shaped cursor.

Scenario 2. Making a legal move.

1. *Player* clicks one of his or her pieces.
2. ChessController converts cursor coordinates to square row and column, requests confirmation that this is a legal square from ChessBoard, and changes cursor to a D.
3. *Player* clicks a legal destination square.
4. ChessController converts cursor coordinates to square coordinates, asks ChessBoard to confirm and perform the move. ChessBoard performs the necessary tests, asking the selected piece to execute piece-related actions. It asks ChessView to redisplay the current square as empty and display the moved piece in the new position. If the new position was occupied by an opponent's piece, it removes the opponent's piece from the game. ChessGame changes cursor to W or B depending on the player whose turn it is next.

Scenarios 3, 4, and 5 follow the same pattern as Scenario 2 and we leave them as exercises. Scenario 6 does not require any further comments and we can now construct the following preliminary class descriptions:

Preliminary class descriptions

Chess. Application model implementing the main user interface.
Components: chessBoard, chessView

Responsibilities:	*Collaborators*
• opening - start new game and display user interface	ChessBoard, ChessView
	chess pieces
• accessing - chess view	ChessView
• action methods - buttons	
◦ help - display help window	
◦ end of game - display dialog and close if required	

ChessBoard. Domain model of the chess board.
Components: squares (TwoDList), currentPlayer (Symbol)

Responsibilities:	*Collaborators*
• initialization	
• accessing - pieces, view (via dependency)	pieces
• making a move and alternating players	pieces
• testing legality of moves and checks	pieces

ChessController. Captures mouse clicks, controls cursor.
Components: firstClick (Boolean), currentCursor, blackCursor, whiteCursor, destCursor

Responsibilities:	*Collaborators*
• initialization - define cursor shapes and control current cursor	
• events - tracking mouse position and button action	
• control of cursor shape	

ChessView. Display of the chess board.
Components:

Responsibilities:	*Collaborators*
• displaying chessboard with pieces	ChessBoard and pieces
• updating - response to ChessBoard changes via dependency	ChessBoard
• controller access	ChessController

Bishop, King, Knight, Rook, Pawn, Queen - represent individual pieces.
Components: color, position
Responsibilities: *Collaborators*

- testing - is proposed move legal?
- visual representation – image of black and white pieces on both backgrounds

From this specification, we can draw the preliminary Object Model in Figure A.3.3.

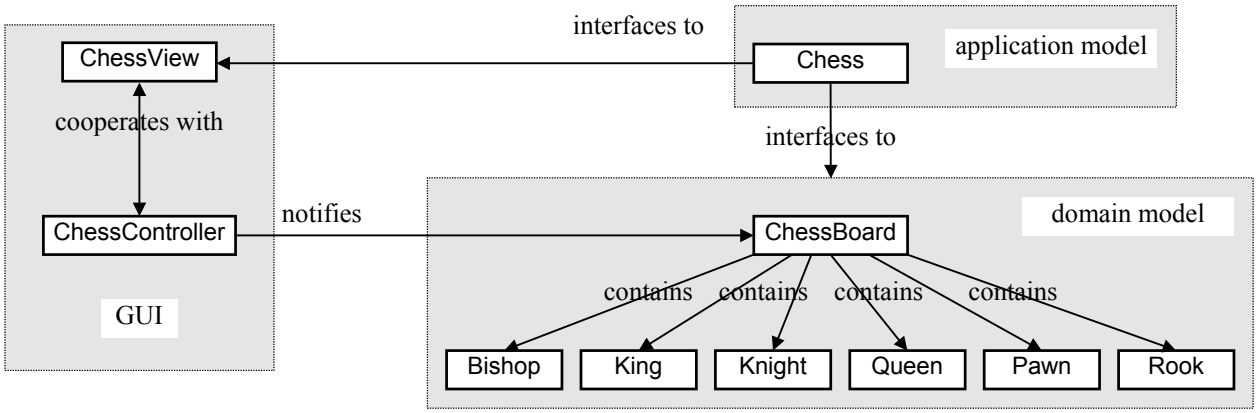

Figure A.3.3. Preliminary Object Model.

A.3.3 Design Refinement

We will now identify abstract classes and the superclass of each class, create a Class Hierarchy diagram, refine responsibilities, and finalize the Object Model.

Abstract classes

The different pieces share a lot of information and behavior: Each has a color and a position on the chess board, all participate in the testing of the legality of a move, and all have their visual representation. We will thus create an abstract class called Piece to define all the common behavior. Class Piece itself is unique and its superclass will thus be Object.

Classes ChessView and ChessController are, of course, in the View and Controller hierarchies. ChessView will be a subclass of View, and ChessController will be a subclass of Controller because we need event notification but no menu.

Class ChessBoard treats ChessView as its dependent and its superclass is thus Model.

Results of this analysis are summarized in the Class Hierarchy diagram in Figure A.3.4 and the following table.

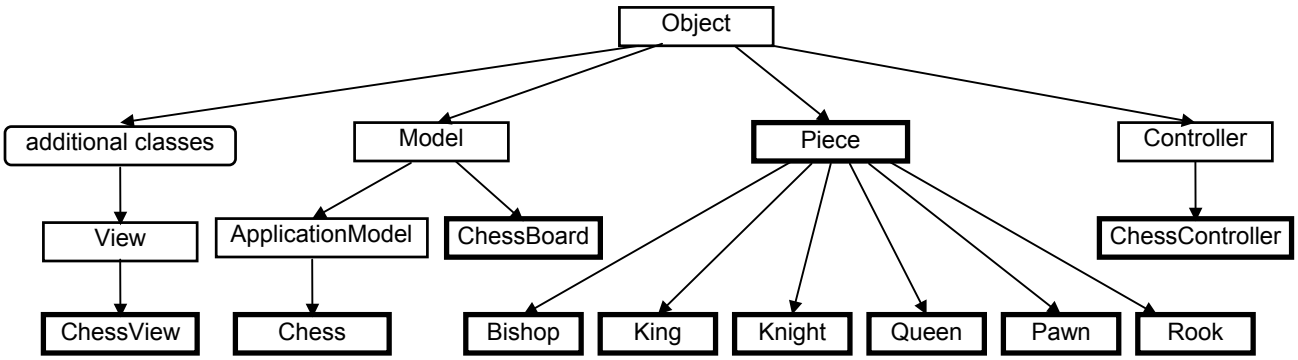

Figure A.3.4. Class Hierarchy. Classes in heavy rectangles are to be implemented.

Class Characteristics:

Class	Type	Superclass	Comment on the choice of superclass
Chess	concrete	ApplicationModel	Mediator between domain model (board and pieces) and GUI.
ChessBoard	concrete	Model	We need dependency to control ChessView.
ChessController	concrete	Controller	Controller provides behaviors such as events, and access to view.
ChessView	concrete	View	View provides persistence and dependency mechanism.
Piece	*abstract*	Object	There is no suitable superclass in the hierarchy.
King	concrete	Piece	Piece was created as an abstraction of individual pieces.
Queen	concrete	Piece	Same comment.
Bishop	concrete	Piece	Same comment.
Knight	concrete	Piece	Same comment.
Rook	concrete	Piece	Same comment.
Pawn	concrete	Piece	Same comment.

Details of selected behaviors

We will now explore and describe, class by class, the more complicated behaviors in order to obtain sufficiently detailed class descriptions suitable for implementation.

Class Chess

The application opens in the usual way and executes the initialization method. Initialization must create a new ChessBoard and a new ChessView, and assign the ChessBoard to the view as its model. Class ChessBoard must be able to modify information such as the name of current player and this requires that the creation method of Chess provide Chess with access to ChessBoard.

The essence of the program is moving pieces. Every move is initiated by a mouse click, mouse clicks are intercepted by the controller, and we will thus continue with the controller.

Class ChessController

The controller responds to button clicks and controls the cursor via methods with predefined event selectors redButtonActivity:, enterEvent:, and exitEvent:. Message redButtonActivity: is sent when the user clicks the red button to select a piece or a destination and its implementation will be as follows:

- Calculate square in row/column coordinates. This is needed to access a piece in ChessBoard.
- If this is the first click of the current player then
    - ask ChessBoard whether the square is a legal starting point
    - remember that the next click will be the second click
    - change cursor shape to D
- If this is not the first click then
    - notify ChessBoard of the destination square
    - ask ChessBoard to make move if it is legal
    - remember that next move will be the first move (even if the move was illegal)
    - if the move was legal, change current player and change cursor to W or B

We will now follow the task of moving a piece by exploring ChessBoard's role in this process. We will return to the controller later.

Class ChessBoard

ChessBoard is responsible for intialization, tests and for triggering the display of the new state.

- Variable squares is initialized to empty squares or to appropriate pieces. The rest is obvious.
- The test of whether the starting position is legal checks that the square contains a piece belonging to the current player. If it does not, the move is illegal.
- Our implementation of the test of the destination and the subsequent move will be as follows:
    - Check if the destination square is legal (empty square or square with opponent's piece)
    - Check if the piece can reach the destination. The move is possible if
        - the piece can reach the destination without considering possible obstructions
        - the path from start to destination is not obstructed (except for the knight)
    - Move piece tentatively to the new position without redrawing the board
    - Check whether the move uncovers current player's king. (To do this, opponent's pieces check whether they can reach the king.) If everything is OK, display new position using dependency of ChessView on ChessBoard, otherwise ignore the move.
    - Is opponent's king in check (player's pieces test whether they can reach opponent's king)? If so, display warning and red label.

To make the operation and design more efficient, we will add the following behaviors and instance variables:

- A method to check whether a move is horizontal, vertical, diagonal, or other ('illegal') because one or more of these tests are required by most pieces. The result is stored in variable kindOfMove.
- Instance variables kings for quick access to the king piece for tests of checks, and blackPieces and whitePieces - again to speed up tests of checks.

Class ChessController

We can now return to ChessController and describe the details of its responsibilities.

- Initialization (occurs automatically when the controller is created): firstClick becomes true, cursor becomes the W cursor because the white player moves first.
- Response to enterEvent which is sent when the cursor enters this controller's view: Remember old cursor (to be displayed when the cursor leaves ChessView) and change cursor to its proper value.
- Response to exitEvent which is sent when the cursor leaves ChessView: Reset cursor to what it was before the cursor entered the chess view.

We conclude that the controller must remember the current and old cursors, and the three cursor images (W, B, and D). To construct them, we will use the *Image Editor* tool, remembering to keep the image to 16@16 pixels, the prescribed Cursor extent.

Class ChessView

Class ChessView is responsible for drawing the chess board at the beginning of a new game, updating it after a legal move, and redrawing it after damage. As we know, this is implemented by displayOn: and update.

Method displayOn: goes through all 64 squares of the chess board. It displays a black or a white square when the square is empty, and if the square contains a piece, asks the piece for the appropriate image and displays it.

Updating is invoked after a move, when ChessBoard issues a changed message. Since each move in this pilot implementation changes only the origin and the destination, we will restrict the response to an update of just those two squares. To be able to do this, we will use update:with: and supply the two squares as an Array argument of the with: keyword.

Class Piece

This class defines shared behaviors and variables of concrete pieces such as King and Queen. The responsibility for testing whether a move can be made is left to concrete subclasses because each concrete subclass uses a different rule. Responsibility for accessing is implemented here.

This completes our design except for the concrete pieces. For a Bishop, legal moves are diagonal. For a Rook, a legal move is either vertical or horizontal. For a Queen, any move that is not classified as 'illegal' is legal; this includes horizontal, vertical, and diagonal moves. Legal moves for Pawn and Knight require special treatment.

The image of a piece is implemented as a class 'resource' method compiled by the Image Editor. These cached images will be used as required.

We will now update our class descriptions, restricting ourselves to a new listing of variables. We don't preclude the possibility that we may have to add auxiliary instance variables during implementation to improve efficiency or to cover any gaps that may still be present.

Updated class descriptions

Class Chess
Instance variables: chessBoard, chessView.

Class ChessBoard
Instance variables: squares, kings, kindOfMove, blackPieces, whitePieces, currentPlayer, oldPosition, newPosition, removedPiece (needed when restoring a tentatively moved piece when the move is illegal).

Class ChessController
Instance variables: firstClick, oldCursor, currentCursor, blackCursor, destCursor, whiteCursor.

Class ChessView
Instance variables: none.

Class Piece
Instance variables: color, position.

Class Bishop and other concrete pieces
Instance variables: none.

Object Model diagram

To complete design, we update the Object Model diagram as in Figure A.3.5.

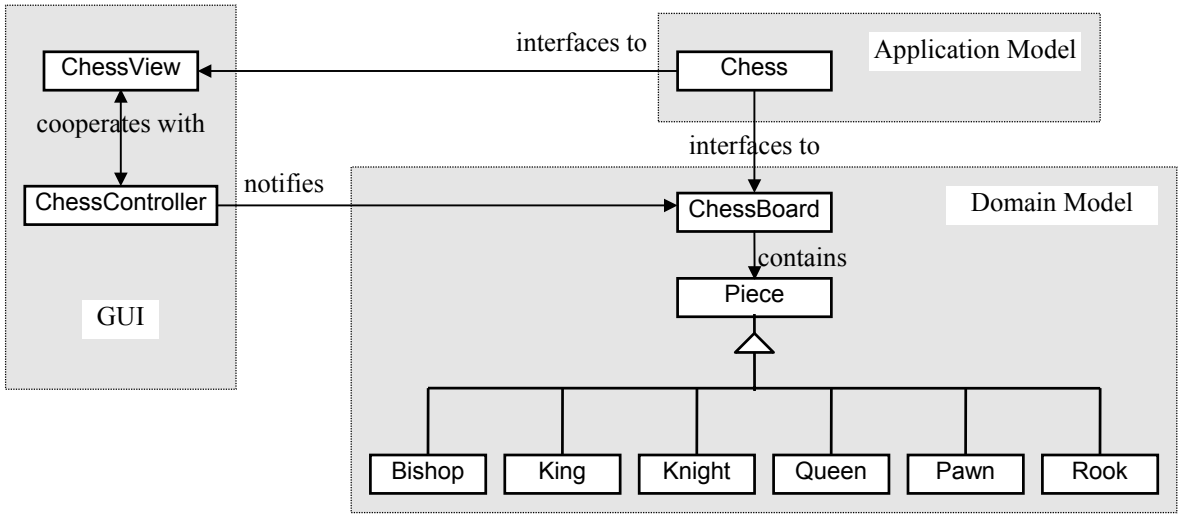

Figure A.3.5. Object Model diagram of chess game classes.

A.3.4 Implementation

Most of the implementation is straightforward and we will limit ourselves to the less trivial methods. The presentation will be organized by class, and classes will be listed alphabetically.

Class Bishop

The creation message for all pieces is color: aColorValue position: aPoint and is defined in class Piece. Besides the resource methods that define bishop's image, the only protocol defined in this class is testing of a legal move. The test checks that the requested move is diagonal as follows:

```
legalMove: kindOfMove to: aPoint
"Return true if this piece can move to this destination."
^kindOfMove == #diagonal
```

where kindOfMove is a Symbol calculated by ChessBoard when the player makes a move. As you can see, its use considerably simplifies the test. The argument aPoint is necessary for pieces such as the knight and since the legalMove:to: message is used polymorphically by all pieces, both arguments must be included even when they are not always needed.

Class Chess

is the application model and its opening message initializes the program as follows:

```
initialize
        chessBoard := ChessBoard newFrom: self. "Let ChessBoard know about myself for accessing."
        chessView := ChessView new initialize.
        chessView model: chessBoard
```

After a player makes a move, ChessBoard switches players and sends the following message to change the player label in the window.

```
player: aPlayer
"Players switched. Assign player and display color label on the UI, hiding the other label."
        | otherPlayer |
```

```
otherPlayer := aPlayer == #white
                                ifTrue: [#black]
                                ifFalse: [#white].
(self builder componentAt: otherPlayer) beInvisible.
(self builder componentAt: aPlayer) beVisible
```

The operation is based on our decision to paint all possible labels on the canvas and display and hide them selectively as needed.

Class ChessBoard

Holds board information and needs to know about Chess to notify it to update display after a move has been made.

```
initializeFrom: app
"Initialize everything for the first game."
        appModel := app.
        self newGame
```

The lengthy code that initializes the squares and creates pieces is taken out into the following methods:

```
newGame
"Initialize square information and create white and black pieces."
        self initializeSquares.
        self initializePieces.
        currentPlayer := #white
```

Method newGame: is reused as a response to the selection *New game* in the closing dialog of the game. The details of initialization are as follows:

```
initializeSquares
“Create holder for board description and assign a piece to each square.”
        | pos |
        squares := TwoDList columns: 8 rows: 8.
        1 to: 8 do: [:col | "Blank empty squares."
                1 to: 8 do: [:row | squares at: row @ col put: nil]].
        1 to: 8 do: [:col | "Place remaining pieces."
                "Place pawns."
                squares at: (pos := col @ 2) put: (Pawn color: #black position: pos).
                squares at: (pos := col @ 7) put: (Pawn color: #white position: pos)].
                "Place all remaining pieces."
                squares at: (pos := 1 @ 1) put: (Rook color: #black position: pos).
                squares at: (pos := 1 @ 8) put: (Rook color: #white position: pos).
                squares at: (pos := 8 @ 1) put: (Rook color: #black position: pos).
                squares at: (pos := 8 @ 8) put: (Rook color: #white position: pos).
                squares at: (pos := 2 @ 1) put: (Knight color: #black position: pos).
                squares at: (pos := 2 @ 8) put: (Knight color: #white position: pos).
                squares at: (pos := 7 @ 1) put: (Knight color: #black position: pos).
                squares at: (pos := 7 @ 8) put: (Knight color: #white position: pos).
                squares at: (pos := 3 @ 1) put: (Bishop color: #black position: pos).
                squares at: (pos := 3 @ 8) put: (Bishop color: #white position: pos).
                squares at: (pos := 6 @ 1) put: (Bishop color: #black position: pos).
                squares at: (pos := 6 @ 8) put: (Bishop color: #white position: pos).
                squares at: (pos := 4 @ 1) put: (Queen color: #black position: pos).
                squares at: (pos := 4 @ 8) put: (Queen color: #white position: pos).
                squares at: (pos := 5 @ 1) put: (King color: #black position: pos).
                squares at: (pos := 5 @ 8) put: (King color: #white position: pos)
```

Auxiliary variables holding white pieces, black pieces, and kings are initialized as follows:

initializePieces

```
“Initialize variables holding black pieces, white pieces, and kings using information about squares.”
        whitePieces := OrderedCollection new: 16.
        blackPieces := OrderedCollection new: 16.
        kings := Array new: 2.
        squares do: [:piece | piece isNil
                        ifFalse: [piece color == #white
                                ifTrue:   [whitePieces add: piece.
                                        piece isKing ifTrue: [kings at: 1 put: piece]]
                                ifFalse:  [blackPieces add: piece.
                                        piece isKing ifTrue: [kings at: 2 put: piece]]]]
```

The essence of the program is the handling of individual moves and besides the necessary manipulation of the user interface, this requires various checks. The method is implemented as follows:

move

```
"Attempt to make a move and return true if the move is legal. Do nothing and return false otherwise.”
        | piece destPiece |
        piece := squares at: oldPosition. “Piece at first click position.”
        “Move is legal only if destination is empty or occupied by opponent.”
        ((destPiece := squares at: newPosition) isNil or: [destPiece color ~~ currentPlayer])
                ifFalse: [^false].
        “Can this piece go to the specified destination?”
        (self piece: piece canReach: newPosition from: oldPosition)
                ifFalse: [^false].
        “Make the move tentatively.”
        self move: piece to: newPosition.
        “Does the move leave the player’s king exposed? If so, retract.”
        self inCheck ifTrue: [self restore: piece. ^false].
        (appModel builder componentAt: #check) beInvisible.
        “Move is OK, get it displayed via dependency.”
        self  changed: #position with: (Array with: oldPosition with: newPosition).
        “Does the move put opponent’s king in check? Display red label if true.”
        self checks ifTrue: [(appModel builder componentAt: #check) beVisible].
        “Move completed, get ready for next player’s move.”
        self switchPlayer.
        ^true
```

Although this method works correctly, it’s design is unsatisfactory. In particular, ChessBoard is a domain object and its implementation should be independent of the user interface. Yet, this method directly accesses a specific component of the user interface – the check label. If the interface changed, this reference would become invalid, and if we wanted to use ChessBoard in another application, we couldn’t. The proper way for a domain object to communicate with the UI is by notifying the application model of the change that occurred and leave the implementation details to it. We leave this modification as an exercise.

As you have noticed, the move method performs a variety of tests and some of them were delegated to the following auxiliary methods:

piece: aPiece canReach: point2 from: point1

```
"Can aPiece go from point1 to point2?"
        “Determine the kind of move (such as horizontal or vertical) being attempted.”
        self kindOfMoveFrom: point1 to: point2.
        “Ask the piece if this move is OK.
        (aPiece legalMove: kindOfMove to: point2)
                ifFalse: [^false].
        “Move is OK but is the path clear?”
        ^self isClearFrom: point1 to: point2
```

In here, method legalMoveFrom:to: checks whether aPiece can, in principle, move from point1 to point2. It does not check whether the path is clear. The method is defined differently for each piece because

it depends on how the piece moves. We saw how Bishop treats legalMoveFrom:to: and other pieces define it similarly. Message isClearFrom:to: is sent only after confirmation that the move is, in principle, legal; it checks whether the path between the two points is clear and its definition is as follows:

```
isClearFrom: point1 to: point2
"Return true if the path from point1 to point2 is clear. Delegate the problem to specialized direction-dependent methods."
        kindOfMove == #horizontal ifTrue: [^self horizontalClearFrom: point1 to: point2].
        kindOfMove == #vertical ifTrue: [^self verticalClearFrom: point1 to: point2].
        kindOfMove == #diagonal ifTrue: [^self diagonalClearFrom: point1 to: point2].
        ^true "Knight does not care about obstructions."
```

The method uses specialized methods that test a horizontal path, a vertical path, and a diagonal path. We could reduce the number of tests somewhat but the definition would be less readable and we thus leave it in this form. The following is an implementation of one of these test methods:

```
horizontalClearFrom: point1 to: point2
"Return true if the horizontal line between the two points is clear."
        | x1 x2 y step |
        x1 := point1 x.
        x2 := point2 x.
        y := point1 y.
        x1 > x2   ifTrue: [step := -1]
                  ifFalse: [step := 1].
        x1 + step to: x2 - step by: step do: [:x | (squares at: x @ y) isNil ifFalse: [^false]].
        ^true
```

The method that determines the kind of move is defined as follows:

```
kindOfMoveFrom: point1 to: point2
"Is the move horizontal, vertical, diagonal, or 'illegal'? Return appropriate symbol."
        kindOfMove := point1 y = point2 y
                        ifTrue: [#horizontal]
                        ifFalse: [point1 x = point2 x
                                  ifTrue: [#vertical]
                                  ifFalse: [(point1 x - point2 x) abs = (point1 y - point2 y) abs
                                                ifTrue: [#diagonal]
                                                ifFalse: [#illegal]]]
```

The chessboard knows about all pieces and so we leave it in charge of king checks. One such check requires finding whether a piece in a new position checks the opponent's king, the other checks whether it uncovers the player's own king to check. Both tests work on the same principle. As an example, the test to determine whether a move leaves the player's king in check is defined as follows:

```
inCheck
"Test whether the king of the current player is in check by asking each opponent piece. Return true if check."
        | position |
        position := self myKing position.
        "Ask each of the opponent's pieces if it can reach the king."
        self opponentPieces do: [:piece | (self
                        piece: piece
                        canReach: position
                        from: piece position)
                                  ifTrue: [^true]].
        ^false
```

Finally, the following method restores the chess board to its original position if we must take back a tentative move that resulted in a check to one's own king:

restore: aPiece
"Attempted move is illegal because it exposes my king to check. Restore original state of board domain model. No need to redraw because we have not drawn anything yet."

```
"Move piece to its original position."
squares at: oldPosition put: aPiece.
"Restore taken piece if any."
removedPiece isNil ifFalse: [self opponentPieces add: aPiece].
squares at: aPiece position put: removedPiece.
aPiece position: oldPosition
```

Class ChessController

This class is in charge of user interaction for the chess view. Its initialization method prepares the controller for the first click and defines the three cursor shapes:

```
initialize
    firstClick := true.
    whiteCursor := Cursor      image: ChessController aWhiteCursor asImage
                               mask: ChessController aWhiteCursorMask asImage
                               hotSpot: 4 @ 8
                               name: 'white moves'.
    blackCursor := Cursor      image: ChessController aBlackCursor asImage
                               mask: ChessController aBlackCursorMask asImage
                               hotSpot: 4 @ 8
                               name: 'black moves'.
    destCursor := Cursor       image: ChessController aDestCursor asImage
                               mask: ChessController aDestCursorMask asImage
                               hotSpot: 4 @ 8
                               name: 'destination'.
    currentCursor := whiteCursor
```

The 'resource' class methods aWhiteCursor, aBlackCursor, and aDestCursor supply the cursor. They were generated automatically by painting the image and mask shapes by the *Image Editor*. Note that the image and the mask are stored by two different commands (Figure A.3.6).

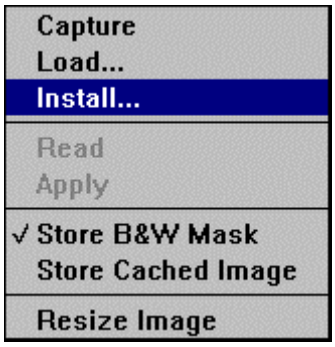

Figure A.3.6. Image Editor allows installing a shape as a cached image or as a mask.

Cursor control methods are implemented by the predefined event selectors as follows:

enterEvent: event
"Remember cursor on entry and switch to the cursor appropriate for the current state."

```
oldCursor := Cursor currentCursor.
Cursor currentCursor: currentCursor.
currentCursor show
```

exitEvent: event
"Return to the saved cursor."

```
Cursor currentCursor: oldCursor.
Cursor normal show
```

The calculation of currentCursor occurs when the mouse button is clicked and is explained below.

The activity associated with pressing the <select> button when the player selects a piece or a destination again uses a predefined event selector. It is defined as follows:

```
redButtonPressedEvent: event
"User clicked <select> button over a square. Relate this to the move now in progress."
        | square |
        "Convert cursor coordinates to row and column numbers."
        square := ((self sensor cursorPointFor: event) / self view squareExtent) truncated + (1 @ 1).
        firstClick ifTrue: "First click selects piece. Check legality, notify ChessBoard, change cursor."
                        [(self model isSquareLegal: square)
                                ifFalse: [^self].
                        self model oldPosition: square. "Remember old position to be able to restore it."
                        firstClick := false.
                        currentCursor := destCursor]
                ifFalse:  "Second click selects destination. Check, make move, change cursor."
                        [self model newPosition: square.
                        self model move.
                        firstClick := true.
                        currentCursor := self model currentPlayer == #white
                                ifTrue: [whiteCursor]
                                ifFalse: [blackCursor]].
        "Ask new cursor to display itself."
        currentCursor show
```

Class ChessView

The main responsibility of this class is, of course, displaying. The displayOn: method is automatically executed for every damage and change, and during opening. It must be able to redraw the whole board and we implement it by enumerating over all squares, displaying squares for the empty ones, and asking occupying pieces to supply their image for the given background (explained later). The definition is as follows:

```
displayOn: aGraphicsContext
"Draw the board for the current configuration of pieces."
squareExtent := self squareExtent.  "Get the extent of a square in pixels."
1 to: 8 do: [:row | 1 to: 8
        do:     [:column |
                | background piece square |
                "Calculate view coordinates of square from its row and column numbers."
                square := column - 1 * squareExtent  @ (row - 1 * squareExtent ).
                "Calculate background color."
                background := (row + column) even
                        ifTrue: [#white]
                        ifFalse: [#lightGray].
                piece := self model squares at: column @ row.          "Piece on current square, if any."
                piece isNil
                        ifTrue:  "The square is empty."
                                [aGraphicsContext paint: (ColorValue perform: background);
                                displayRectangle: (0 @ 0 extent: squareExtent  @ squareExtent )
                                                at: square]
                        ifFalse:  "The square contains a piece. Get its image from the piece."
                                [aGraphicsContext displayImage:
                                        (piece imageForPlayer: piece color
                                        onBackground: background)
                                        at: square]]]
```

Note that we added a new instance variable called squareExtent to hold the size of a square. Its value will be useful for updating as well and Chess calculates it from the actual window size using lazy evaluation because during initialization, the chess board view has not yet been built. Alternatively, we could do the calculation in postBuildWith:.

Another responsibility of the view is response to update requests. The update method is executed whenever ChessBoard (the model of ChessView) executes the changed message which happens when it accepts a legal move. We use the update:with: form because the model (a ChessBoard) must supply the two Rectangle objects corresponding to the squares that are to be redrawn. The implementation is as follows:

```
update: aSymbol with: anArray
"The model has changed. Invalidate the affected squares."
        | square |
        1 to: 2 do: [:i |       square := anArray at: i.
                                self invalidateRectangle: (square - 1 * squareExtent  extent: squareExtent)]
```

Note that we restrict invalidation to the two affected squares. Although this does not speed up redrawing, it eliminates flashing which would occur if the whole board was redrawn.

Class Knight

Like all other pieces, Knight only defines its shape and a test for a legal move. The definition is as follows:

```
legalMove: kindOfMove to: aPoint
"Return true if this piece can move to aPoint. Two vertical steps and one horizontal step or vice versa are legal."
        | absX absY |
        absX := (position x - aPoint x) abs.
        absY := (position y - aPoint y) abs.
        kindOfMove == #illegal      "#horizontal, #vertical, and #diagonal, are disallowed for kinghts."
        ifTrue: [^(absX = 2 and: [absY = 1])
                                or: [absX = 1 and: [absY = 2]]].
        ^false
```

Class Piece

This class is used mainly for shared variables and does not define any interesting methods, except for the creation method

```
color: aSymbol position: aPoint
        ^(self new) color: aSymbol; position: aPoint
```

calculation of the piece image to display for the current player

```
imageForPlayer: symbol1 onBackground: symbol2
        ^(self class imageForPlayer: symbol1 onBackground: symbol2) asImage
```

where

```
imageForPlayer: symbol1 onBackground: symbol2
"Retrieve the appropriate image for this piece and for the current player."
^self perform: (symbol1 == #white
                        ifTrue: [symbol2 == #white
                                        ifTrue: [#whiteOnWhite]
                                        ifFalse: [#whiteOnBlack]]
                        ifFalse: [symbol2 == #white
                                        ifTrue: [#blackOnWhite]
                                        ifFalse: [#blackOnBlack]])
```

and method isKing which defines every piece *not* to be a King by default. (Method isKing is used to assemble the array of kings.) The definition is listed below and only class King redefines it to return true. This is a typical trick used to eliminate multiple definitions when only one class has a different implementation, and method isNil is an example of a library method defined in this way.

```
isKing
        ^false
```

These selected methods should give you a good idea of our implementation and we leave the rest and improvements to you as exercises.

Main lessons learned:

- A cursor may be constructed with the Image Editor. Remember that cursor size is limited to 16@16 and that the mask shape must be stored as a mask.
- Redrawing should always be restricted to a minimum to minimize flashing. This can be achieved by using invalidateRectangle: instead of invalidate. In some situations, one can also design displayOn: to redraw only the required part of the view.
- Smalltalk normally leaves damage repair by invalidation until the processor has nothing more important to do (lazy invalidation). To redisplay without delay, use invalidateRectangle:repairNow:.

Exercises

1. Complete and test the implementation.
2. Redesign the application by introducing a new class called Move with components kindOfMove (a Symbol) and startSquare and endSquare (both Points). Consider implementing squares as instances of BoardSquare capable of displaying themselves and knowing their chess piece.
3. Implement all rules of chess such as castling, taking en passant, and so on. Does this extension require a major change in the original design?
4. We have seen that illegal situations such as division by zero are generally handled by the exception mechanism. Give a detailed description of how illegal moves could be handled as exceptions.
5. Remove the direct coupling between the domain model ChessBoard and the user interface.
6. Moving a piece takes a noticeable time in the current implementation. When you analyze what is happening, you will find that although we limit display to only two squares (using invalidateRectangle:), the whole displayOn: method must still be executed and all 64 squares recalculated. It seems that performance could be improved by making sure that only the two affected pieces are re-evaluated. Estimate how much this modification could speed up redisplay and implement it.
7. Measure how long it takes to draw the interface for various drawing strategies including invalidate, invalidateRectangle:, and the improvement suggested in the previous exercise. Compare the seriousness of flashing of the various approaches.
8. How would our implementation have to be modified if we wanted to use it as the interface of a computer player?
9. Improve the game by allowing players to initiate an automatic log of their moves. Allow the game log to be printed or replayed under the control of an additional button. Keep names and scores of players.
10. Enforce a time limit on each move. When a move is not completed within the limit, the player automatically loses the game.
11. Write a series of simple computer chess player programs starting with a program that makes random moves.
12. Use or modify suitable parts of the chess program to answer the following question:
    a. Which positions can be reached by a knight from a given starting position in exactly three moves?
    b. Which positions can be reach by a knight from a given starting position in at most three moves?

c. How many moves are required for a knight to reach a given square from a given starting square?
d. What is the maximum number of moves for a knight to reach any position on the board from a given starting square?
e. Find a configuration of eight queens such that none can reach any other in one move. This problem is called the problem of eight queens.
f. Find all queen positions for the previous exercise.

Conclusion

In this appendix, we presented the complete development of an application with a custom user interface component. The notable features were the creation of custom cursors and handling of input events. We also used view invalidation restricted to a small part of the view and mentioned that although this does not speed up display, it minimizes flashing and thus improves the quality of the user interface.