

# 12

## A Simple Graphical Application

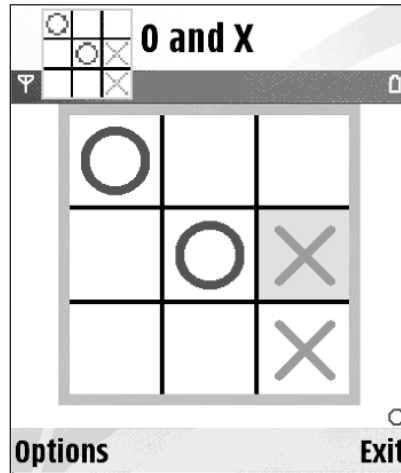
After a series of chapters describing individual aspects of Symbian OS, it is time to pull a few threads together. In Chapter 11, we described a minimal GUI application that simply showed how the UI framework and the application UI fit together. Now we look at a complete, simple, but non-trivial example application that forms a basis for the examples used in several of the following chapters.

In this chapter, we cover some aspects of application design and introduce the idea of how to program a user interface that does not assume a particular screen size. This latter point is important if you want to target your application at multiple user interfaces, as it helps speed up your porting efforts. In addition, we show how an application can use the principles described in Chapter 7 to save and restore its persistent data.

Before looking at the application's structure and behavior in more detail, we start with a general introduction to the Noughts and Crosses (otherwise known as Tic-Tac-Toe) application. Figure 12.1 shows the application running on a phone using the S60 user interface.

This application runs a game of noughts and crosses in which two players take turns to make their moves. The object of the game is to place three noughts (or crosses) in any row, column or diagonal before your opponent manages to do so.

We've implemented the game in a way that minimizes the differences needed to run on phones using the S60 and UIQ user interfaces. Depending on the nature of the phone it is running on, you can make a move either by using a combination of the cursor and pushbutton keys or by tapping on the screen.



**Figure 12.1** The Noughts and Crosses application

We've also kept the logic relating to the game itself very simple so that the bulk of the code illustrates issues of a general nature, relevant to most applications. In particular, we've designed the layout of the display in a way that can easily be adapted to show a rectangular grid with any number of tiles and to display on phones with a variety of screen formats. The grid layout is defined by a set of constants in `oandxdefs.h`:

```
static const TInt KTilesPerRow = 3;
static const TInt KTilesPerCol = 3;
static const TInt KNumberOfTiles = (KTilesPerRow*KTilesPerCol);
```

All the application's layout and drawing code is based on these values. By changing them, the layout can be adapted for games using differently sized and shaped boards.

The same header file also defines values that are specific to the Noughts and Crosses game, including the constant `KTilesPerSide` (which, unlike the earlier values, assumes that the board layout is square) and the states that can be associated with each tile:

```
static const TInt KTilesPerSide = 3;

enum TTileState
{
    ETileBlank = 0,
    ETileNought = 1,
    ETileCross = 4
};
```

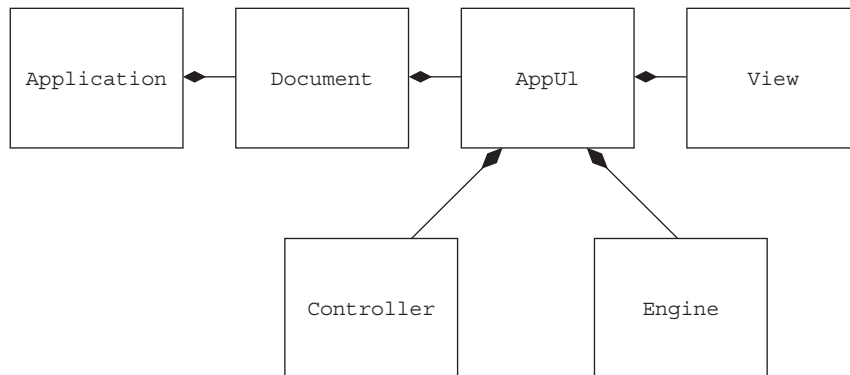
The state values are chosen so that `ETileCross` is greater than the product of `ETileNought` with either `KTilesPerRow` or `KTilesPerCol`, which helps with the calculation of a winner.

This file also defines the height of the application's status window, used to show whose turn it is to play:

```
static const TInt KStatusWinHeight = 20; // in pixels
```

In Chapter 11 you saw that the simplest Symbian OS graphical application needs instances of four classes: the application, the document, the application UI and a view. For this game, we've added two more classes: a controller and a model, or engine. Since the game control logic is so simple, we could have implemented it entirely within the application UI class, rather than defining separate controller and engine classes. However, we've chosen to add these classes from the start, in preparation for the more complex form of the game that is developed in Chapter 20.

The overall structure of the game is shown in Figure 12.2.



**Figure 12.2** Structure of the Noughts and Crosses application

We cover the S60 version of the game first and then look at the changes that are needed to create the UIQ version.

## 12.1 Implementing the Game on S60

### The Application Class

The S60 class definition for the Noughts and Crosses application class is listed below. As you can see, it is very similar to the corresponding class in Chapter 11's basic application: it implements `AppDllUid()`, to report the application's UID to the application framework, and `CreateDocumentL()`.

```

class COandXApplication : public CAknApplication
{
public:
    // From CApaApplication
    TUid AppDllUid() const;

protected:
    // From CEikApplication
    CApaDocument* CreateDocumentL();
};

```

This is fairly typical for applications. It is rare for an application class to override any of the other application class member functions.

## The Document Class

The class definition for the Noughts and Crosses document class, listed below, provides the implementation of `CreateAppUiL()`, which is always required. In addition, it implements persistent storage of the game's state in the `StoreL()` and `RestoreL()` functions, using the principles described in Chapter 7.

```

class COandXDocument : public CAknDocument
{
public:
    static COandXDocument* NewL(CEikApplication& aApp);
    virtual ~COandXDocument();

    // From CEikDocument
    CEikAppUi* CreateAppUiL();
    void StoreL(CStreamStore& aStore, CStreamDictionary& aStreamDic) const;
    void RestoreL(const CStreamStore& aStore,
                 const CStreamDictionary& aStreamDic);
    CFileStore* OpenFileL(TBool aDoOpen, const TDesC& aFilename, RFs& aFs);
private:
    COandXDocument(CEikApplication& aApp);
private:
    COandXAppUi* iAppUi;
};

```

The implementations of `StoreL()` and `RestoreL()` delegate to the application UI class the actual task of storing and restoring the application's data, but they retain the responsibility for associating the application's UID with the ID of the stream in which the data is stored.

```

void COandXDocument::StoreL(CStreamStore& aStore,
                           CStreamDictionary& aStreamDic) const
{
    TStreamId id = iAppUi->StoreL(aStore);
}

```

```

    aStreamDic.AssignL(KUIdOandXApp, id);
}

void COandXDocument::RestoreL(const CStreamStore& aStore,
                             const CStreamDictionary& aStreamDic)
{
    TStreamId id = aStreamDic.At(KUIdOandXApp);
    iAppUi->RestoreL(aStore, id);
}

```

The application framework provides support for an application to save its document data in an application-specific direct file store and, in UIQ, it is sufficient just to implement the two functions listed above.

In contrast, S60 applications are assumed not to be document-based and the default behavior is that the application framework never calls these two functions. In order to restore the support, you need to override the `OpenFileL()` function to call `CEikDocument`'s implementation of `OpenFileL()`:

```

CFileStore* COandXDocument::OpenFileL(TBool aDoOpen,
                                       const TDesC& aFilename, RFs& aFs)
{
    return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
}

```

## The Application UI Class

The application UI class normally forms the core of an application, and the `COandXAppUi` class is no exception. In addition to its standard role of handling much of the interaction with the user, `COandXAppUi` owns the application's view, controller and engine instances and supplies the central logic for saving and restoring the application's persistent data.

```

class COandXAppUi : public CAknAppUi
{
public:
    COandXAppUi();
    virtual ~COandXAppUi();

    // New functions
    void ReportWhoseTurn();
    void ReportWinnerL(TInt aWinner);

    // From CEikAppUi, for persistent data
    TStreamId StoreL(CStreamStore& aStore) const;
    void RestoreL(const CStreamStore& aStore, TStreamId aStreamId);
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);

private:
    // From MEikMenuObserver

```

```

void DynInitMenuPanel(TInt aResourceId, CEikMenuPane* aMenuPane);
// From CEikAppUi
void HandleCommandL(TInt aCommand);
void ConstructL();

public:
    // AppUi owns the engine, controller and application view.
    COandXEngine* iEngine;
    COandXController* iController;
private:
    COandXAppView* iAppView;
    // Set if the application view was added to the control stack.
    TBool iStacked;
};

```

As you can see from the above class definition, part of the user interaction consists of reporting significant events such as whose turn it is and, at the end of a game, who (if anyone) has won. In the case of the S60 version, the application UI class also handles all aspects of using menu commands, including dynamically setting the menu content according to the current state of the game. We have more to say about using menus later in this chapter.

The second-phase constructor and the destructor make it clear that the application UI class owns the engine, the controller and the view, which are all created in a leave-safe way:

```

void COandXAppUi::ConstructL()
{
    BaseConstructL(EAknableSkin);
    iEngine = COandXEngine::NewL();
    iController = COandXController::NewL();
    iAppView = COandXAppView::NewL(ClientRect());
    AddToStackL(iAppView); // Enable keypresses to the view
    iStacked = ETrue;
    ReportWhoseTurn();
}

COandXAppUi::~COandXAppUi()
{
    if (iStacked)
    {
        RemoveFromStack(iAppView);
    }
    delete iAppView;
    delete iController;
    delete iEngine;
}

```

It is worth noting that, it is the responsibility of the application UI class to ensure that the view is added to the control stack, via a call to `AddToStackL()`, to ensure that it has the opportunity to receive and process keypresses. Since this call can fail, it is necessary to set a flag if

it succeeds, so that the flag can be tested (in this case, since there is only one view, in the destructor) before the view is removed from the control stack.

This application uses two different ways to report events to the user. Reporting whose turn it is to play, which happens between every move in the game, is delegated to the view:

```
void COandXAppUi::ReportWhoseTurn()
{
    iAppView->ShowTurn();
}
```

If either player wins, which obviously happens no more than once per game, the application UI class reports the winner by means of a UI-specific dialog that displays the appropriate text, read from the application's resource file. The S60 version uses an information note:

```
void COandXAppUi::ReportWinnerL(TInt aWinner)
{
    TBuf<MaxInfoNoteTextLen> text;
    iEikonEnv->ReadResource(text, aWinner==ETileCross
        ? R_OANDX_X_WINS : R_OANDX_O_WINS);
    CAknInformationNote* infoNote = new (ELeave) CAknInformationNote;
    infoNote->ExecuteLD(text);
}
```

The application's persistent data is saved and restored via the `StoreL()` and `RestoreL()` functions of the application UI class which, you may recall, are called from the application's document class.

```
TStreamId COandXAppUi::StoreL(CStreamStore& aStore) const
{
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(aStore);
    stream << *this; // alternatively, use ExternalizeL(stream)
    stream.CommitL();
    CleanupStack::PopAndDestroy();
    return id;
}

void COandXAppUi::RestoreL(const CStreamStore& aStore,
                          TStreamId aStreamId)
{
    RStoreReadStream stream;
    stream.OpenLC(aStore, aStreamId);
    stream >> *this; // alternatively use InternalizeL(stream)
    CleanupStack::PopAndDestroy();
}
```

`Store()` and `Restore()`, respectively, call the `ExternalizeL()` and `InternalizeL()` functions of the application UI class via the

<< and >> operators. As you can see from the following listing, the application UI class itself has no persistent data. The functions deal directly with the view's data, which is simply the ID of the control that currently has focus, and leave the engine and the controller to take care of their own data.

```
void COandXAppUi::ExternalizeL(RWriteStream& aStream) const
{
    iEngine->ExternalizeL(aStream);
    iController->ExternalizeL(aStream);
    aStream.WriteInt8L(iAppView->IdOfFocusControl());
}

void COandXAppUi::InternalizeL(RReadStream& aStream)
{
    iEngine->InternalizeL(aStream);
    iController->InternalizeL(aStream);
    ReportWhoseTurn();
    iAppView->MoveFocusTo(aStream.ReadInt8L());
}
```

The S60 application UI class is also responsible for handling menu commands, via the `DynInitMenuPaneL()` and `HandleCommandL()` functions. We discuss them – and the differences between the ways they are handled in S60 and UIQ – later in this chapter.

Finally, it is worth pointing out that we provide access to the controller and engine via global static functions, declared (in `oandxappui.h`) as:

```
GLREF_C COandXAppUi* OandXAppUi();
GLREF_C COandXController& Controller();
GLREF_C COandXEngine& Engine();
```

The implementations of these functions are:

```
GLDEF_C COandXAppUi* OandXAppUi()
{
    return static_cast<COandXAppUi*>(CEikonEnv::Static()->AppUi());
}

GLDEF_C COandXController& Controller()
{
    return *OandXAppUi()->iController;
}

GLDEF_C COandXEngine& Engine()
{
    return *OandXAppUi()->iEngine;
}
```

We have chosen to use global functions in this example because they accurately represent the intention that these functions should be

accessible from anywhere in the program. Furthermore, they do not need verbose source text to call them, which is useful in example code. We would not use them in production code, since there is no control of when and from where they are called. Additionally, calling a global function involves accessing thread-local storage, which is slow compared with other function calls. We could improve the efficiency by implementing them as static members of a class, but this relies on the availability of writable static data and so would not work on all phones. The best solution would be to implement them as normal member functions of, for example, the application UI class and pass a (non-owning) reference to the constructor of each class that needs to access them. We have not done that in this example because it would add a significant amount of incidental complexity to each of the constructors concerned.

## The Controller Class

The controller class, whose definition is listed below, is responsible for managing the state of the game and its logic, including updating the data in the engine class.

```
class COandXController : public CBase
{
public:
    static COandXController* NewL();
    virtual ~COandXController();
enum TState
{
    ENewGame, EPlaying, EFinished
};
// state
inline TBool IsNewGame() const;
// game control
void Reset();
// stream persistence
void ExternalizeL(RWriteStream& aStream) const;
void InternalizeL(RReadStream& aStream);

TBool HitSquareL(TInt aIndex);
TBool IsCrossTurn() const;
void SwitchTurn();
private:
    void ConstructL();
private: // private persistent state
    TState iState;
    TBool iCrossTurn;
};
```

In this stand-alone, non-communicating, version, the game state is simply one of the three values `ENewGame`, `EPlaying` or `EFinished`, together with a flag that indicates which player is next to play. The only externally accessible state-query functions needed are `IsCrossTurn()`,

which simply returns the value of `iCrossTurn`, and `IsNewGame()`, coded as:

```
inline TBool COandXController::IsNewGame() const
{ return iState==ENewGame; }
```

The central logic of the controller is contained in the `HitSquareL()` function:

```
TBool COandXController::HitSquareL(TInt aIndex)
{
    if (iState == EFinished)
    {
        return EFalse;
    }
    if (iState == ENewGame)
    {
        iState = EPlaying;
    }
    if (Engine().TryMakeMove(aIndex, IsCrossTurn()))
    {
        SwitchTurn();
        TInt winner = Engine().GameWonBy();
        if (winner)
        {
            iState = EFinished;
            OandXAppUi()->ReportWinnerL(winner);
        }
        return ETrue;
    }
    return EFalse;
}
```

This function is called from the application's board view whenever a player attempts to make a move. It disallows the move if the game state is `EFinished`; if this is the first move in the game (i.e. the game state is `ENewGame`), it sets the state to `EPlaying`. If the game is in play, it calls the `TryMakeMove()` function of the engine class to check if the move is valid; if so, it records the move and switches whose turn it is. A call to the `GameWonBy()` function of the engine class checks if the last move resulted in a win by either player and, if it did, the game state is set to `EFinished` and a call to the `ReportWinner()` function of the application UI class displays the result.

In its current form, `HitSquareL()` does not check for, or report, a drawn game. If there is no winner, the game continues until all tiles contain a nought or a cross (or a player uses a menu item to start a new game). The functionality for a drawn game is added in the communicating version of the game, described in Chapter 20.

The controller is reset for a new game by calling `Reset()`, which cancels the current game, clears the board (by calling the `Reset()` function of the engine class) and sets Noughts as the current player:

```

void COandXController::Reset()
{
    Engine().Reset();
    iState = ENewGame;
    if (IsCrossTurn())
    {
        SwitchTurn();
    }
}

```

The `SwitchTurn()` function simply negates the flag held in `iCrossTurn` and calls the `ReportWhoseTurn()` function of the application UI class:

```

void COandXController::SwitchTurn()
{
    iCrossTurn = !iCrossTurn;
    OandXAppUi() ->ReportWhoseTurn();
}

```

Persistence of the state of the controller class is handled by the `ExternalizeL()` and `InternalizeL()` functions, which are called from the application UI class. Their actions are, respectively, to write and read the controller's member data to or from the passed stream:

```

void COandXController::ExternalizeL(RWriteStream& aStream) const
{
    aStream.WriteUInt8L(iState);
    aStream.WriteInt8L(iCrossTurn);
}

void COandXController::InternalizeL(RReadStream& aStream)
{
    iState = static_cast<TState>(aStream.ReadUInt8L());
    iCrossTurn = static_cast<TBool>(aStream.ReadInt8L());
}

```

The controller is created via a call to its static `NewL()` function, which is coded as:

```

COandXController* COandXController::NewL()
{
    COandXController* self=new(ELeave) COandXController;
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop();
    return self;
}

```

The second-phase constructor is simply:

```
void COandXController::ConstructL()
{
    Reset();
}
```

We have coded it in this way to preserve the standard form of class construction, as described in Chapter 4. However, since – in this case – `ConstructL()` does not leave, it is worth pointing out that we could have simplified matters by not defining a `ConstructL()` function and doing everything in the `NewL()` function.

## The Engine Class

The engine class, whose class definition is listed below, represents the game data and contains the functions that operate on that data. Since Noughts and Crosses is a simple game, the engine class is correspondingly straightforward. The class itself has no knowledge of the overall state of the game, nor of whose turn it is to make a move. As you saw in the previous section, both of these are the responsibility of the controller.

```
class COandXEngine : public CBase
{
public:
    static COandXEngine* NewL();
    virtual ~COandXEngine();

    void Reset();
    TInt TileStatus(TInt aIndex) const;
    TBool TryMakeMove(TInt aIndex, TBool aCrossTurn);
    TTileState GameWonBy() const;
    // persistence
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);
private:
    COandXEngine();
    TInt TileState(TInt aX, TInt aY) const;
private:
    TFixedArray<TTileState, KNumberOfTiles> iTileStates;
};
```

The current state of the board is represented by a simple array of tile states, one for each component tile. Each tile state contains one of the three values `ETileBlank`, `ETileNought` or `ETileCross`.

The key functions are `TryMakeMove()` and `GameWonBy()`, both of which are called from the controller. `TryMakeMove()` only allows a move to be made if the corresponding tile is initially blank. If the move is allowed, the function sets the tile-state variable corresponding to the

tile to either `ETileCross` or `ETileNought`, depending on the value passed in `aCrossTurn`.

```
TBool COandXEngine::TryMakeMove(TInt aIndex, TBool aCrossTurn)
{
    if (iTileStates[aIndex] == ETileBlank)
    {
        iTileStates[aIndex] = aCrossTurn ? ETileCross : ETileNought;
        return ETrue;
    }
    return EFalse;
}
```

`GameWonBy()` checks if there is a winner and, if so, whether the winner is the Nought player or the Cross player. It does so by determining if there is a horizontal, vertical or diagonal line of three noughts or three crosses. If there is such a line, it returns either `ETileNought` or `ETileCross`; otherwise it returns zero, to indicate that, as yet, there is no winner.

```
TTileState COandXEngine::GameWonBy() const
{
    const TInt KNoughtWinSum = KTilesPerSide * ETileNought;
    const TInt KCrossWinSum = KTilesPerSide * ETileCross;

    // is there a row or column of matching tiles?
    for (TInt i = 0; i < KTilesPerSide; ++i)
    {
        TInt rowSum = 0;
        TInt colSum = 0;
        for (TInt j = 0; j < KTilesPerSide; ++j)
        {
            rowSum += TileState(j, i);
            colSum += TileState(i, j);
        }
        if (rowSum == KNoughtWinSum || colSum == KNoughtWinSum)
            return ETileNought;
        if (rowSum == KCrossWinSum || colSum == KCrossWinSum)
            return ETileCross;
    }

    // is there a diagonal of matching tiles?
    TInt blTrSum = 0; // bottom left to top right
    TInt tlBrSum = 0; // top left to bottom right
    for (TInt i = 0; i < KTilesPerSide; ++i)
    {
        tlBrSum += TileState(i, i);
        blTrSum += TileState(i, KTilesPerSide - 1 - i);
    }
    if (blTrSum == KNoughtWinSum || tlBrSum == KNoughtWinSum)
        return ETileNought;
    if (blTrSum == KCrossWinSum || tlBrSum == KCrossWinSum)
        return ETileCross;
    return ETileBlank; // No winner
}
```

As with the controller class, the `ExternalizeL()` and `InternalizeL()` functions of the engine class, called from the application UI class, simply write and read the state – in this case, the values of each of the board’s tiles – to and from the appropriate stream:

```
void COandXEngine::ExternalizeL(RWriteStream& aStream) const
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        aStream.WriteInt8L(iTileStates[i]);
    }
}

void COandXEngine::InternalizeL(RReadStream& aStream)
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        iTileStates[i] = static_cast<TTileState>(aStream.ReadInt8L());
    }
}
```

Like the controller class, the engine class is constructed by means of a call to its static `NewL()` function and, again, there are no potentially leaving calls to be made from a second-phase constructor. For the engine class, we’ve chosen to illustrate an alternative way of implementing construction in such a case, where the non-leaving calls are made from the constructor rather than from the `NewL()` function:

```
COandXEngine* COandXEngine::NewL()
{
    return new(ELeave) COandXEngine;
}

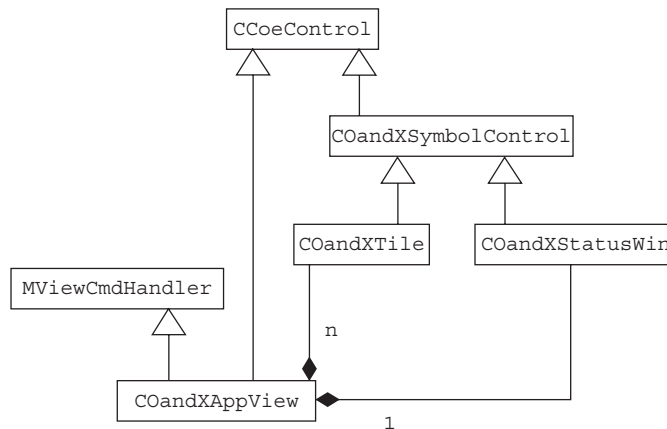
COandXEngine::COandXEngine()
{
    Reset();
}
```

## The View Class

Views and the view architecture are discussed in Chapter 14 and the detailed behavior of controls is described in Chapters 15, 17 and 18. In this chapter, therefore, we only briefly describe the general aspects of controls and views, concentrating on those features that are specific to the Noughts and Crosses application.

An application’s view is an instance of a window-owning control (as defined in Chapter 15) and derives, directly or indirectly, from the `CCoeControl` class. For the Noughts and Crosses application, the view is a compound control, with the view’s area being tiled with a number of subsidiary controls.

We've chosen to implement the S60 view class by deriving directly from `CCoeControl`. S60 applications can derive from more specific classes, such as `CAknView`, but we decided not to for this application in order to reduce the differences between the S60 and UIQ versions, and because many of the S60-specific views are more suited to text displays. The view also inherits from an `MViewCmdHandler` interface class, which we describe later.



**Figure 12.3** The View class of the Noughts and Crosses application

Figure 12.3 shows that all the components of the view are themselves also controls, in the sense that they all derive from `CCoeControl`. The view owns a single status-display control and a series of tiles.

Both the status display control and the tiles need to display either a nought or a cross, so we've chosen to implement them as subclasses of a generic symbol-drawing control, which knows how to draw those symbols. The two subclasses simply add the specification of the size and location, within their own area, of the symbol.

As you can see from the class definition of the symbol-drawing base class, its only member function is to draw a nought or a cross symbol:

```

class COandXSymbolControl : public CCoeControl
{
protected:
    void DrawSymbol(CWindowGc& aGc, const TRect& aRect,
                    TBool aDrawCross) const;
};
  
```

The implementation of this function is:

```

void COandXSymbolControl::DrawSymbol(CWindowGc& aGc,
    const TRect& aRect, TBool aDrawCross) const
  
```

```

{
TRect drawRect(aRect);

// Shrink by about 15%
drawRect.Shrink(aRect.Width()/6, aRect.Height()/6);

// Pen size set to just over 10% of the shape's size
TSize penSize(aRect.Width()/9, aRect.Height()/9);
aGc.SetPenSize(penSize);
aGc.SetPenStyle(CGraphicsContext::ESolidPen);

if (aDrawCross)
{
aGc.SetPenColor(KRgbGreen);

// Cosmetic reduction of cross size by half the line width
drawRect.Shrink(penSize.iWidth/2, penSize.iHeight/2);

aGc.DrawLine(drawRect.iTl, drawRect.iBr);
TInt temp;
temp = drawRect.iTl.iX;
drawRect.iTl.iX = drawRect.iBr.iX;
drawRect.iBr.iX = temp;
aGc.DrawLine(drawRect.iTl, drawRect.iBr);
}
else // draw a circle
{
aGc.SetPenColor(KRgbRed);
aGc.SetBrushStyle(CGraphicsContext::ESolidBrush);
aGc.DrawEllipse(drawRect);
}
};

```

You see this code again in Chapter 15, where the more detailed aspects of drawing this control are discussed; here, it is sufficient to note that it is written to be as independent as possible of the size or position of the symbol to be drawn. The fact that the same code successfully draws both the large symbols in the tiles and the much smaller symbols in the status area shows that this approach is both feasible and sensible.

The symbol is centered in the rectangle (which is always a square) passed to the function in `aRect`, but shrunk to leave a small border, and the pen width is set to match the size of the symbol. Note that the cross symbol is shrunk by a further small amount since, if the circles and crosses were both drawn to exactly the same size, the cross would look larger to the eye. This kind of attention to detail can make a large difference to the appearance of any application.

The tile and status window control classes both derive from `COandX-SymbolControl`:

```

class COandXTile : public COandXSymbolControl
{
public:
COandXTile();

```

```

~COandXTile();
void ConstructL(RWindow& aWindow);
// New function
void SetOwnerAndObserver(COandXAppView* aControl);
// From CCoeControl
TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent,
                             TEventCode aType);
TCoeInputCapabilities InputCapabilities() const;
protected:
void FocusChanged(TDrawNow aDrawNow);
void HandlePointerEventL(const TPointerEvent& aPointerEvent);
private:
void Draw(const TRect& aRect) const;
// New function
void TryHitL();
private:
MViewCmdHandler* iCmdHandler;
};

class COandXStatusWin : public COandXSymbolControl
{
public:
static COandXStatusWin* NewL(RWindow& aWindow);
~COandXStatusWin();
private:
COandXStatusWin();
void ConstructL(RWindow& aWindow);
void Draw(const TRect& aRect) const;
}

```

Note that, even though S60 devices do not normally use touch screens, we include code that handles input from keys – via `OfferKeyEventL()` – and any pointer device – via `HandlePointerEventL()`. It is quite safe to do this: if a particular device does not support one or the other input device, the relevant code simply is not called. Including both options makes it easier to port the code to different devices.

The status window's `Draw()` function simply calculates the square in which to draw the symbol and then calls `DrawSymbol()`, obtaining the flag indicating which symbol to draw by calling the `IsCrossTurn()` function of the controller class:

```

void COandXStatusWin::Draw(const TRect& /*aRect*/) const
{
CWindowGc& gc = SystemGc();
TRect boxRect = Rect();
gc.Clear(boxRect);
TInt boxHeight = boxRect.iBr.iY - boxRect.iTl.iY;
boxRect.iTl.iX = boxRect.iBr.iX - boxHeight;
DrawSymbol(gc, boxRect, Controller().IsCrossTurn());
}

```

The `Draw()` function for each tile works in a similar way:

```
void COandXTile::Draw(const TRect& /*aRect*/) const
{
    TTileState tileType;
    tileType = iCmdHandler->TileStatus(this);

    CWindowGc& gc = SystemGc();
    TRect rect = Rect();

    if (IsFocused())
    {
        gc.SetBrushColor(KRgbYellow);
    }
    gc.Clear(rect);
    if (tileType!=ETileBlank)
    {
        DrawSymbol(gc, rect, tileType==ETileCross);
    }
}
```

In this case, the symbol to be drawn is determined by a call to the `TileStatus()` function of the view class, by a mechanism that is described later. The tile which currently has focus is drawn with a yellow background; the others are drawn with the default brush color, which is white.

During the game, a player places a nought or a cross in a particular tile either by tapping on it or by using cursor keys to move the highlight to that tile and pressing the phone's Selection or Action key. A pointer event is handled by the tile's `HandlePointerEventL()` function:

```
void COandXTile::HandlePointerEventL (const TPointerEvent& aPointerEvent)
{
    if (aPointerEvent.iType == TPointerEvent::EButton1Down)
    {
        TryHitL();
    }
}
```

A key event is handled by `OfferKeyEventL()`:

```
TKeyResponse COandXTile::OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                         TEventCode aType)
{
    TKeyResponse keyResponse = EKeyWasNotConsumed;
    if (aType!=EEventKey)
    {
        return keyResponse;
    }
    switch (aKeyEvent.iCode)
    {
        case EKeyOK:
```

```

    TryHitL();
    keyResponse = EKeyWasConsumed;
    break;
default:
    keyResponse = EKeyWasNotConsumed;
    break;
}
return keyResponse;
}

```

In both cases, this results in a call to the tile's `TryHitL()` function, which is implemented as:

```

void COandXTile::TryHitL()
{
    if (iCmdHandler->TryHitSquareL(this))
    {
        DrawDeferred();
    }
}

```

`TryHitL()` calls the `TryHitSquareL()` function of the view class, using the same mechanism that is also used to call `TileStatus()`.

The `COandXAppView` class definition has 19 member functions, not counting those that it inherits from its base classes, but only the two mentioned above are called from a tile. As an encapsulation aid, we have chosen to declare these two functions, as pure virtual functions, in an `MViewCmdHandler` interface class:

```

class MViewCmdHandler
{
public:
    virtual TBool TryHitSquareL(const COandXTile* aControl) = 0;
    virtual TTileState TileStatus(const COandXTile* aControl) const = 0;
};

```

The view class inherits from this interface and provides the concrete implementations, as indicated by the bold portions of the `COandXAppView` class definition:

```

class COandXAppView : public CCoeControl, public MCoeControlObserver,
                    public MViewCmdHandler
{
public:
    static COandXAppView* NewL(const TRect& aRect);
    virtual ~COandXAppView();

    // From CCoeControl
    TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                TEventCode aType);

```

```

// new functions
void MoveFocusTo(const TInt aIndex);
TInt IdOfFocusControl();
void ShowTurn();
void ResetView();

private:
COandXAppView();
void ConstructL(const TRect& aRect);

void SwitchFocus(TInt aFromIndex, CCoeControl* aToControl);
void DrawComps(TRect& aRect) const;
COandXTile * CreateTileL();

// From CCoeControl
void Draw(const TRect& aRect) const;
void SizeChanged();
TInt CountComponentControls() const;
CCoeControl* ComponentControl(TInt aIndex) const;

// From MCoeControlObserver
void HandleControlEventL(CCoeControl* aControl, TCoeEvent aEventType);

// From MViewCmdHandler
TBool TryHitSquareL(const COandXTile* aControl);
TTileState TileStatus(const COandXTile* aControl) const;

private:
RPointerArray<COandXTile> iTiles; // View owns the tiles
COandXStatusWin* iStatusWin;    // and its own status window.
TRect iBoardRect; // Board area
TRect iBorderRect; // Bounding rectangle for border
TInt iTileSide; // Tile dimension, allowing for line widths
                // and border
};

```

Note that the functions are declared public in the interface class, but private within `COandXAppView`. By providing each tile with an `MViewCmdHandler*` pointer to the enclosing view, we expose only those two functions (they are not even available via a `COandXAppView*` pointer).

The interface class is a useful way of encapsulating the interaction between the control and the view or, indeed, the rest of the program. The control does not care about any of the view's internal details, provided that it correctly handles the request to play in a particular square and the querying of a square's content. Similarly, the view does not care whether the command originated with a key event, a pointer event, or some other kind of event – it just requires these functions to be called at the right time with the right parameters.

This is a useful technique, which is used frequently within a variety of Symbian OS APIs, and it is a good one to adopt for your programs, especially if they are more complicated than the Noughts and Crosses application. As a further illustration, think about the way commands

reach `HandleCommandL()`. The menu bar uses an `MEikMenuObserver` for this and does not otherwise care about the large API of the `CEikAppUi` class. The button bar uses the `MEikCommandObserver` interface. Similarly, command buttons use an `MCoeControlObserver` interface, which the button bar implements by converting button events into application commands.

`TryHitSquareL()` simply calls the controller's `HitSquareL()` function, to determine whether the attempt is successful:

```
TBool COandXAppView::TryHitSquareL(const COandXTile* aControl)
{
    return Controller().HitSquareL(Index(aControl));
}
```

It returns `ETrue` only if the square was previously empty and now contains a symbol, otherwise it returns `EFalse`.

As was mentioned earlier, `TileStatus()` is called from each tile when it needs to draw itself, in order to determine which, if any, symbol should be drawn in the tile. The function's action is simply to request the information from the engine:

```
TTileState COandXAppView::TileStatus (const COandXTile* aControl) const
{
    return Engine().TileStatus(Index(aControl));
}
```

The majority of the view's functions are described in Chapter 15, where they are used to illustrate the general behavior of controls. Here, we mention only `ShowTurn()`, which is called from the application UI class and the controller whenever there is a change in whose turn it is to play. It simply requests the status window to redraw itself:

```
void COandXAppView::ShowTurn()
{
    iStatusWin->DrawDeferred();
}
```

How `DrawDeferred()` works is explained in Chapter 17; here it is sufficient to know that it results in a call to the status window's `Draw()` function. As we saw earlier, there is no need for `ShowTurn()` to pass a parameter indicating whose turn it is, since the status window's `Draw()` function queries the controller directly for the symbol it needs to draw.

## Command Menus

The application uses only two commands: one to specify whether the Nought player or the Cross player should make the first move, and one

to abandon the current game. The first of these is only relevant before a game starts and the second is only used while a game is in progress. In consequence, we've chosen to use a dynamic command menu, which shows only those commands that can sensibly be used, given the current state of the game. We've implemented the menu in a way that illustrates some of the options that are available for setting up and changing the content of a command menu. Although at least one of the commands, that to select the first player, could reasonably be implemented via a dialog, we've chosen not to do so, leaving the discussion of dialog use to Chapter 16.

For S60, the specification of the command menu starts in the resource script, with the definition of an `EIK_APP_INFO` resource:

```
RESOURCE EIK_APP_INFO
{
    menubar = r_oandx_menubar;
    cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
}
```

This resource contains a pointer to the resource that contains the application's menu bar and a Control Button Array that defines the softkey labels – in this case, Options and Exit – that are to appear in the control pane.

The menu bar resource, in turn, specifies one or more `MENU_PANE` resources that list the menu items they contain. As in this case, an S60 menu bar normally contains only one menu pane. If you specify more than one menu pane, the S60 UI concatenates their contents into a single list.

```
RESOURCE MENU_BAR r_oandx_menubar
{
    titles =
    {
        MENU_TITLE
        {
            menu_pane = r_oandx_menu;
        }
    };
}
```

For this application, the menu pane specifies only one menu item; the second menu item is constructed dynamically.

```
RESOURCE MENU_PANE r_oandx_menu
{
    items =
    {
        MENU_ITEM
```

```

    {
        command = EOandXNewGame;
        txt = "New game";
    }
};
}

```

Each menu item specifies the text to be displayed and a non-zero value, unique within the application, used to identify the command. The command IDs are normally defined via an enumeration in the application's HRH file; for this application, they are defined, in `oandx.hrh`, as:

```

enum TOandXIds
{
    EOandXNewGame = 0x1000, // start value must not be 0
    EOandXSwitchTurn
};

```

The resource file specifies only the command ID and text for the New game command. The other command is added dynamically according to context, using one or other of the following two resources for its text:

```

RESOURCE TBUF r_oandx_o_moves_first
{
    buf = "Noughts move first";
}

RESOURCE TBUF r_oandx_x_moves_first
{
    buf = "Crosses move first";
}

```

It is worth noting here that a truly commercial application would read the text for these resources – and most other text within the resource script – from a separate file, in order to facilitate translation into other languages. We have not done so in order to make the meaning of these resources clearer in the text of this book.

S60 applications that need to modify the content of their command menus should supply an implementation of the `DynInitMenuPaneL()` function of the application UI class, which is called each time a menu is about to be displayed. The `aResourceId` parameter identifies the menu and `aMenuPane` points to a menu pane that has already been constructed from the relevant resource. For this application, the implementation is:

```

void COandXAppUi::DynInitMenuPaneL(TInt aResourceId,
                                   CEikMenuPane* aMenuPane)
{
    if (aResourceId == R_OANDX_MENU)

```

```

{
    if (iController->IsNewGame())
    {
        CEikMenuItem::SData item;
        iCoeEnv->ReadResource(item.iText,
            iController->IsCrossTurn() ? R_OANDX_O_MOVES_FIRST :
                R_OANDX_X_MOVES_FIRST);
        item.iCommandId = EOandXSwitchTurn;
        item.iFlags = 0;
        item.iCascadeId = 0;
        aMenuPane->AddMenuItemL(item);

        aMenuPane->DeleteMenuItem(EOandXNewGame);
    }
}
}

```

The function first checks that the resource ID is for the relevant menu, but does not modify the menu pane (which therefore displays the New game menu item) if a game is in progress. Otherwise, a new menu item is constructed, in a `CEikMenuItem::SData` data structure, using the relevant resource text, and added to the menu pane, and the New game item is deleted.

When the user selects a command, the S60 UI responds by calling the `HandleCommandL()` function of the application UI class. This is normally implemented via a `switch` statement, with each case corresponding to one of the possible command IDs:

```

void COandXAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        case EEikCmdExit:
        case EAknSoftkeyExit:
        {
            SaveL();
            Exit();
        }
        break;
        case EOandXNewGame:
        {
            iController->Reset();
            iAppView->ResetView();
        }
        break;
        case EOandXSwitchTurn:
        {
            iController->SwitchTurn();
        }
        break;
        default:
            break;
    }
}
}

```

In the Noughts and Crosses application, each case, with the exception of the Exit command, corresponds to one of the command IDs specified in `oandx.hrh` and is handled simply by making the appropriate changes to the controller's state.

Depending on how the Exit command is initiated, it may have either of the standard command IDs shown in the above code. The application's response must always be to call `Exit()` but, in addition, it calls `SaveL()` to save the current game state in the application's persistent data.

## 12.2 Differences for UIQ 3

One of the major influences in the development of UIQ 3 was the need to make it easier for developers to support multiple form factors (both in terms of screen dimensions and use of portrait or landscape alignments) and a variety of input methods (touch screen or keyboard; menu-driven or softkey-driven interaction) from a single code base.

In order to facilitate such flexibility, all UIQ 3 applications need to derive their views from the new `CQikViewBase` or `CQikMultiPageViewBase` classes. The UIQ version of the Noughts and Crosses application derives its main view from the `CQikViewBase` view class (which is a subclass of `CCoeControl`) and not directly from `CCoeControl` itself. However, since it only has one view, it makes no specific use of the view architecture that is described in Chapter 14.

### The Application UI Class

There is a small difference in the way that the application UI class creates and initializes the View. Remember that the `S60 ConstructL()` function of the application UI class is implemented as:

```
void COandXAppUi::ConstructL()
{
    BaseConstructL(EAknEnableSkin);
    iEngine = COandXEngine::NewL();
    iController = COandXController::NewL();
    iAppView = COandXAppView::NewL(ClientRect());
    AddToStackL(iAppView); // Enable keypresses to the view
    iStacked = ETrue;
    ReportWhoseTurn();
}
```

The destructor needed to take note of `iStacked` in order to determine whether or not it should remove the view from the control stack. For UIQ, the corresponding code is:

```
void COandXAppUi::ConstructL()
{
```

```

CQikAppUi::ConstructL(); // initiate the standard values
iEngine = COandXEngine::NewL();
iController = COandXController::NewL();
// Create the view and add it to the framework
iAppView = COandXView::NewL(*this);
AddViewL(*iAppView);

ReportWhoseTurn();
}

```

The most significant difference is that the UIQ version registers the view by calling `AddViewL()`, which also takes care of adding it to the control stack. After this call, the application UI class has full ownership of the view, so you do not have to worry about either deregistering the view or removing it from the control stack when the application UI class is destroyed.

## The View Class

The majority of the remaining source code differences that are needed in a UIQ version of the application stem from the differences between `CQikViewBase` and `CCoeControl`. So, while both forms of the view own their component controls, the mechanisms of ownership and access are different. A view that derives from `CCoeControl`, as in the S60 version, creates its component controls (normally in its `ConstructL()` function) and stores pointers to them in specific items of member data. In the S60 version, the relevant `ConstructL()` code is:

```

...
for (TInt i = 0; i < KNumberOfTiles; i++)
{
    User::LeaveIfError(iTiles.Append(CreateTileL()));
}
...
iStatusWin = COandXStatusWin::NewL(Window());

```

The controls are normally explicitly deleted in the destructor:

```

COandXAppView::~COandXAppView()
{
    for (TInt i=0; i<KNumberOfTiles; i++)
    {
        delete iTiles[i];
    }
    iTiles.Close();
    delete iStatusWin;
}

```

As described more fully in Chapter 17, access to each child control is via the `CountComponentControls()` and `ComponentControl()` functions that each component-owning control must implement:

```

TInt COandXAppView::CountComponentControls() const
{
    return KNumberOfTiles + 1;
}

CCoeControl* COandXAppView::ComponentControl(TInt aIndex) const
{
    if (aIndex==KNumberOfTiles)
    {
        return iStatusWin;
    }
    else
    {
        return const_cast<COandXTile*>(iTiles[aIndex]);
    }
}

```

CQikViewBase provides its own mechanism for component control ownership, so views that derive from this class do not need to provide explicit data members to store pointers to the components, nor do they need to provide code in the destructor to delete them. In addition, such views do not need to (and should not) provide implementations of `CountComponentControls()` and `ComponentControl()`.

Instead, UIQ views add component controls – normally from within the view's implementation of `ViewConstructL()` – to a built-in array that is initialized by a call to `InitComponentArrayL()` and then accessed via a `Components()` function, for example:

```

void COandXView::ViewConstructL()
{
    ViewConstructFromResourceL(R_OANDX_UI_CONFIGURATIONS);
    InitComponentArrayL();
    for (TInt i = 0; i < KNumberOfTiles; i++)
    {
        COandXTile * tile = new(ELeave) COandXTile;
        AddControlLC(tile, i);
        tile->ConstructL(this);
        tile->SetFocusing(ETTrue);
        CleanupStack::Pop(tile);
    }
    // Status window added, but not created here
    AddControlLC(iStatusWin, KNumberOfTiles);
    CleanupStack::Pop(iStatusWin);
    iStatWinNotAppended = EFalse;
    // Status window now owned by the view

    UpdateCommandsL(Controller().IsNewGame(), Controller().IsCrossTurn());
    SetFocusByIdL(0);
}

```

In the UIQ version of the application, as you can see from the above code, each tile is created and then appended to the array with a call to `AppendLC()` which, as its name implies, leaves a pointer to the

tile on the cleanup stack. The tile is removed from the cleanup stack once any potentially leaving initialization is complete. The view takes ownership of all component controls that are added in this way and they are automatically deleted when the view is destroyed.

If you think carefully about the code sequence for adding a component into a view, you might begin to wonder if it is truly safe. For example, what if the actions performed by `AddControlLC()` cause a leave before the control is added to the cleanup stack? Furthermore, if the view takes ownership of the control, should `AddControlLC()` leave the control on the cleanup stack at all? Could a leave before the control is popped from the cleanup stack result in the control being deleted twice – once from the cleanup stack and once by the owning view? In Symbian OS, it is worth asking questions like this, since leave-safeness is paramount.

In fact, this sequence is safe. `AddControlLC()` adds the control to the cleanup stack before performing any potentially leaving actions, and the item that is added to the cleanup stack is one that is specifically designed to transfer ownership of the view (in a leave-safe way) as it is popped.

The creation of the status window follows a slightly different pattern. The reason for this is that the status window is accessed from the controller, via the `ReportWhoseTurn()` function of the application UI class, and the first access occurs before the view's `ViewConstructL()` has been called. (To optimize start-up time, particularly for applications with multiple views, `ViewConstructL()` is only called at the time that the view first needs to be made visible.) To prevent this attempted access from causing a panic, we need to create the status window in the view's `ConstructL()` and add it in `ViewConstructL()`.

However, there is then a risk that the status window will not be deleted if the code leaves between its creation and it being successfully added. We solved this problem by including the `iStatWinNotAppended` flag in the class data, setting it when the status window is created and clearing it when the window has successfully been added to the array. The view's destructor has to delete the status window only if the flag is set:

```
COandXView::~COandXView()
{
    if (iStatWinNotAppended)
        // Once appended, system code takes care of a component's deletion
        {
            delete iStatusWin;
        }
}
```

It is also worth pointing out that the initial construction of the view, from `ViewConstructL()`, is by means of a call to `ViewConstructFromResourceL()`. In suitable cases, an entire multi-page view can be constructed from the resource accessed by this function, in the same way that a dialog can be constructed from a resource, as described in Chapter 16. In the present case, the resource specifies a single page, without supplying any content:

```
RESOURCE QIK_VIEW_PAGES r_oandx_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = EOandXViewPage;
        }
    };
}
```

Since the UIQ version of the view class does not have accessible implementations of `ComponentControl()` or `CountComponentControls()`, functions that, in an S60 application, call one or both of these functions need a different implementation. In the Noughts and Crosses application, the functions that are affected are those associated with which tile has focus. In the S60 version, these are `IdOfFocusControl()` and `MoveFocusTo()`:

```
TInt COandXAppView::IdOfFocusControl()
{
    TInt ret = -1;
    for (TInt i=0; i<KNumberOfTiles; i++)
    {
        if (ComponentControl(i)->IsFocused())
        {
            ret = i;
            break;
        }
    }
    ASSERT_ALWAYS(ret>=0, Panic(EOandXNoTileWithFocus));
    return ret;
}

void COandXAppView::MoveFocusTo(const TInt index)
{
    TInt oldIndex = IdOfFocusControl();
    if (index != oldIndex)
    {
        SwitchFocus(oldIndex, ComponentControl(index));
    }
}
```

`SwitchFocus()` is implemented as:

```
void COandXAppView::SwitchFocus(TInt aFromIndex, CCoeControl* aToControl)
{
    ComponentControl(aFromIndex)->SetFocus(EFalse, EDrawNow);
    aToControl->SetFocus(ETrue, EDrawNow);
}
```

In the UIQ version, the two corresponding functions are `IdOfFocusedControl()` and `SetFocusByIdL()`, whose basic implementations are:

```
TInt COandXView::IdOfFocusedControl()
{
    TInt i;
    for (i=0; i<KNumberOfTiles; i++)
    {
        CCoeControl *control=ControlById<CCoeControl>(i);
        if (control->IsFocused())
            break;
    }
    return i;
}

void COandXView::SetFocusByIdL(TInt aControlId)
{
    RequestFocusL(ControlById<CCoeControl>(aControlId));
}
```

In the UIQ version, we changed the names of both functions to emphasize the differences in implementation. Note, in particular, that `SetFocusByIdL()` is a leaving function, whereas `MoveFocusTo()` is not. Fortunately this does not give rise to any other complications in the current application, since they are called only from within leaving functions.

While on the subject of focus, there are two further differences that need mentioning. First, the default status of newly created controls in S60 is for them to be capable of accepting focus, which is not the case in UIQ. In UIQ you need to call `SetFocusing(ETrue)` on each control that needs to accept focus (in this case, on each of the board's tiles). In contrast, in S60, you need to call `SetFocusing(EFalse)` on each control, such as the game's status window, that you do not want to be able to accept focus. Secondly, UIQ 3 views provide automatic, intelligent key- or button-based navigation between any of the component controls that can accept focus – that is, those controls for which `SetFocusing(ETrue)` has been called. In consequence, the UIQ version has no need to implement the view's `OfferKeyEventL()` function.

## Commands

UIQ 3 views are primarily designed for use in applications with multiple views, each with its own set of commands, so they are not seen at their

best in an application such as the Noughts and Crosses application, which only has a single view. The application does, however, allow the basic differences to be illustrated.

A major difference in UIQ is that commands are treated in a more abstract way than in S60. A UIQ device may be set up to access its commands by means of either a pointer device or softkeys, and the commands themselves may or may not be displayed in a command menu. In addition, commands may be provided from a variety of sources within the application, such as one or more individual controls – or even from outside the application itself, for example, from a front-end processor.

The UIQ application framework contains a command model that deals with all these variations. In consequence, the application programmer need only be concerned with which commands should be available and not with how they are presented or selected.

The first change is to the resource script. As we saw earlier, an S60 application uses an `EIK_APP_INFO` resource, to specify the menu bar and the menu items it contains. In contrast, a UIQ 3 application must specify an empty `EIK_APP_INFO` resource, plus a `QIK_VIEW_CONFIGURATIONS` resource that lists the UI configurations it supports. In the Noughts and Crosses application, these resources are:

```
RESOURCE EIK_APP_INFO { }

RESOURCE QIK_VIEW_CONFIGURATIONS r_oandx_ui_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            command_list = r_oandx_portrait_commands;
            view = r_oandx_layout;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikSoftkeyStylePortrait;
            command_list = r_oandx_portrait_commands;
            view = r_oandx_layout;
        }
    };
}
```

Each configuration specifies its list of available commands via a `QIK_COMMAND_LIST` resource:

```
RESOURCE QIK_COMMAND_LIST r_oandx_portrait_commands
{
    items =
    {
```

```

QIK_COMMAND
{
    id = EEikCmdExit;
    type = EQikCommandTypeScreen; // Only visible in debug
    stateFlags = EQikCmdFlagDebugOnly;
    text = "Close (debug)";
},
QIK_COMMAND
{
    id = EOandXNewGame;
    type = EQikCommandTypeScreen;
    text = "New game";
},
QIK_COMMAND
{
    id = EOandXFirstPlayer;
    type = EQikCommandTypeScreen;
    text = "Noughts move first";
}
};
}

```

As in the S60 version, the commands are updated dynamically, but the mechanism is different. S60 applications update their menus via the `DynInitMenuPanel()` function, which is called immediately before a menu is made visible.

In UIQ, since a command may be selected by means of a softkey, or from a toolbar, there is no guarantee that a menu pane is displayed before the selection takes place. In consequence, you cannot use `DynInitMenuPanel()` to update the available commands. Indeed, since some commands may be permanently visible, UIQ applications need to perform such updating each time the view's state changes in a way that affects the commands.

In many cases, it would be appropriate to implement the view's `HandleControlEventL()` function, which is called (with an `EEventStateChanged` event code) each time one of the view's controls changes its state. In the Noughts and Crosses application, all significant state changes take place in the Controller, so we've taken the more direct option of adding a `SetStateL()` function to the Controller, implemented as:

```

void COandXController::SetStateL(TState aState)
{
    iState = aState;
    OandXAppUi()->UpdateCommandsL(IsNewGame(), IsCrossTurn());
}

```

This calls an `UpdateCommandsL()` function in the application UI class, which simply calls a similarly named function in the view:

```

void COandXView::UpdateCommandsL(TBool aIsNew, TBool aIsCrossTurn)
{

```

```
iCommandManager.SetAvailable(*this, EOandXNewGame, !aIsNew);
iCommandManager.SetAvailable(*this, EOandXFirstPlayer, aIsNew);
iCommandManager.SetTextL(*this, EOandXFirstPlayer,
    aIsCrossTurn?R_OANDX_O_FIRST:R_OANDX_X_FIRST);
}
```

The following points are relevant when updating the menus:

- While a game is in progress, there is no need to worry about whether the text for the ‘Who moves first’ command is updated correctly or not, because that command is only available (and visible) before a new game is started.
- When executing the ‘Who moves first’ command, the game is always new, so the only interest is in setting the correct text for the next potential use of that command.

In addition, the controller’s `Reset()` function is renamed to `ResetL()` because it can now leave:

```
void COandXController::ResetL()
{
    Engine().Reset();
    if (IsCrossTurn())
    {
        SwitchTurn();
    }
    SetStateL(ENewGame);
}
```

`ResetL()` cannot be used to initialize the controller (and engine) because `SetStateL()` calls the view’s `UpdateCommandsL()` function, and the view does not exist at the time the controller is constructed. Instead, in the controller’s `ConstructL()`, we explicitly set the relevant data:

```
...
Engine().Reset();
iState = ENewGame;
iCrossTurn = EFalse;
...
```

Since UIQ expects an application to have multiple views and each view may have a different set of commands, command processing is implemented within each individual view. A UIQ application, therefore, does not supply a `HandleCommandL()` function in the application UI class, but implements this function in each of its views. Note that the argument to this function is a reference to an instance of `CQikCommand`, rather than the command ID that is passed to this function in the

application UI class. Otherwise, the implementation is broadly similar to that in the S60 version:

```
void COandXView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        case EOandXNewGame:
            Controller().ResetL();
            SetFocusByIdL(0);
            DrawNow();
            break;
        case EOandXFirstPlayer:
            Controller().SwitchTurn();
            UpdateCommandsL(Controller().IsNewGame(),
                           Controller().IsCrossTurn());
            break;
        default:
#ifdef _DEBUG
            OandXAppUi()->SaveGameStateL();
#endif
        // The Go back and Exit commands are handled by CQikViewBase.
        CQikViewBase::HandleCommandL(aCommand);
        break;
    }
}
```

## Persistence

In UIQ, the default behavior is to allow applications to use persistent data, so, unlike in S60, there is no need to supply an implementation of the document's `OpenFileL()` function.

UIQ applications do not normally have a Close command (except for testing purposes, in debug builds). Instead, system code may instruct an application to close at any time that memory needs to be freed, first giving it an opportunity to save its persistent data. In order to access the functionality more easily, we have added – but only in debug builds – a call to the `SaveGameStateL()` function of the application UI class in the default case of the view's `HandleCommandL()`. The implementation is very simple:

```
void COandXAppUi::SaveGameStateL()
{
    SaveL();
}
```

We have to supply this new function because the `SaveL()` function of the application UI class is protected, and so cannot be called directly from the view. With this call in place, the persistent data is saved each time the Close (Debug) command is selected.

Since the application's persistent data contains one item – the ID of the tile that currently has focus – from the view, there is one final complication that we have to deal with. It is caused by the view's controls being created and initialized in the `ViewConstructL()` function which, as mentioned earlier, is called as late as possible, just before the view first becomes visible. This means that the document's `RestoreL()` function (and, at least when the application is run for the first time, its `StoreL()` function) are called before the view is fully initialized.

We resolve this problem by adding an `iInitialFocusId` data member to the `COandXView` class and coding the view's `SetFocusByIdL()` function as:

```
void COandXView::SetFocusByIdL(TInt aControlId)
{
    CCoeControl * control = ControlById<CCoeControl>(aControlId);
    if (control)
    {
        RequestFocusL(control);
    }
    else // Control does not yet exist
    {
        iInitialFocusId = aControlId;
    }
}
```

If the control does not yet exist, the value is copied into `iInitialFocusId` and used, within `ViewConstructL()`, to move focus to the correct control after it has been created.

We also need to deal with the case where the persistent data needs to be written before the tiles exist (this only happens once, the first time the application is run, when the application's INI file is being created). The relevant data is obtained by a call to the view's `IdOfFocusedControl()`, whose implementation is:

```
TInt COandXView::IdOfFocusedControl()
{
    TInt i;
    for (i=0; i<KNumberOfTiles; i++)
    {
        CCoeControl *control=ControlById<CCoeControl>(i);
        if (!control) // Not yet created
        {
            return 0;
        }
        if (control->IsFocused())
        {
            break;
        }
    }
    return i;
}
```

If a control does not yet exist, the function simply returns zero, which is the correct default value.

## Summary

This chapter has described a simple but non-trivial application to play a game of Noughts and Crosses.

After a brief introduction to the game, the chapter explained the overall architecture of the application, before describing each of the S60 version's major component classes in more detail.

The chapter concluded with a discussion of the main differences between the S60 and UIQ versions of the game.