

UML example, including illustration of a framework

1. Introduction (see text p224)

Rough definition: A framework is a reusable piece of architecture.

Discussion: Suppose we have a collection of objects. Then a framework describes how these objects work together, usually by

- defining classes which will be sub-classed when the framework is applied
- describing the collaborations between the objects

To use a framework you implement the variable parts of the framework, for example by sub-classing from what is provided. Some of the classes are normally abstract so that you have to create a sub-class before you can use the framework.

2. Introductory Framework Example - Board Games (text Chap. 16)

2.1 *The problem*

A company plans to produce a range of on-screen versions of well-known two-player games, starting with "Noughts and Crosses" (i.e. Tic-Tac-Toe) and Chess. In such games, two users share a screen and make moves in turn. The program makes sure that only legal moves are allowed and also declares the winner (or draw).

To make implementation of new games easy, and to ease the work of maintaining many different games, the company wants to design a game **framework**, which can be reused as generally as possible.

2.2 Preliminary analysis

Note: In text it is assumed that Java will be used and that a game is implemented as an applet (but this will hardly affect our discussion).

Aim is to develop

- I) A suitable architecture for the family of games, and
- II) Any common functionality that is found.

In practice it is only through experience in analysing (and maybe implementing) several such games that we will get a really good notion of what the framework should consist of, particularly of any common functionality. However, we attempt to give an impression of what is involved.

After analysis of "Noughts and Crosses" and Chess, the authors come up with the following list of features that these 2 games have in common:

1. Game is played by 2 players
2. Games is played on a squared board (9 and 64 for the particular games)
3. Players move alternately
4. To make a move is to alter the state of the board by adding, removing and/or moving some tokens which are things (marks or pieces) on the board
5. Any token on the board is owned by one or other player
6. All relevant information is available to both players
7. Which moves are legal depends on the state of the board (i.e. which tokens are where) possibly together with factors like the history of the play
8. Who wins depends on the same factors as in 7

The authors mention that on looking at some other games, it can happen that features 2 (e.g. hexagonal board), 3 and 5 may not hold (tokens may not belong to a specific player).

2.3 Identifying Classes and Relationships¹

Authors state that the "noun identification" technique alone is not adequate to identify classes for this example (although of course it does suggest obvious candidates such as **board** and **token**).

They describe the technique they applied to identify classes and their relationships as follows:

- i) Begin with a few candidate classes and make a draft class model
 - ii) Considering examples of actual games, think through the main interactions, and begin to sketch corresponding CRC cards
- Repeat i) and ii) until a satisfactory class model is formulated.

Draft Class Model:

The framework classes identified are (*not too surprisingly!*):

Player	CurrentPosition	Token	Move	Game
--------	-----------------	-------	------	------

with corresponding draft CRC cards to clarify what is intended:

¹ Authors concluded that "use case" analysis was not likely to be helpful in this problem.

Player	
Responsibilities	Collaborators
Maintain any required user data Identify a player of the game Provide a visual symbol of the player	
CurrentPosition	
Responsibilities	Collaborators
Maintain data seen by user: position; whose turn it is; eventually who winner is Accept a user move and package it for validation	
Token	
Responsibilities	Collaborators
Represents a token of the game Maintain position of token (<i>TBC</i> ²) [Or is this CurrentPosition's job?- See after Figure 16.3] Provide a visual symbol for the token	
Move	
Responsibilities	Collaborators
Encapsulate what changes in one player's turn Know how to confirm itself? (<i>TBC</i>)	
Game	
Responsibilities	Collaborators
Understand the rules of the game: Validate moves Determine winner Retain any necessary information about past moves	

² TBC = "To Be Confirmed" is quite commonly used to identify a tentative statement that must be confirmed later. Similarly, for TBD = "To Be Decided (or Determined)".

Iteration of Class Model for "Noughts and Crosses":

In our limited presentation the framework approach will not yield any common functionality across games (as we will not consider enough of them) but *will provide a sound architecture* for each specific application.

Generally, the need is to implement classes to fulfill the responsibilities identified (in the CRC cards). For some or all of the framework classes, this means creating a specialized sub-class that fulfills, for the particular game concerned, the responsibilities assigned to that framework class. For "Noughts and Crosses", authors use naming convention of adding prefix "OXO" to identify any such specialized class (*although see Board below!*).

In some cases, objects of more than one class may be needed to fulfill the responsibilities of a single object of a framework class. In particular, for CurrentPosition the domain objects Board and Square can collaborate to maintain position of the game as seen by a user where,

Board	
Responsibilities	Collaborators
Maintain data seen by user: position; whose turn it is; eventually who winner is Accept a user move and package it for validation	
Square	
Responsibilities	Collaborators
Maintain data pertaining to a single square: where it is; whether it contains a token, and, if so, what kind Say whether a point falls inside the square	

Thus, Board is a subclass of (probably) abstract class CurrentPosition and it has a non-inherited attribute which is the collection of (9) Squares.

2.4 Interactions (illustrated within "Noughts and Crosses")

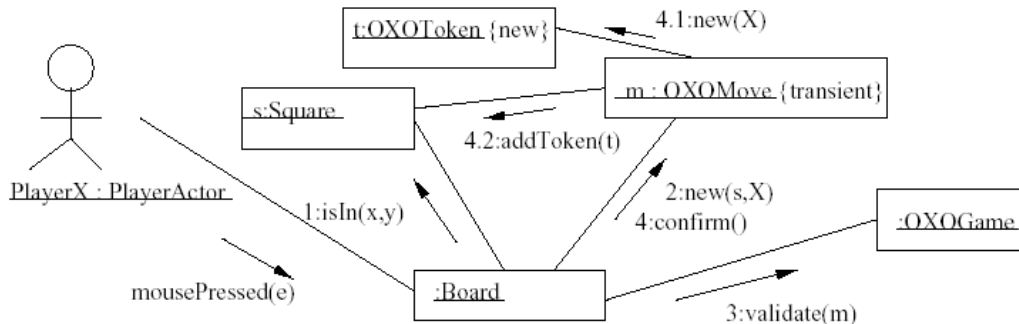


Figure 16.3 Collaboration diagram for an X move in Noughts and Crosses.

[mousePressed(e)] **Player** clicks on screen representation of board, causing a message to be sent to **Board** with information on where click happened.

[1:isIn(x,y)] **Board** with help from its **Squares** works out which **Square** has been clicked in.

[2:new(s,X)] **Board** knows it is **Player X's** turn (*how?*), so the meaning of the click is that X wishes to place a token on the square. So, **Board** creates a new **OXOMove** object which knows the **Square** and **Player** concerned.

[3:validate(m)] **Board** passes the **OXOMove** to **OXOGame** for validation. First, **OXOGame** checks that there is not already an **OXOToken** on the **Square**. We will assume there is not so the move is valid.

Next, **OXOGame** checks whether this move finishes the game. We will assume it does not and so **OXOGame's** reply is that the move is OK and that **Player O** is now to move.

[4:confirm()] **Board** asks the **OXOMove** to update the position it displays to the user

[4.1:new(X)] so (a) **OXOMove** creates a new X **OXOToken**

[4.2:addToken(t)] on the **Square** it refers to

(b) forgets the **OXOMove** which is no longer needed (to be collected as garbage)

(c) records that next move is to be made by O

(d) waits for next click, which will represent O's move.

2.5 Possible Class Model for "Noughts and Crosses"

Having analysed the "X move" interaction, the authors present the following as a possible class diagram:

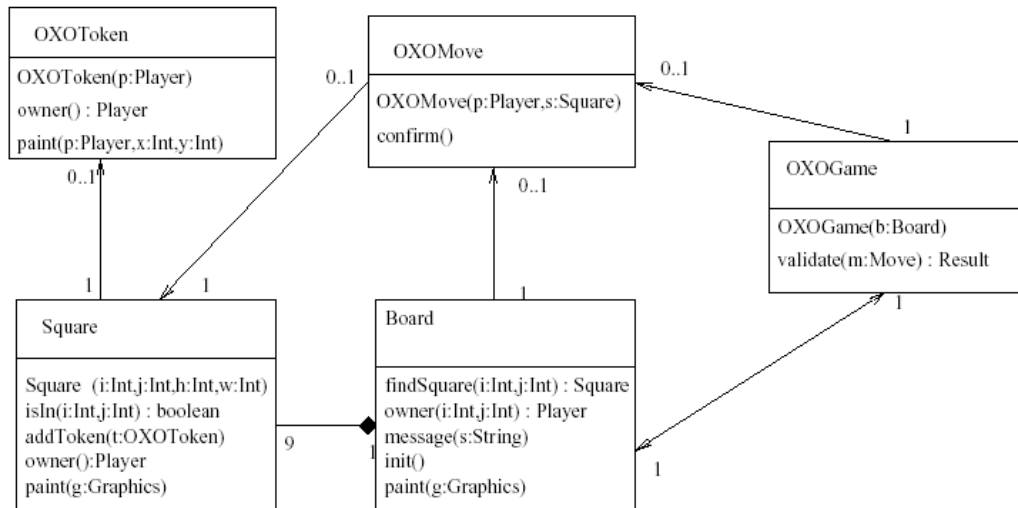


Figure 16.4 Class diagram for Noughts and Crosses.

The diagram is simplified in that the base (framework) classes are not included.

Also, **OXOPlayer** is omitted, which might have been thought to be a necessary sub-class. The authors state that

- (i) There is no need for a specialized sub-class so, if anything, **Player** should appear in Figure 16.4
- (ii) At this stage, at least, a Player object is essentially a boolean data item, on which every class depends, so including it would clutter the diagram unnecessarily.

A few operations have been added in comparison with Figure 16.3 - the reason for "init()" in **Board** is that it was decided to make **Board** a Java applet.

2.6 Back to the framework

Finally (as far as our presentation goes), having seen how one game fits into the initial notions of what the framework should be, the authors want to make a first attempt at describing the framework itself in more detail.

2.6.1 Draft Class Diagram for Framework

How the framework classes work together:

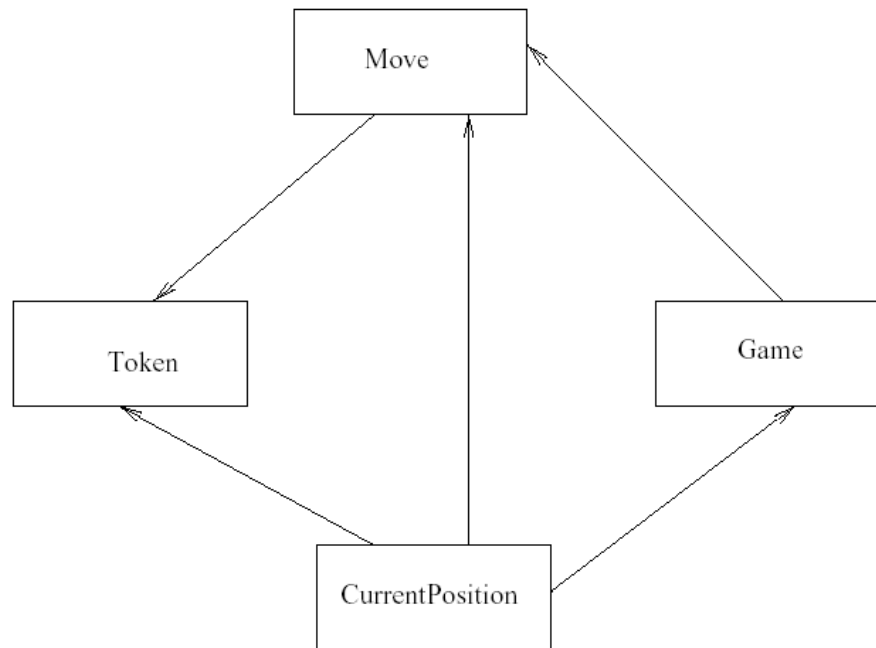


Figure 16.5 Class diagram for games framework.

Notice that the authors have really limited the level of detail:

- Only the associations which ***must*** exist between the abstract framework classes themselves are shown.
- Multiplicities are omitted.

Remark: Using a general framework may lead you into developing a more complex design than would be necessary for a particular application.

2.6.2 A useful State Diagram for Framework

It may be helpful, as part of the framework documentation, to include one or more state diagrams. The authors state that such diagrams could be particularly useful in guiding implementation of operations identified. For example,

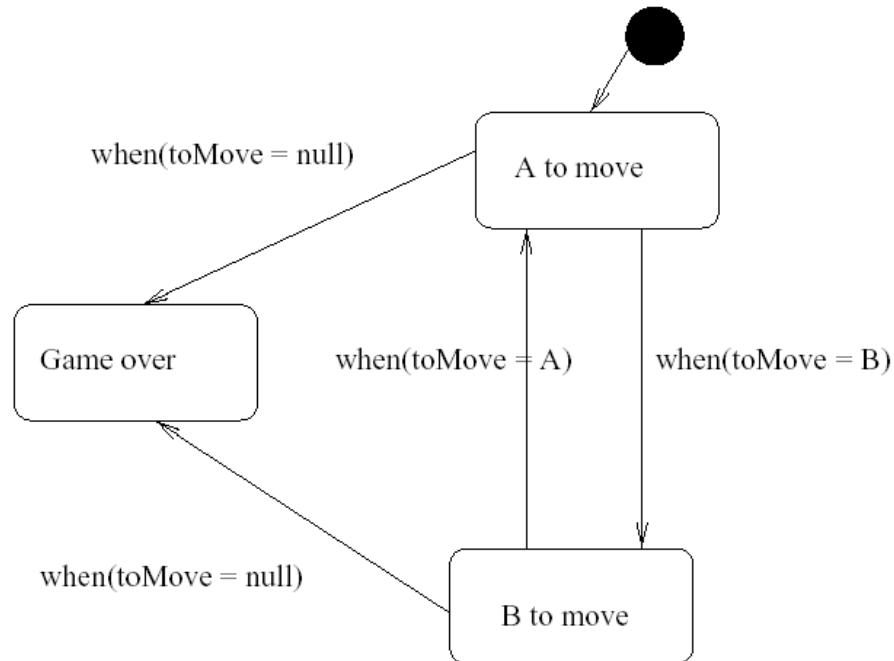


Figure 16.6 State diagram for `CurrentPosition`.

CurrentPosition is considered to have the most complex or interesting behaviour in that its behaviour most clearly varies depending on the history of what has been done to the object so far.