Case studies: Outline

I. Automated Banking System
II. Library
III. Noughts and Crosses

Definition of the problem

1. use case: Make the first move
   a. introducing MVC modelling
2. use case: Make an arbitrary move
   a. a more detailed MVC model
3. Design patterns

Case Study: Noughts and Crosses

Game played on a 3x3 square board between two players. The first player chooses a square on the board and places an X token on it. The other player then chooses an empty square and puts an O token on it. Thereafter, the players alternate placing X and O tokens respectively in empty squares.

The winner is the first player to complete a straight line (horizontal, vertical or diagonal) of three of their tokens. If the board is filled without either player achieving this, the game is drawn.

Design a system to support the game. The system will receive inputs representing moves, will show the current game position and will report when a game has been won or drawn. Once a game is over, the board can be cleared and a new game started.

Where do we start?

Even with a fairly simple system like this, it is far from obvious. We could identify nouns:

game, board, player, square, nought, cross, move, current game position

BUT, that does not seem to get us too far ...

Not too clear what other classes are required and how they relate to each other.

Use cases

Let us therefore try to get a better hold on what is required by considering use cases.

make the first move

A player inputs a position.
The system displays a board with a cross in the chosen square.
The system reports that the game is not finished.

How has this helped us?
Modelling user interaction

It seems to emerge that user interaction, e.g. user's moves, and the rendering of information, e.g. the rendering of the board, play a relevant part in our system.

We discussed earlier the benefits of separation of concerns (modularity) as far as GUI are considered:

concentrate on modelling the underlying software system,

the GUI will be supplied separately.

We would benefit from a framework that can enable us to deal with GUIs, but allows us to keep them separate from the underlying main software application.

GUIs

GUIs can be very complex to program.

Modern OO languages give a lot of support - Visual C++, Delphi, Java.

Advantage: Possible to create GUIs relatively easily and quickly.

Disadvantages: A lot of what is going on is hidden from the user and the level of abstraction is language dependent making it difficult to produce a language independent design.

Business logic can also get tied up in GUI logic, making it difficult to transfer system to new GUIs/methods of interaction.

GUIs

In Java, for example, the event loop is hidden.

To be able to handle events, an object must be declared to be an EventListener

```java
public interface EventListener

The EventListener interface is the primary method for handling events. Users implement the EventListener interface and register their listener on an EventTarget using the AddEventListener method. The users should also remove their EventListener from its EventTarget after they have completed using the listener.

from Java Class Library Documentation
```

and must register itself with the graphical objects that can generate the events in which it is interested.

GUIs

An EventListener object contains the definition of an appropriate event handler.

The actual interaction that takes place as the result of pressing a button is not made explicit.
A framework enabling us to deal with GUIs separately from the underlying main software application should
- define a suitable interface between GUIs and the rest of the system, and
- define the interaction protocol between the various components.

The Model-View-Controller (MVC) framework provides such a neat separation between GUIs and the rest of the system, specifying a clear interaction protocol.

MVC, or architecture first, was developed with Smalltalk, and now is widely used in many GUI environments, e.g. Windows, OS X.

The Model-View-Controller framework consists of three main components:

- **Model**: The underlying system.
- **View**: Graphical representation of the model’s state (i.e. the output from the system).
- **Controller**: Graphical components used to interact with the environment (i.e. the input to the system).

**Use case**

We are going to use the Model-View-Control architecture for our Noughts and Crosses system.

The use case shows us that class Board and Square (and the Cross token if it is a class) are part of the View. The Board view class merely displays the current state. We will rename it BoardView.

The Controller will handle the input of the position.

But what about the Model?
It must keep track of the state of the game, i.e. it must hold the state of the board.

**Scenario**

To help us identify the required objects, let us now sketch out a scenario to see how this behaviour can be achieved by objects sending messages to each other, i.e. let us use a sequence diagram.
Scenario

The Controller receives an input and passes this information to the Model.
As that will cause the Model state to be changed, the Controller then causes the View to be updated.

State information

Remember that the Controller and the View objects have no state information, i.e. no memory (no "permanent" information) except perhaps details like their current position on the screen.

It is the Model object that keeps (ideally) all the internal information about the current state of the game.

Note how the purpose of a View object is simply to obtain information from the Model and decide how that information is to be displayed to the user.

Hence, each Square object has to interrogate the Model (using getVal to determine the value to be displayed).

State information

The Model object has no knowledge of, or control over, how information is to be displayed. It does not even know that there are view objects.

Modularity/separation of concerns: a fundamental concept in computer science.
E.g. updates and changes in the Viewer do not affect the Model.

Dealing even with this very simple behaviour has helped us understand what is required:
- we provide support to user input (Controller activity),
- we maintain a state coherently - e.g. by dealing with concurrent access - (Model activity), and
- we display things (View activity).

Case studies: Outline

I. Automated Banking System
II. Library
III. Noughts and Crosses

Definition of the problem

1. use case: Make the first move
   a. introducing MVC modelling
2. use case: Make an arbitrary move
   a. a more detailed MVC model
3. Design patterns
Scenario

Let us now consider a slightly more complex scenario:

make an arbitrary move

A player inputs a position. The system checks that the position is empty and that the game is not already finished.

The system reports if that is the case and if so, ignores the input.

If the move is valid, the system adds an X or O as appropriate to (the representation of) the board.

The board is then displayed.

The system reports whether the game has been won, drawn or is not yet finished.

Reasoning about this scenario, we note that the system can keep track of whether the next move is an X or an O; it does not have to be part of the user input.

We also note that it is the Model that will check whether or not a move is valid (e.g. if the chosen square is empty).

It now seems reasonable to separate the Model into two objects: MoveHandler to deal with the current move and BoardState to remember the state of the game.

The MoveHandler object will interact with the BoardState object to check that a move is valid as the current state of the board is needed for that.

Let us now represent the scenario where we make a successful arbitrary move as a sequence diagram.

The BoardView object will call each of the 9 Square objects in turn to display each of them. Optimisation: could it call only the updated square?

Yes, as it knows the current move, BUT remember, as a general principle, that it is better to keep BoardView as much decoupled as possible from Model, e.g. better not to duplicate the state representations [BoardView would need one], which then have to be kept consistent.
Class diagram

Three view classes: Reporter, BoardView and Square.
Two model classes: MoveHandler (which is a control object) and BoardState (which is an entity object).

Summary

What have we learnt?

The class structure was not obvious.

Again, use cases and interaction diagrams can be of real help in understanding requirements and identifying classes.

The MVC architecture imposes a standard structure on our solution (for a given class of systems, e.g. useless for systems without a display/control kind of interface).

An alternative architecture would have been for the BoardView and Square classes to have held state information as well as displaying that information.

Case studies: Outline

I. Automated Banking System
II. Library
III. Noughts and Crosses

Definition of the problem

1. use case: Make the first move
   a. introducing MVC modelling
2. use case: Make an arbitrary move
   a. a more detailed MVC model
3. Design patterns

Summary

Must emphasise: there is no such thing as the correct solution; there are lots of possibilities.

Advantage of MVC is that it leads to a separation of concerns; the resulting model should therefore be easier to understand, modify and maintain.