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.

Considerations on 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.

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
{
    // primary methods for handling events
}
```

The actual interaction that takes place as the result of pressing a button is not made explicit.

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

from Java Class Library Documentation
**Model-View-Controller**

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.

**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: fundamental 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
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:**

- 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;
- the **Model** that will check whether or not a move is valid (e.g. if the chosen square is empty);
- the **Model** consists of two objects:
  - MoveHandler to deal with the current move, and
  - BoardState to remember the state of the game.

The MoveHandler interacts with BoardState to check that a move is valid (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, i.e. 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—a control object, and BoardState—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
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. Low level design
4. Design patterns

Low level design

X and O: we have not said anything about how each position is input. That is the best approach during analysis and high level design.

In low level design, we can make a decision.

Two obvious alternatives:

- Have text boxes to receive the required square position.
- Select the required square with the mouse.

In low level design, we may know the target language. For example, if it is Java, theReporter class can be a TextField. [exercise: what is a TextField?]

Low level design

Let us assume that we select the required field with the mouse.

In Java, we can pick up the mousePressed event (in theController), get the co-ordinates of where the mouse was pressed and then query theBoardView object to find the corresponding row and column position of the square.

TheController then sends this information toMoveHandler (theModel) to do internal processing.

As the applet or main class of an application is theController, we can give it a more meaningful name, such asXandO.

Separation of concerns

The relevant part of the sequence diagram is:

Note that theModel is unaffected by how the position is input. It receives the relevant row and column number of theSquare from theController, but does not need to know how that was obtained.
**Separation of concerns**

Review of the advantages of the MVC architecture:

By separating the GUI from the underlying application (the Model), each part can be developed separately.

The application has no knowledge of how it is to be displayed; it could for example have several Views.

One View could present the information in an application as a table while another could present it as a graph.

That choice would be part of the GUI and be independent of the Model.

The purpose of the View is purely to display the data it obtains from the model.

The View does not know what the data means.

---

**Introducing Design Patterns**

The MVC framework is a classic design pattern, i.e. a structure and a kind of interaction, whose relevance goes far beyond the XandO game.

It actually encodes a general schema that can be applied anytime a user interacts (C) with a system updating the information embedded in the state of the system (M), and when a form of presentation of such information through an interface is in place (V).

Following the principle of reuse, it would be wise to be able to precisely describe such a pattern, or better, a well-tested and commonly agreed version of such a pattern, so as to be able

- to easily recognise when it can be applied to a given scenario, and
- to straightforwardly apply it to our system design.

More in general it would be desirable to have a library of such design patterns.

---

**The MVC Design Pattern**

To understand patterns we represent them using class and sequence diagrams.

The class diagram on this slide represents the support to the an MVC pattern, from a Java perspective.

The abstract superclass Observable and the interface Observer are defined in the Java library.

http://download.oracle.com/javase/6/docs/api/java/util/Observable.html
The MVC Design Pattern

Let us look at the main components of this pattern.

Observable is an abstract superclass that has the methods

- + addObserver
- - notifyObservers
- - setChanged

and maintains a list of its registered observers.

Observer is an interface offering the operation update.

Model is a concrete class that maintains the state that is of interest to its observers. It inherits the methods addObserver, notifyObservers and setChanged.

When an object of class Model is modified, it calls its inherited setChanged and notifyObservers methods.

When we write the Model subclass all we need to do is add a call of setChanged and notifyObservers, specifically for the case at hand.

View is a concrete class that implements the Observer interface by defining an update method.

It registers with a Model object by calling its addObserver method and when it subsequently receives a call of its update service, it goes and gets new information from the Model object.

Note that all the work of maintaining the list of observers and notifying them is defined in the Observable superclass.

A Model class can be a subclass of the abstract class Observable.

A View class can register itself with an observable model component.

When the observable Model class changes, it can call notifyObservers to inform the registered View classes that a change has occurred.

The View classes can then redraw their display.

Note that this is very language dependent with lots occurring under the bonnet. It is therefore very difficult to model in a language independent way as we do not know how much we are going to get for free from a language.

This is a good argument for separation of Model from the View and Controller; we can have one Model and several sets of Views and Controllers for different languages.
The MVC Design Pattern

For our Noughts and Crosses program, the class diagram is:

- **Observable**
  - notifyObservers(void)
  - addObserver(void)
  - setChanged(void)

- **Observer**
  - update(void)

- **BoardState**
  - addObserver(void)
  - addObserver(void)
  - setChanged(void)

- **BoardView**
  - update(void)

The class diagram shows the static structure. To understand behaviour, we need to think in terms of objects and deal with sequence diagrams.

The first thing to be clear about is that there are only two objects:

- **BoardState** that offers
  - the public operations addObserver and getVal and
  - the private operations notifyObservers and setChanged

- **BoardView** that offers
  - the public operation update.

There are two different parts to the behaviour:

**Registration**: The BoardView object registers with the BoardState object by calling the addObserver() operation of BoardState.

**Updating**: When the state of the board changes, the BoardState object calls its own private operations:
- setChanged();
- notifyObservers();

Execution of notifyObservers() by BoardState causes the update() operation of BoardView to be called.

Execution of update() by BoardView causes a callback (e.g., a call of getVal()) to get the required new information from BoardState.

Such flow of control is summarised by the following sequence diagrams: