Model View Controller meets Monad

Gianluca Aguzzi, 2021

“ ***It is all about composition*** ”

Goals

- show an end-to-end **functional** application
- leverage some well-consolidated functional libraries
- understand limitations (if any) and the improvements

Target Application

Tic Tac Toe

OOP Design

Everything is an object

Clean interface, state incapsulated, side effect as methods call.

Let's try to build an *old-fashion* application 😄

Model

```
/* two players (X, O)  */
enum Player {
    X, O, None;
}
/* a board 3x3 */
interface TicTacToe {
    Player get(int X, int Y);
    TicTacToe (or void??) update(int x, int y, Player p);
    boolean isOver;
    Player getTurn;
}
```

View

```
//a là view model?
interface ViewBoard {
    List<String> getRow(int row);
    List<List<String>> getAllBoard();
}

interface View extends ClickCellSource {
    void render(ViewBoard board);
    void winner(String player);
}
```

Put some design pattern 😄

```
public interface ClickCellSource {
    void attach(Observer observer);
    interface Observer {
        void notify(int X, int Y);
    }
}
```

Controller

```
public interface Game extends ClickCellSource.Observer {
    void start();
}

public class TicTacToeGame implements Game {
    private final TicTacToe ticTacToe;
    private final TicTacToeView ticTacToeView;

    public static TicTacToeGame playWith(
        final TicTacToe ticTacToe,
        final TicTacToeView ticTacToeView) {...}

    ....
```

Putting all togheter

```
public static void main(String[] args) {
    final TicTacToeView view = SwingView.createAndShow();
    final TicTacToe model = TicTacToeFactory.empty();
    final Game game = TicTacToeGame.playWith(model, view);
    game.start();
}
```

Clean enoght right?

What do you think?

Task

> Task represents a specification for a possibly lazy or asynchronous computation, which when executed will produce an A as a result, along with possible side-effects.

What does it refer you to?

```
trait Task[+A] {
    final def flatMap[B](f: A => Task[B]): Task[B] = ...
    final def map[B](f : A => B): Task[B] = ...
    //some interesting extesions
    def memoize: Task[A] = ...
}
object Task {
    def pure[A](a : A) : Task[A]
    def defer[A](a : Task[A]) : Task[A]
}
```

A Little taste

```
object App extends TaskApp {
    def
}
```

Observable

> a data type for modelling and processing asynchronous and reactive streaming of events with non-blocking back-pressure.

We use it to implement the Functional Reactive Programming

Books

1. **Scala with Cats Book**
2. **Category Theory for Programmers**
3. **Functional Reactive Programming**

References