

Basuke's Blog

現場のコード

現場のコード : 最終回 MVCを理解してテスト可能なコードへ

2012/12/31 | BASUKE | コメントをどうぞ



2011年にMOSAのMOSADENコーナーに連載させてもらった「現場のコード」第四回にして最終回です。結局ここが書きたかったというMVCのお話。CocoaのMVCを実践していくために必要な考え方を話しております。

しかし、1年前の今日、2011年の大晦日にドタバタと公開してますな。まったくもってMOSA関係者の皆様には、原稿が遅くて本当にご迷惑をおかけしました。ごめんなさい。連載の機会をいただきありがとうございました。

初出：2011年12月31日

前回からずいぶんと時間が空いてしまいました。前回はテストできるコードの条件について考えたところで、では実際にテストしていくためには、どのようなコードを書いたら良いのか考えていきます。テストできるコードを書く上で何より大事なことは、まずMVCという考え方を身につけて実践していくことだと僕は考えてます。

MVCがテストへの道筋

MVCという考え方とは、コードをモデル（M）、ビュー（V）、コントローラ（C）の三つの要素に分解していくことで、オブジェクト指向プログラミングにおいて広く支持される設計手法として知られています。当然ながら単位はオブジェクト、このオブジェクトはMなのか、Vなのか、Cなのかという役割分担がされることになります。Cocoaのプログラミングにおいて、よいコードを書きたいのであれば、このMVCというデザインパターンにまず準拠すべきです。これはアップルの公式ドキュメントでも明確に規定されています。

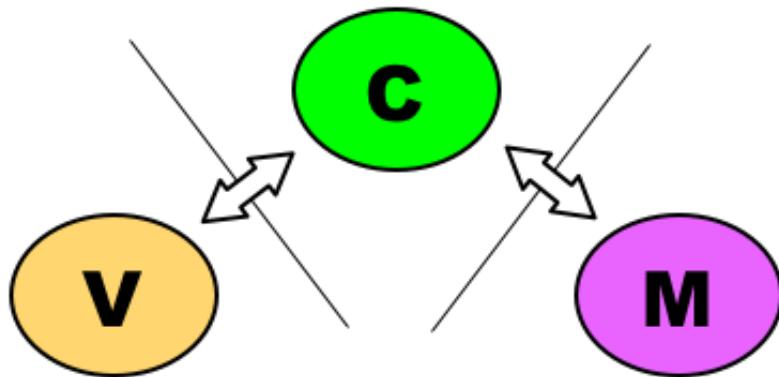
The Model-View-Controller Design Pattern

MVCというのは、もともとSmalltalkで生まれた設計手法で、それが広くいろいろな環境に適用されていきました。とくに、インターネットの普及とともに広がったJavaの世界では、サーバー側のプログラムを構成する上で重要な概念として定着しています。その流れで、他言語のウェブプログラマの間でも、当然の技法というぐらいに定着しています。

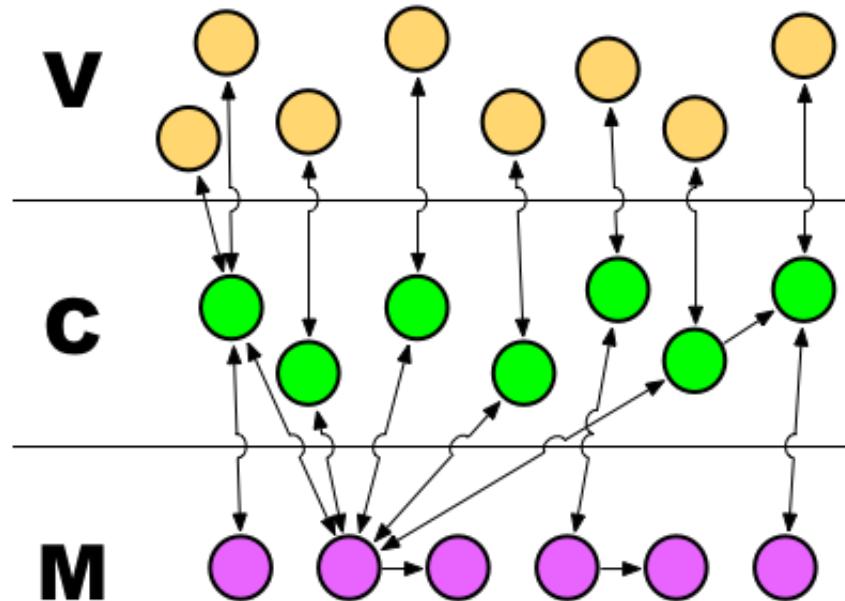
Mac OS XやiOSで当たり前に使っているObjective-Cも、元々はSmalltalkのオブジェクト指向を素直にCに適用しようとした、いわば直径の子孫と言えます。ということで、そのObjective-CのCocoaでも、MVCは大事な役割としてフレームワークの思想を支えているのです。

そうした、歴史あるMVCという考え方、当然ながら環境によってずいぶんと解釈というか実装方法が違っているのです。ウェブのサーバーの世界で使われるMVCは、そのなかでもかなり特異な方向に進化（？）します。そして同じデスクトップ環境でも、SmalltalkのMVCとCocoaのMVCでも違いがあります。正直なところ、Smalltalkについてびしっと言えるほど知識が無いので、いい加減なことを言ってしまいそうです。ここではCocoaのMVCについて話すということで、Smalltalk方面な方からの突っ込みはお手柔らかにお願いします。

さて、MVCと言えばよく以下の図を見るかと思います。



シンプルですね。オブジェクトはMかVかCのどちらの領域に属することになります。この図ではそれぞれ一つずつの円がいかにもオブジェクトづらしてますが、これはもちろん簡略化した概念図です。実際にはたくさんのオブジェクトの集合体になります。より現実に近づけて描くと以下のようにになります。



ポイントは二つ、当然ながらたくさんのオブジェクトがあることと、MVCそれぞれのレイヤーに属しているということです。レイヤーとして捉えることで、その境界をまたぐ必要がある、ということを意識することが大事です。

モデル（M）は、データでありロジックであり、アプリケーションが本質的に役に立つための全てのものがモデルです。よくデータ構造のことをモデルと勘違いしている人を見かけますが、単にデータ構造を記述するだけでなく、計算方法や通信などのロジックも含めて、あらゆるプログラムの構成要素がモデルになります。逆説的ですが、以下のVでもCでも無いものがモデルとも言えます。

ビュー（V）は、実際に画面に表示されるものを表します。ビューは自分自身がどのように画面上に表示されるかの実装方法を知っています。また、ビューはユーザーからの入力に対する反応をする役目も担ってます。

コントローラ（C）は、この二つをつなぐものです。単純に1:1:1ではなく、上の図のように一つのコントローラに対して複数のビューやたくさんのモデルがつながることが普通です。コントローラの担当する役割は、MとVの間をつなぐこと、基本それだけです。それ以外に関して、あまり責任を負うべきではありません。この辺については後述します。

ちなみに、オリジナルのSmalltalkではMとVはもう少し密接なつながりがありますので、MからVに線が引かれていることが普通ですが、Cocoaにおいては必ずCを間に挟むことが推奨されています。

再利用について

オブジェクト指向やMVCの話でよく出てくるキーワードが「再利用」です。一度書いたコードをもう一度使えるかどうかという議論で、オブジェクト指向で書かれたものは再利用

しやすいという文脈で語られます。MVCの考え方で設計を行なっていくと再利用することができる場合が多いです。

実際には再利用にも大きく三つのケースがあると思います。一つ目はオブジェクトの再利用（共有）、二つ目は同じアプリの別バリエーションでの再利用、三つ目がプロジェクトを超えた、いわゆるソースコードの再利用です。この三つのパターンにおいてMVCの各レイヤーの再利用がどれにあたるのかを考えます。

ビューに関しては、大半はシステムが事前に用意したパートの組み合わせで済んでしまいます。ビューをコーディングするケースとして一つ考えられるのは、アプリ全体のUITemaを変更したいという場合です。その場合には、システムが用意するパートと似た見た目、似た動作をするビューを作ることになります。この場合には再利用できるコードを書くことになります。

一方、モデルの多くは比較的簡単に再利用できます。汎用的な計算ロジック、通信方法、データベースへの永続化、暗号に関する手法など、こうしたものはアプリを構築する上のインフラ的な役割をするもので、モデルの中でも再利用できる種別のものと言えます。

これら、共通パートのビューとインフラを支えるモデル達を再利用する場合、すでに別のプロジェクトで実績をあげているものなので安心して使えます。再利用することでさらに信頼性を増していきます。積極的にライブラリにまとめていきたい部分です。三つのパターンで言えばプロジェクトを超えた再利用にあたるものと言えます。

それに対して、コントローラはたいていの場合は再利用しません。ビューとモデルが違っていればコントローラは新規に作ることになるでしょう。再利用する価値がほぼないです。

アプリケーション独自のビューというのも微妙です。アプリ内部で複数の場所で使う、ということはあるでしょうが、固有の表現方法、固有の内容を扱っているので、別のアプリでそのまま使えることはほぼ無いでしょう。

同様に、アプリケーション独自のモデルというのも再利用は難しいです。これこそがアプリの中枢な訳で、アプリが違えばそこで使うロジックも違ってきます。再利用できるケースは少ないです。ただ、こうしたモデルは、アプリの中では複数の場所から共有されることになります。オブジェクトの共有という形での再利用です。単純なリファクタリングによって、コードから重複部分がまとめられた、というような話はこのレベルの再利用を意味します。

別の見方をすると、モデルはバリエーションを超えて再利用され得るものです。例えばiPad版を新たに作ろうと言うことになった場合には、UIは基本的に作り直しなので、それ

に伴いコントローラの多くも新しく組むことになるでしょう。でも、iPhoneでもiPadでも本質的には同じ目的で動くものです。なのでモデルに関しては共通で使えないといけないはず、という理屈になります。つまり、iPad版を作ることを考えたときに共通して使えるようなオブジェクトのクラスは、少なくともモデルであるとも言えます。もしコントローラにそういうロジックが入り込んでいたら、モデルとして切り出してみましょう。

なお、再利用というのは名前の通り、既存のものを再度利用することであり、いずれ使われるかもしれないものを見越して、再利用を前提とした設計を行うというのはちょっと間違っています。MVCに従ってオブジェクトを構成していくば、いずれ結果的に再利用できるコードが見つかりやすいだろう、ぐらいのものとして考えておく方が健全です。いつか使われるかもしれない機能を潜り込ませてしまうと、逆に拡張するときの足手まといになってしまふことがあります。気をつけましょう。

MVC間の主従関係

オブジェクトの主従関係というと、誰が何を作るかです。一応ちゃんと決まっています。基本、ビューはコントローラが作ります。モデルはコントローラ、もしくは別のモデルによって作られます。ではコントローラは誰が作るのか。コントローラは別のコントローラによって作られます。

では最初のコントローラは誰かというと、AppDelegateになります。AppDelegateはアプリケーションレベルでのコントローラです。コントローラといえばビューコントローラのことと勘違いしがちですが、それ以外にもいろんな形でコントローラが存在できるのです。

作る方がオブジェクトのオーナーシップを持ち、作られたオブジェクトは作った側のことを知らないのが普通です。

一つのコントローラが複数のビューへの参照を持つことはよくあることですが、作られたビューが他のコントローラと共有されるということはまれです。ビューはコントローラによって作られ、コントローラに占有される形になります。

一方、モデルは複数のコントローラ間で共有されます。共有されなくてはいけないということは無いのですが、前の画面と次の画面で同じモデルを共有する場合や、いろんな場所から同じロジックを呼び出すなどの使われ方が普通です。

MVCの層を超えたやり取り

みんなそれぞれオブジェクトなので、メッセージを送信しながらプログラムの目的を実現していくことになります。このとき、各レイヤーをまたいだやり取りにはパターンがあります。

まずはビューとコントローラの間です。ビューに表示される内容を準備するのはコントローラの役割です。そのため、コントローラはビューの生成時に、ビューへの参照を内部に持ち、それをたどって値をビューに設定していきます。IBOutletで象徴される役割です。

逆にビューは明示的にコントローラのことを知っていてはいけません。ビューからコントローラへは、コントローラが自ら設定したデリゲートやデータソース、ターゲット・アクションによって情報がやり取りされます。デリゲートを設定するのはコントローラでなくてはいけません。

次はコントローラとモデルの間です。まずコントローラからモデルへのやり取りに関しては、モデルを使うと決めたコントローラが参照を持つことになりますので、普通にモデルを利用できます。逆にモデルはコントローラのことを深く知っていてはいけません。ビューのときと同じです。モデルもコントローラと深く結びつくことは良いことではないのです。基本はコントローラからモデルを利用する方向です。

とはいっても、モデル側で何か変更があったことを伝えないと話が始まらないことが多いです。先に述べた通りモデルは複数のコントローラによって共有されることが多いので、デリゲートは使いづらいです。そこで、モデルからの変更をノーティフィケーションを使って通知を受けたり、キー値監視（Key Value Observing、KVO）を使って監視する手法が使われます。単独で所有されるモデルであればデリゲートもあり得ます。Core LocationのCLLocationManagerなどは、モデルでありながらデリゲートを採用しています（だから使いにくいのですが）。

コントローラ同士でやり取りを行うことは少ないですが、コントローラの生成や破棄は別のコントローラが担当するのでその場合にはコントローラ間でやり取りが発生します。身近な例では、presentModalViewController:animated:で別のビューをモーダル表示する場合などです。この場合には開いたビューからDoneボタンで戻ってこないといけなかつたりします。このためやり取りのためのプロトコルを定義してデリゲートを設定させます。コントローラも基本的には作った別のコントローラの単独所有物になりますので、デリゲートで問題ありません。

モデル間のやり取りはもう少しいろいろあります。モデルが別のモデルを使うことも多々あるからです。ここは柔軟にいろんな方法を試すことが出来ます。ただし、一つだけやってはいけないことは双方向の依存関係を作ってしまいますことです。双方が相手のことを深く知りていなくてはいけない関係だと、主従関係がはっきりせずに生成破棄のタイミングも曖昧になります。ここまで見てきたCとV、CとMの関係のように、片方は相手のことを良く知っているけど、相手は自分のことをよく知らない状態にしておくことが大事です。そうしておけば、依存関係が減り関係が一般化してきます。抽象度も高まり再利用の可能性も上がります。

テストのためにMVCを導入する意味

MVCについてだいぶ話しましたので、この辺りでテストに話を戻しましょう。すばり、MVCの導入の意味は、この考え方従ってどんどんモデルに切り出してしまいましょう、ということにつきます。モデルへ切り出せば、ユーザーとのインターフェースやビューへの反映などを気にすること無く単体テストが実行可能になります。

ここまでルールに従っておけば、モデル層に居るオブジェクトはコントローラ、もしくは他のモデルから参照されるものとなっているはずです。コントローラへの強い参照は持っていない。なので単独で取り出しても問題なく使える状態になっているはずです。他のモデルとの依存関係があったとしても、そのモデルごとテストするなど、方法は見つかるはずです。分離すればテストできるのです。

ではビューとコントローラはどうするのか？残念ながらこの二つは単体テストではうまいことテストできません。ビューはシステムとの依存関係も強く持ちますし、見た目に関しての話はもともと単体テストには向きません。イベント処理なども難しいです。コントローラは、アプリが実行しているときの外部の環境に非常に近くに存在するため、テストは難しいことが多いです。

コントローラを太らせない

そもそも、テストしづらいと言う性格を持つコントローラですから、コントローラには極力ロジックを置かないことが大事になります。もちろん、新たなデータ構造を定義することもしない方が得策です。

モデルにコードを移すというのは、つまりはこういうことです。コントローラに書いている多少なり複雑な操作を、最低限必要なメンバー変数をもった新たなクラスを作り、処理ごと移動させるのです。例えば、テーブルビューコントローラで使うデータを直接配列で管理しているような部分があったら、それを配列での直接管理から、自前のコレクションオブジェクトに移動させるのです。実質的な違いは無いように感じるかもしれません、このコレクションオブジェクトは単体でテストできるようになります。コントローラを初期化してテーブルビューのデータソースとして動作する部分に必要な処理を、直接呼び出すテストをすれば良くなります。例えばフィルター機能が後で必要になったときでも、コレクションオブジェクトの機能拡張と見た目などを分離して勧められるので安定性の面でも有利に働きます。何よりクラスが单機能になっていくので、コードがシンプルになります。テストできる以上に、コードが整理されていくという大事なメリットもついてくるのです。

クラスが増えることを恐れない

心理面に訴えかけることなのでしょうか、なぜかクラスが増えることをいやがる人が多い

ように思います。断言しますが、巨大な小数のクラスと、小さいたくさんのクラスがあるとして、同じ規模のアプリケーションを書いたとすると、確実に後者の方が見通しがよくなり、たいていの場合はコード量もトータルで減ると思います。

でも、クラスが増えることに伴うある大きな問題点についてはわかる気もします。それは、クラスに名前を付けなくてはいけないことです。新しいクラスを定義する訳ですから当たり前なのですが、名前、特に英語でつけないといけないというのはとてもプレッシャーになります。このプレッシャーから、別のクラスに分割することをせずに、同じコントローラの中にメソッドを書いてしまうことになるのでしょうか。

でも、コントローラに書いていっても今度はメソッド名で苦労するはずです。いずれは名前も尽きてメソッド名をつけるのに苦労するはめになります。ここはがんばって名前を付ける訓練をしていきましょう。クラス名は、最初の頃は「名詞+動詞+er」でつけていくのが簡単です。例えばTokenizer、XMLParser、など汎用的なものから、CreditBalancerとかFractalDrawerなどが考えられます。動詞の語彙を増やしていくべきは後々便利に使えますので試してみてください。フレームワークのクラス名のつけ方なども参考にしてみるといいかと思います。

ビューコントローラを分割して入れ子にする

ビューコントローラが太りだす原因には、デリゲートが複雑になりすぎるというケースが考えられます。例えば、テーブルビューが二つあるような場合を考えてみましょう。テーブルビューのデータソースのメソッドで、どちらのテーブルビューのための処理かどうかを判定したりしていませんか？コントローラのコードの中にif-else文があるようだと、結構怪しいです。if文ならまだ問題ないですが、if-else文は二種類のことを一つのクラスで行っている匂いがします。そういうコードを書いている場合、单一のコントローラとしては大きくなりすぎと考えて、二つの別のビューコントローラに分割して、それぞれ担当を分けてあげるという方法を試してみてください。場合によってはきれいに分割できます。コントローラ間の依存関係が強くなってしまうのが気持ち悪いですが、コントローラですので、すっきりとしていることの方が大事です。

MVCを徹底させてテスト可能に

コードの質を上げるためにテスト可能にする。そのためにはモデルにコードを移していくてテスト可能な部分を増やす。そうすることで、アプリケーション全体のバグは減り、本来の機能テストに時間が充てられるようになります。コードはMVCを意識して書かれているので、適切な分離度になります。単体テストが書いてあれば、リファクタリングがいつでも出来ます。コードの質をさらに高めていくことが出来ます。こうして良い開発サイクルが出来上がっていくということになり、安心して開発に取り組んでいくことが出来るようになるという訳です。