Martin Fowler — GUI Architectures. Часть 2 / Хабр

Всем привет еще раз. Это опять я. Внутри хабраката перевод еще одного параграфа статьи Мартина Фаулера.

На этот раз затронута тема MVC. Фаулер весьма популярно про него написал. Я постарался популярно перевести:) Теперь можно понять, почему все носятся с MVC, как с писанной торбой. И, кстати, Фаулер прав — очень много где и много кто воспринимают MVC по-своему. Сам Фаулер пишет про оригинальный MVC, который работал на платформе Smalltalk. Очень познавательно.

Предыдущая часть тут[1]. Оригинал статьи тут[2]. Очень желательно прочитать первую часть, потому как Фаулер там определил общую задачу-пример, которую он решает при помощи описываемых архитектур. Если не прочитать про эту задачу, будет немного не ясно, про что речь.

Следующую часть перевода напишу, когда разозлюсь и возьму себя в руки.

Model-View-Controller. Пожалуй, самый распространенный шаблон в разработке UI — Модель-Представление-Контроллер (MVC). К тому же, чаще всего самый неправильно понимаемый. Я уже потерял счет сколько раз я видел что-то, описываемое как MVC, которое им не оказывалось. Откровенно говоря, это происходит потому, что в наши дни классический MVC уже не подходит для толстых клиентов. Посмотрим, откуда произошел MVC.

Перед тем, как мы углубимся в MVC, важно помнить, что он был одной из первых попыток серьезно подойти к решению UI-задач независимо от масштаба решения. В 70-ые годы понятие «Графический пользовательский интерфейс» было не очень популярным. Шаблон «Формы и элементы», которые я описал выше, появился позже, чем MVC. Описал же я его в первую очередь потому, что он проще, причем, не всегда в хорошем смысле этого слова. Я буду рассматривать Smalltalk 80 MVC на уже известном примере про концентрацию мороженого. При этом я позволю себе несколько вольностей и дам некоторое описание платформы Smalltalk 80. Для начала скажу, что система была монохромной.

Сердцевиной идеи MVC, как и основной идеей для всех последующих каркасов, является то, что я называю «отделенное представление» (Separated Presentation[3]). Смысл отделенного представления в том, чтобы провести четкую границу между доменными объектами, которые отражают наш реальный мир, и объектами представления, которыми являются GUI-элементы на экране. Доменные объекты должны быть полностью независимы и работать без ссылок на представление, они должны обладать возможностью поддерживать (support) множественные представления, возможно даже одновременно. Этот подход, кстати, так же был одним из важных аспектов Unix-культуры, позволяющий даже сегодня работать во множестве приложений как через командную строку, так и через графический интерфейс (одновременно).

В MVC под моделью понимается доменный объект. Доменные объекты совсем никак не относятся к UI. Для того, чтобы уложить наш пример с концентрацией и замерами в MVC, в качестве модели возьмем объект «замер» (reading), у которого есть все интересующие нас поля. (Как видно, присутствие элемента списка (list box) должно немного усложнить модель, но этот нюанс мы некоторое время проигнорируем).

В MVC я обращаюсь с обычными объектами доменной модели (Domain Model[4]), в отличие от «форм и элементов», где я работал с набором записей (Record Set[5]). Это одно из отличий между этими двумя шаблонами, обусловленное их дизайном. Предполагается, что в «формах и элементах» большинство людей хочет легко манипулировать данными из реляционной БД, в то же время как MVC предполагает работу с обычными Smalltalk объектами.

Презентационная часть MVC создана из двух элементов: представления и контроллера. Работа контроллера состоит в том, чтобы забрать введенные пользователем входящие данные и сообразить, что с ними сделать.

Тут мне следует подчеркнуть, что в нашем примере представление и контроллер существуют не в единственном числе. Мы должны иметь пару представление-контроллер для каждого элемента на экране, для каждого элемента управления на экране и для самого экрана в целом. Поэтому, в нашем примере, первой реакцией на ввод данных пользователем является совместная работа контроллеров, определяющая, какой элемент сейчас эти данные получает. Таким элементом будет являться текстовое поле фактической концентрации. Обрабатывать введенные данные будет контроллер этого текстового поля.

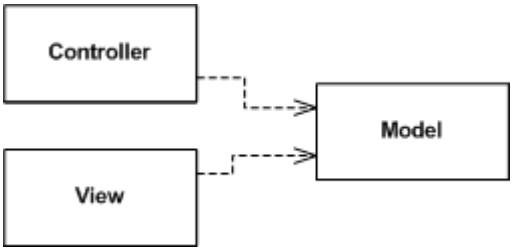


Рисунок 4. Основные связи между моделью, представлением и контроллером. (Я называю их основными, потому что на самом деле представление и контроллер могут быть связанными друг с другом непосредственно. Однако, разработчики в основном не используют эту связь.)

Как и последующие среды разработки, платформа Smalltalk предоставляла возможность создавать обобщенные UI-компоненты, которые можно использовать повторно. В нашем случае, таким компонентом будет пара представление-контроллер. Оба являются обобщенными (generic) классами (*зависят от т.н. конфигурации — прим. перев.*). Обобщение нужно для того, чтобы их можно было использовать в разных приложениях. Первым вводится представление оценки (assessment view), которое будет исполнять роль всего экрана и определять местоположение более простых элементов. Такой порядок похож на тот, что был в «формах и элементах». Однако, в отличие от объекта формы в «формах и элементах», MVC не имеет обработчиков событий текстовых полей в контроллере экрана (assessment controller).

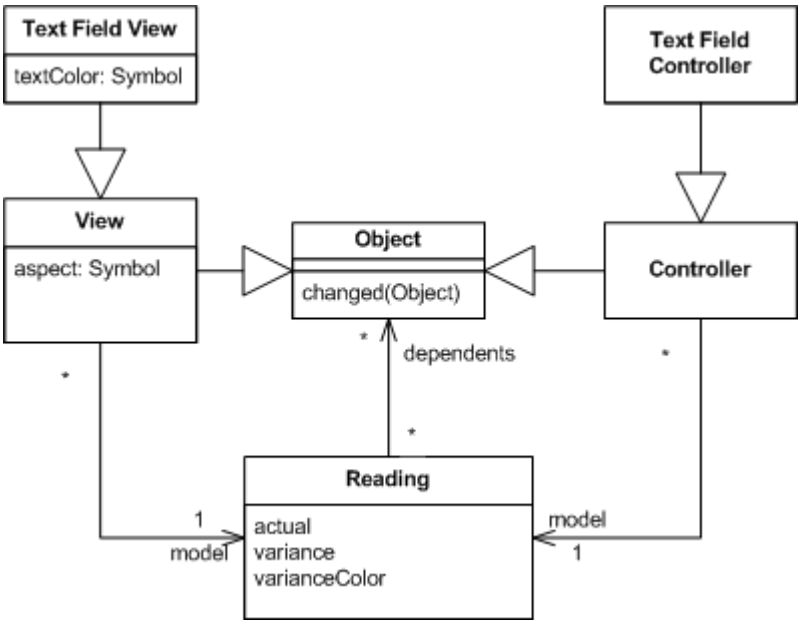
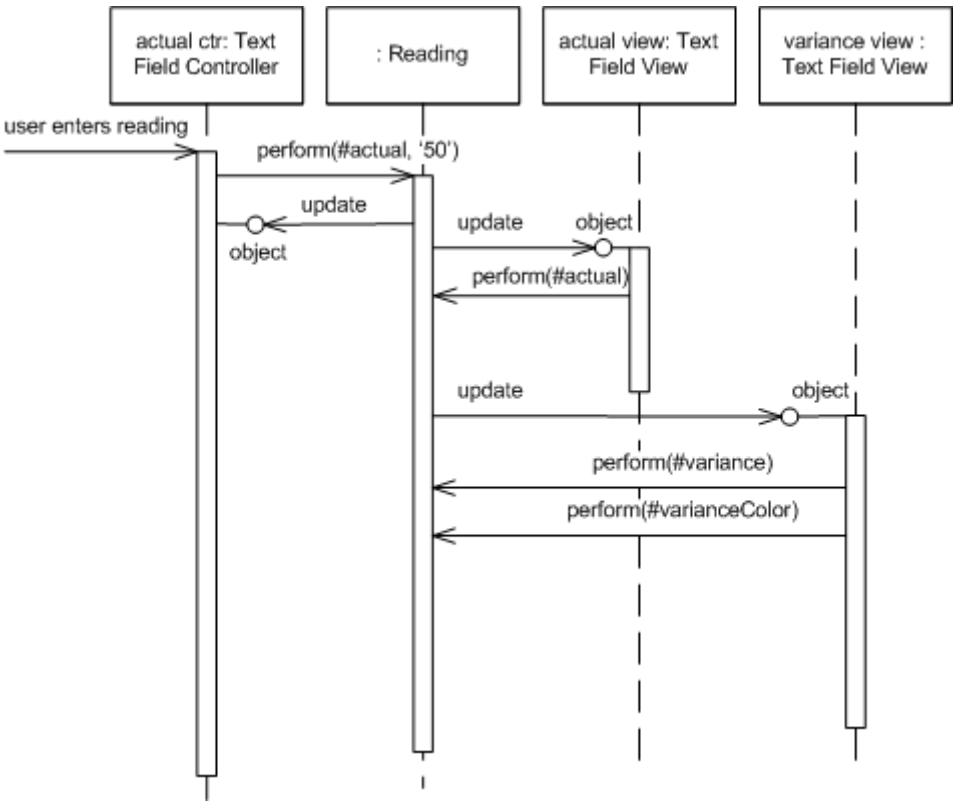


Рисунок 5. Классы MVC-версии для примера с концентрацией мороженого.

Рассмотрим прецедент ввода фактического значения. Конфигурация элемента текстового поля фактического значения состоит из его связи с моделью (объект замера), т.е. метода, который необходимо вызвать у модели, когда у элемента поменяется текст. После инициализации экрана выбранный метод будет равен "#actual" (символ '#' в начале строки обозначает интернированную строку). Как только текст введен, контроллер делает рефлексивный вызов указанного метода у объекта замера (меняет значение поля actual). По сути, точно такой же механизм происходил в привязке данных (Data Binding[6]). Элемент управления связан с нижележащим объектом (записью) и знает, каким методом (т.е. колонкой) он управляет.

Рисунок 6. Изменение поля фактического значения



Итак, в получившемся решении нет никаких общих обозревающих объектов (вспомним объект формы с обработчиками событий). Вместо этого, элементы управления обозревают модель, поведение которой определяется в ней же (а не в объекте формы). То есть, когда требуется определить значение разницы показателей (variance), модель справляется с этим сама.

Обозреватели встречаются в MVC. По правде, это одна из идей, порожденная MVC. В нашем случае все представления и контроллеры обозревают модель. Когда модель изменяется, реагируют представления. Представление текстового поле фактического значения получает сведения, что объект замера изменился, после чего оно вызывает метод модели, определенный как аспект для данного текстового поля — в нашем случае #actual — и выставляет свое значение в поле. (Почти точно так же происходит для цвета текстового поля, но там свои «призраки», до которых я скоро доберусь.)

Обратите внимание, что непосредственно контроллер текстового поля не выставляет значение в представление текстового поля. Он обновляет модель и позволяет механизмам обозревателя позаботиться об обновлениях других объектов. Этот подход отличается от того, что был в «формах и элементах», где обновление объекта формы обновляло элемент управления, что в свою очередь обновляло нижележащий набор записей (record-set) через привязку данных. Эти два подхода я выделяю в шаблоны и называю соответственно: потоковая синхронизация (Flow Synchronization[7]) и синхронизация через обозреватель (Observer Synchronization[8]). Они описывают разные способы синхронизации между состоянием экрана (**screen state**) и состоянием сессии (**session state**). «Формы и элементы» осуществляют синхронизацию данных через поток обращений приложения к разным элементам управления, которые нужно обновлять непосредственно через прямое обращение к ним. MVC осуществляет синхронизацию через обновления модели и затем полагается на обновление представлений элементов управления через их связи с обозревателем.

Потоковая синхронизация становится более явной, когда в приложение отсутствует привязка данных. В таком случае, приложение должно явно провести синхронизацию на одном из своих важных этапов (application flow) — например, при открытии экрана или по событию нажатия на кнопку «сохранить».

Одним из последствий синхронизации через обозреватель (Observer Synchronization[9]) является то, что контроллер не знает об изменениях какого-либо другого элемента, если вдруг пользователь как-то его (т.е. другой элемент) поменял. В то время, как объекту формы в «Формах и элементах» нужно внимательно следить за целостностью состояния экрана, когда

что-то меняется (что может быть очень трудоемко при сложной компоновке формы), контроллеру нет нужды делать то же самое.

Это полезное преимущество особенно удобно, когда открыты несколько экранов, которые обозревают одни и те же объекты модели. Классическим пример MVC — таблица, например экран с данными, плюс пара окон различных графиков этих данных. Окну с таблицей не нужно знать о том, что открыты другие окна. Оно просто обновляет модель, а синхронизация (Observer Synchronization[10]) делает все остальное. С потоковой синхронизацией (Flow Synchronization[11]) потребовалось бы каким-то образом знать, какие другие окна открыты, чтобы произвести в них обновления.

При всех преимуществах синхронизации через обозреватель (Observer Synchronization[12]), у нее есть один недостаток. Проблемой синхронизации Observer Synchronization[13] является сам обозреватель — посмотрев на код, вы не сможете сказать, что происходит. Когда я разбирался в примерах, реализованных на Smalltalk 80, я очень четко вспомнил про это неудобство. Я могу знать, что происходит в участке какого-то кода, но как только вступает в дело обозреватель, мне требуется трэйсер и дебаггер, чтобы понять, что делается дальше. Поведение обозревателя трудно понять и дебажить, потому что оно неявное.

Различные подходы к синхронизации состояний можно пронаблюдать в приведенных диаграммах. Однако, самым важным и влиятельным отличием MVC является его использование отделенного представления (Separated Presentation[14]). Расчет отклонения между фактическим и целевым значениями должно быть поведением доменной модели и не зависеть от UI. Поэтому мы реализуем это поведение в доменном слое системы — того слоя, который представляет собой объект замера (reading). Метод вычисления отклонения является абсолютно логичным для объекта замера и не имеет никакого понятия о пользовательском интерфейсе.

Сейчас, однако, мы можем рассмотреть пару усложнений текущего положения вещей. Есть две неудобные области, через которые я проскочил, рассказывая про теорию MVC. Первая проблемная область — назначение цвета текстовому полю разницы. Эта логика не совсем принадлежит доменному объекту, поскольку цвет, которым мы подсвечиваем значение отклонения, не является его частью. Тем не менее, какая-то часть такой логики все-таки является его частью. Например, оценка качества отклонения: хорошее (больше пяти процентов), плохая (меньше десяти процентов) и нормальное (все остальные значения). Вычисление оценки — это доменная логика. Раскраска текстового поля — это логика представления. Проблема заключается в том, где поместить логику представления, т.к. она не является частью логики нашего обычного текстового поля.

С этой проблемой столкнулись первые разработчики на Smalltalk, они же предложили решения этой проблемы. Решение, которое показано выше — оно грязное, поскольку она нарушает «чистоту» доменной логики. Я признаюсь в случайном акте «загрязнения» и постараюсь, чтобы оно не вошло в привычку (*это Фаулер так шутит — прим. перев.*).

Еще мы можем решить эту проблему так же, как «Формах и элементах» — пусть экран оценки обозревает представление текстового поля отклонения. Когда его значение изменится, экран отреагирует и назначит цвет текстовому полю. Но такое решение вовлекает использование механизмов обозревателя — а чем больше в них копаться, тем больше (экспоненцально) усложняется работа с ними, так же увеличивается число лишних связей между различными представлениями.

Я бы предпочел разработать новый тип UI-элемента. Характерной чертой поведения этого элемента будет то, что он запрашивает оценку у доменного объекта, сравнивает ее с внутренней таблицей значений и цветов и подсвечивает ее полученным из таблицы цветом. И таблица, и запрос значения у доменного объекта будут назначаться представлением оценки на этапе сборки самого себя. Точно таким же образом назначается аспект значения для текстового поля, чтобы оно следило за ним. Этот подход будет работать очень хорошо, если у меня будет легкий способ наследования от текстового поля (в нем я добавлю новое поведение). Есть ли

такой способ — зависит от того, как хорошо разработанные компоненты поддерживают наследование — в Smalltalk это очень легко, а в других средах может быть посложнее.

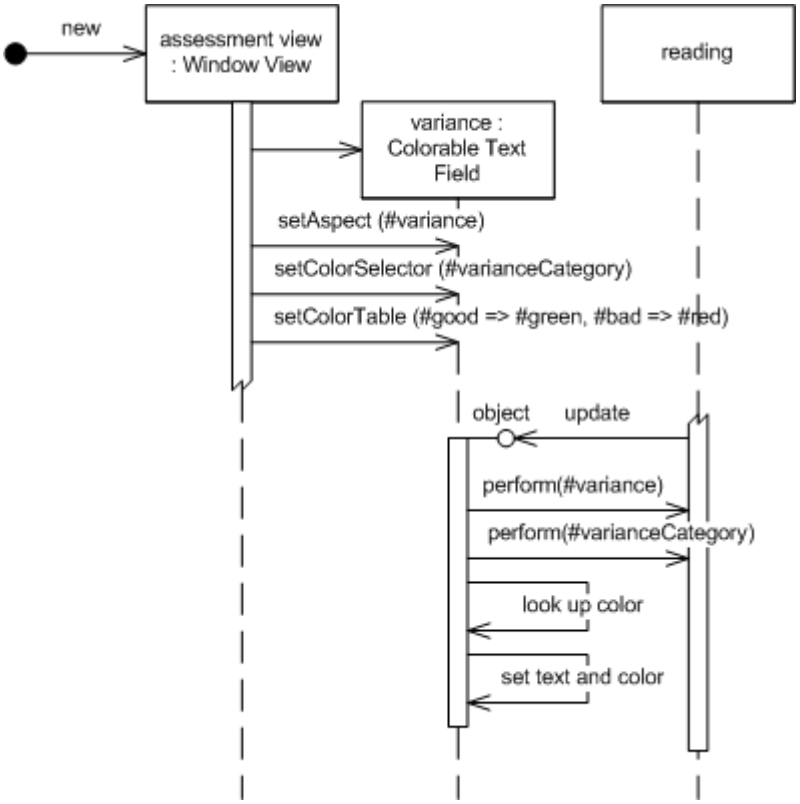


Рисунок 7: Использование наследника класса текстового поля, который поддерживает определение цвета.

Последним штрихом будет создание специального доменного объекта, того, что будет следить за элементами на экране, но при этом от него не зависеть. Другими словами, создадим модель для экрана. Методы, которые раньше вызывались для объекта замера, будут делегироваться моделью в объект замера. Так же в ней будут определены методы «только UI», например, определение цвета.

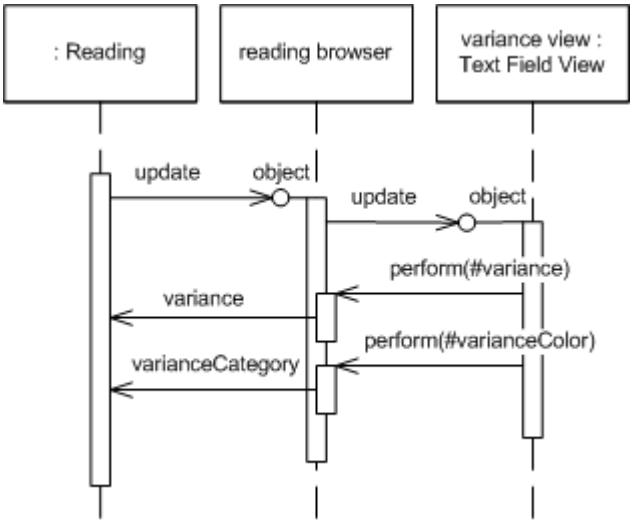


Рисунок 8: Использование промежуточной модели представления (Presentation Model[15]) в обработке логики представления.

Предложенное решение работает во многих случаях, и как мы увидим позже, стало обычной методикой, которой придерживаются разработчики Smalltalk. Я называю его моделью представления (), потому что оно разработано для (и является частью) слоя представления.

Модель представления решает другую проблему логики представления — проблему состояния представления. Основное понятие MVC предполагает, что состояние представления наследуется от состояния модели. Как в этом случае определить, какая станция выбрана в списке? Модель представления (Presentation Model[16]) решает эту проблему тем, что предоставляет место, куда можно разместить такого рода состояние. Похожая проблема возникает, если у нас есть кнопки «сохранить», которые можно нажать только тогда, когда изменились данные — такое состояние определяется взаимодействием с моделью, а не самой моделью.

Сейчас, я думаю, самое время определить характерные черты MVC.

- Сделайте четкое разделение между представлением (представление и контроллер) и доменной моделью — разделенное представление (Separated Presentation[17]).
- Разделите элементы UI на контроллер (реагирование на ввод данных пользователем) и представление (для отображения состояния модели). Контроллер и представление не должны (в большинстве случаев) взаимодействовать непосредственно, только через модель.

- Пусть представления (и контроллеры) обозревают модель. Это позволит нескольким окнам всегда находится в актуальном состоянии без прямого взаимодействия друг с другом — синхронизация через обозреватель (Observer Synchronization[18]).

Продолжение следует[19].

References

1. ↥ тут
   (http://acerv.habrahabr.ru/blog/50830/)
2. ↥ тут
   (http://www.martinfowler.com/eaaDev/uiArchs.html)
3. ↥ Separated Presentation
   (http://www.martinfowler.com/eaaDev/SeparatedPresentation.html)
4. ↥ Domain Model
   (http://martinfowler.com/eaaCatalog/domainModel.html)
5. ↥ Record Set
   (http://martinfowler.com/eaaCatalog/recordSet.html)
6. ↥ Data Binding
   (http://www.martinfowler.com/eaaDev/DataBinding.html)
7. ↥ Flow Synchronization
   (http://www.martinfowler.com/eaaDev/FlowSynchronization.html)
8. ↥ Observer Synchronization
   (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
9. ↥ Observer Synchronization
   (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
10. ↥ Observer Synchronization
    (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
11. ↥ Flow Synchronization
    (http://www.martinfowler.com/eaaDev/FlowSynchronization.html)
12. ↥ Observer Synchronization
    (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
13. ↥ Observer Synchronization
    (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
14. ↥ Separated Presentation
    (http://www.martinfowler.com/eaaDev/SeparatedPresentation.html)
15. ↥ Presentation Model
    (http://www.martinfowler.com/eaaDev/PresentationModel.html)
16. ↥ Presentation Model
    (http://www.martinfowler.com/eaaDev/PresentationModel.html)
17. ↥ Separated Presentation
    (http://www.martinfowler.com/eaaDev/SeparatedPresentation.html)
18. ↥ Observer Synchronization
    (http://www.martinfowler.com/eaaDev/MediatedSynchronization.html)
19. ↥ следует
    (http://acerv.habrahabr.ru/blog/53920/)