Подписаться

7 февраля в 04:48

Охота на мифический MVC. Обзор, возвращение к первоисточникам и про то, как анализировать и выводить шаблоны самому

Проектирование и рефакторинг*, Анализ и проектирование систем*

> *— Не понимаю, почему люди так восхищаются этим Карузо? Косноязычен, гугнив, поёт — ничего не разберешь!*
> *— А вы слышали, как поёт Карузо?*
> *— Да, мне тут кое-что из его репертуара Рабинович напел по телефону.*

Детектив по материалам IT. Часть первая

Я осознаю, что писать очередную статью на тему Модель-Вид-Контроллер это глупо и вредно для «кармы». Однако с этим «паттерном» у меня слишком личные отношения – проваленный проект, полгода жизни и тяжелой работы «в корзину».

Проект мы переписали, уже без MVC, просто руководствуясь принципами – код перестал быть похож на клубок спагетти и сократился наполовину (об этом позже, в обещанной статье про то, как мы применяли «принципы» в своем проекте). Но хотелось понять, что же мы сделали не так, в чем была ошибка? И в течении долгого времени изучалось все, что содержало аббревиатуру MVC. До тех пор пока не встретились исходные работы от создателя – Трюгве Реенскауга…

И тогда все встало на свои места. Оказалось что фактически на основе принципов мы пере-изобретали «original MVC». А то, что зачастую преподносится как MVC, не имеет к нему никакого отношения… впрочем также как и к хорошей архитектуре. И судя по тому сколько людей пишет о несостоятельности «классического MVC», спорит о нем и изобретает его всевозможные модификации, не одни мы столкнулись с этой проблемой.

Более 30 лет собранные в MVC идеи и решения остаются наиболее значимыми для разработки пользовательских интерфейсов. Но как ни странно, несмотря на существующую путаницу и обилие противоречивых трактовок, разработчики продолжают довольствоваться информацией «из вторых рук», черпая знания о MVC из википедии, небольших статей в интернете и фреймворков для разработки веб-приложений. Самые «продвинутые» читают Мартина Фаулера. И почему-то почти никто не обращается к первоисточникам. Вот этот пробел и хотелось бы заполнить. И заодно развеять некоторые мифы.

> **Мифы: MVC создавался для языка SmallTalk**

Концепция MVC была сформулирована Трюгве Реенскаугом (Trygve Reenskaug) в результате его работы в Xerox PARC в 1978/79 годах. Как правило создание MVC связывают с языком SmallTalk, но это не совсем так. На самом деле Реенскауг работал в группе, занимавшейся разработкой портативного компьютера "для детей всех возрастов" Dynabook под руководством Алана Кэя (Alan Kay).

Чтобы оценить масштаб и революционность того проекта, нужно иметь ввиду что это были годы, когда для работы с ЭВМ требовалось штудировать многостраничные мануалы и иметь ученую степень. Задача, которую пытался решить Алан Кэй, состояла в том, чтобы сблизить компьютер и рядового пользователя, «сломать» разделяющую их стену. Он хотел обеспечить пользователя средствами, которые были бы предельно простыми и удобными, но при этом давали бы возможность управлять компьютером и сложными приложениями.

Именно тогда/там закладывались основы графического интерфейса, формировалось понятие "дружелюбного интерфейса". А также разрабатывался язык SmallTalk, вместе с концепциями объектно-ориентированного программирования, чтобы неподготовленный пользователь “мог понимать и писать программы”. Вот как описывает увиденное в Xerox PARC в 1979 году Стив Джобс – How Steve Jobs got the ideas of GUI from XEROX (from 6.30)

Проект велся около 10 лет, группой очень сильных разработчиков. Найденные в результате решения, подходы, принципы и в области пользовательских интерфейсов, и в области объектно ориентированного программирования и вообще в разработке больших и сложных компьютерных систем были в какой-то

степени проссумированы Реенскаугом и составили основу MVC. Так что MVC это действительно прежде всего совокупность направляющих архитектурных идей. В SmallTalk-80 эти идеи всего лишь получили свою первую значимую реализацию. Причем сделано это было уже после ухода Реенскауга из Xerox PARC и без его участия.

К сожалению в течении долго времени о «реальном MVC» не было практически никакой доступной информации. Первая серьезная публикация от создателей появилась лишь 10 лет спустя – "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". Даже Фаулер упоминает, что он изучал MVC по работающей версии SmallTalk – *"у меня был доступ к работающей версии Smalltalk-80, чтобы я мог изучить MVC. Я не могу сказать, что это исследование было тщательным, но оно позволило мне понять некоторые аспекты решения, которые другие описания объяснить не смогли"*.

Так что не удивительно появление «мифов» и разнообразных трактовок. Проблема заключается в том, что многие «вторичные» источники описывают MVC не только в искаженном, но еще и в обманчиво-упрощенном виде, как правило в виде некой формальной схемы.

В результате многие действительно считают MVC схемой или паттерном (из-за чего постоянно возникает вопрос – какая же из множества существующих схем «правильная» и почему их так много?). В более продвинутом варианте MVC называют **составным** паттерном, то есть комбинацией нескольких паттернов, работающих совместно для реализации сложных приложений (тут обычно упоминаются Observer, Strategy и Composite). И лишь немногие понимают, что MVC это прежде всего набор архитектурных идей/принципов/подходов, которые могут быть реализованы различными способами с использованием различных шаблонов...

К последним относится в частности Мартин Фаулер. Вот что он пишет: *"MVC часто называют паттерном, но я не вижу особой пользы воспринимать его как паттерн, поскольку он включает в себя множество различных идей. Разные люди читают про MVC в различных источниках и извлекают от туда разные идеи, но называют их одинаково — «MVC». Это приводит к большой путанице и кроме того служит источником недоразумений и непониманию MVC, как будто бы люди узнавали про него через «испорченный телефон»…. Я уже потерял счет сколько раз я видел что-то, описываемое как MVC, которое им не оказывалось."*[GUI Architectures]

Рискну предположить, что одна из причин «испорченного телефона» заключается в том, что большинство вторичных источников «за кадром» оставляют самое главное – собственно сами архитектурные идеи, заложенные в MVC его создателями, и те задачи, которые они пытались решить. Как раз все то, что позволяет понять суть MVC и избежать огромного количества подводных камней и ошибок. Поэтому в данной статье я хочу рассказать о том, что обычно остается «за кадром» – MVC с точки зрения заложенных в него архитектурных принципов и идей. Хотя схемы тоже будут. Вернее с них и начнем.

Но сначала ссылки. Исходный доклад Реенскауга – "The original MVC reports". Позже Реенскауг все это более четко сформулировал и оформил в своей последующей работе "The Model-View-Controller (MVC). Its Past and Present". Возможно кому-то будет интересна страница, где собраны записи Ренскауга, относящиеся к тому периоду, с его комментариями - MVC XEROX PARC 1978-79.

Уже упоминавшаяся первая публикация о MVC в языке SmallTalk-80 от разработчиков только в улучшенном качестве "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System" (Glenn Krasner и Stephen Pope). Хорошим дополнением служит также статья "Applications Programming in Smalltalk-80.How to use Model-View-Controller" (автор SteveBurbeck учавствовал в разработке компилятора SmallTalk для IBM на основе Smalltalk-80, а также в разработке MacApp). Ну и если кто-то хочет полного погружения – "Smalltalk-80. The Interactive Programming Environment" от знаменитой Адель Голдберг в дискуссиях с которой Реенскаугом и создавались термины Model, View, Controller.

Схемы MVC

Для того, чтобы стало понятно, о чем идет речь и в чем заключается проблема, давайте вначале все же разберем наиболее типичные «схемы» MVC. Это важно, поскольку часто к схемам не дается никаких пояснений и к тому-же бывает, что определения заимствуются из одного места, а схемы из другого. В результате можно встретить одинаковые описания MVC с совершенно разными диаграммами, что очень запутывает.

Итак, несмотря на то, что MVC трактуется и изображается очень по разному, во всем этом многообразии все же можно выделить общее «ядро». Общим является то, что везде говорится о неких трех частях — Модели, Виде и Контроллере, которые связаны между собой определенным образом, а именно:

1. Модель ничего не знает ни о Виде, ни о Контроллере, что делает возможным ее разработку и тестирование как независимого компонента. И это является главным моментом MVC.

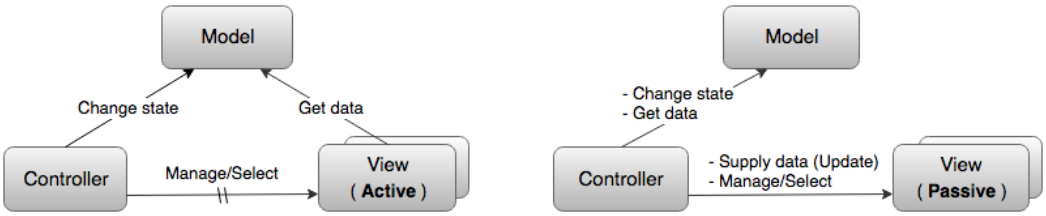
2. Вид отображает Модель. И значит, он каким-то образом должен получать из нее нужные для отображения данные. Наиболее распространены следующие два варианта: 1) **Активный Вид**, который знает о Модели и сам берет из нее нужные данные. 2) Пассивный Вид, которому данные поставляет Контроллер. В этом случае Вид с Моделью никак не связан.

   Видов может быть несколько — они могут по разному отображать одни и те же данные, например в виде таблицы или графика, или же отвечать за отображение разных частей данных из Модели.

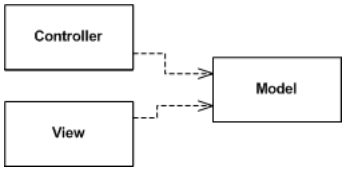
3. Контроллер является пожалуй самым неоднозначным компонентом. Тем не менее общим является то, что Контроллер всегда знает о Модели и может ее изменять (как правило в результате действий пользователя).

А также он может осуществлять управление Видом/Видами (особенно если их несколько) и соответственно знать о Видах, но это не обязательно.

Отсюда мы получаем базовые (максимально упрощенные) схемы двух наиболее часто встречающихся разновидностей MVC. Перечеркнутой линией обозначена необязательная связь Контроллера с Видом.



Вот так базовая схема выглядит у Фаулера: "Основные связи между Моделью, Видом и Контроллером. (Я называю их основными, потому что на самом деле Вид и Контроллер могут быть связанными друг с другом непосредственно. Однако, разработчики в основном не используют эту связь.)":



Далее. Модель, как и Вид, тоже может быть Пассивной либо Активной. **Пассивная Модель** никак не воздействует ни на Вид ни на Контроллер. В этом случае все изменения Модели отслеживаются Контроллером и он же отвечает за перерисовку Вида, когда это необходимо.

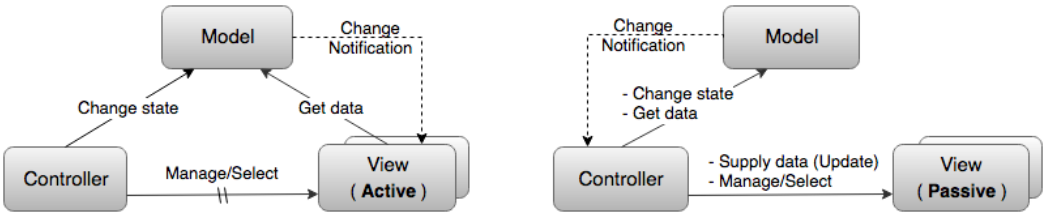
Но обычно, под MVC все таки подразумевают вариант с **Активной Моделью**.

«Активная Модель» **оповещает** о том, что в ней произошли изменения. И делает она это посредством шаблона *Наблюдатель*, рассылая уведомления о изменениях всем своим «подписчикам». «Активный Вид» подписывается на эти сообщения сам и таким образом знает когда нужно заново считать из модели нужные ему данные и обновиться. В случае «Пассивного Вида», подписчиком является Контроллер, который затем уже обновляет Вид.

Шаблон *Наблюдатель* позволяет Модели с одной стороны информировать Вид или Контроллер о том что в ней произошли изменения, а с другой фактически ничего о них «не знать» (кроме того что они реализуют некий заданный интерфейс «подписчика») и тем самым оставаться независимой. Это называется **слабым связыванием** и считается вторым ключевым моментом MVC.

Именно поэтому, когда говорится, что MVC это составной шаблон, то в первую очередь в качестве одного из его компонентов упоминается паттерн *Наблюдатель*. На диаграммах слабое связывание принято рисовать пунктирной стрелкой, но многие это правило игнорируют.

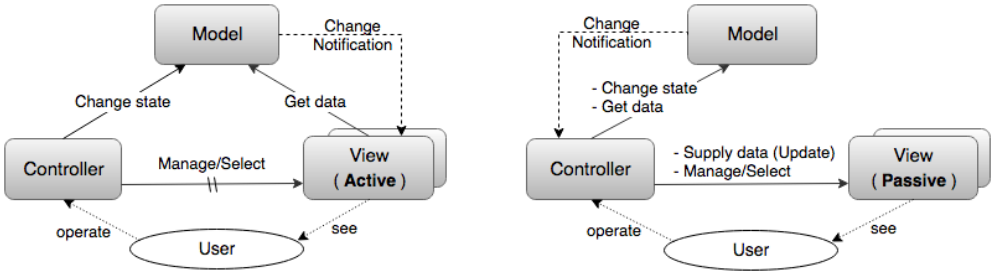
Таким образом, более продвинутые «схемы MVC» будут выглядеть так:



Замечание: *встречаются авторы, которые в термины Пассивная и Активная модель вкладывают совсем иной смысл. А именно то, что обычно принято называть Тонкой моделью (модель содержащая исключительно данные) и Толстой моделью (полноценная модель содержащая всю бизнес логику приложения).*

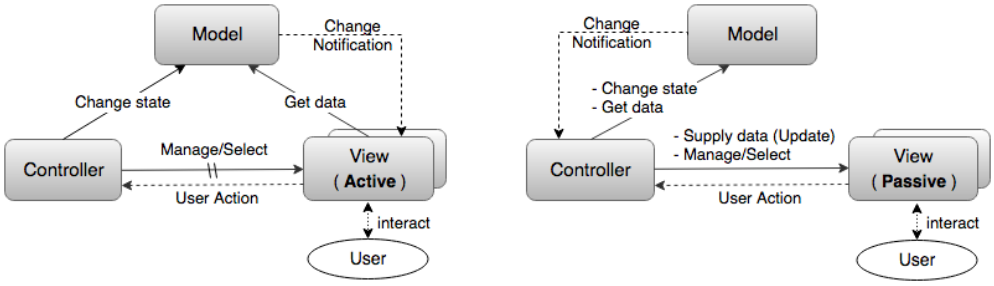
Ну и последнее. Вообще говоря MVC, в любой своей разновидности, это прежде всего шаблон для разработки приложений с пользовательским интерфейсом и его главное назначение – обеспечить взаимодействие приложения с пользователем. Поэтому в полноценной MVC схеме (явно или неявно) должен присутствовать **пользователь**. И тут в основном встречаются две трактовки:

1. Пользователь управляет приложением через Контроллер, а Вид служит исключительно для отображения информации о Модели, и пользователь его лишь видит



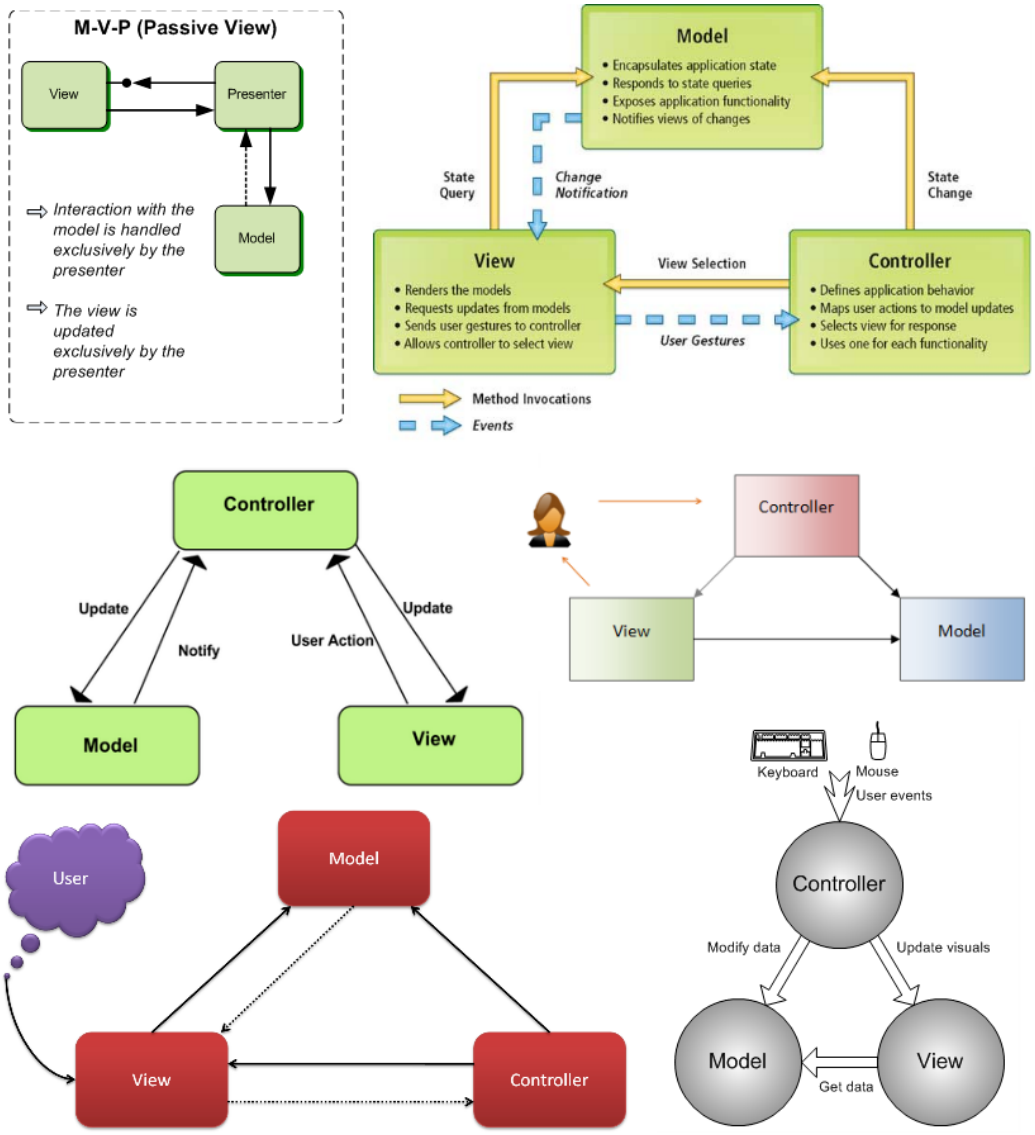
Часто указывают/рисуют лишь то, что пользователь действует на Контроллер, а то что он видит Вид опускается.

2. Пользователь взаимодействует только с Видом. То есть Вид не только отражает Модель, но также принимает команды пользователя и передает их Контроллеру. В этом случае между Видом и Контроллером образуется еще одна связь: прямая (Вид знает о Контроллере и напрямую передает информацию) или, чаще всего, ослабленная (Вид просто рассылает информацию о действиях пользователя всем заинтересованным подписчикам а Контроллер на эту рассылку подписывается)



Замечание: *нужно иметь ввиду, что вариант с* ***Пассивным Видом****, когда Вид никак не связан с Моделью и данные для отображения ему поставляет Контроллер, иногда называют MVC, а иногда выделяют в отдельную разновидность — MVP и тогда Контроллер переименовывают в* ***Презентер****.*

Для иллюстрации всего вышесказанного несколько диаграмм «из интернета» (надеюсь стало понятнее почему они такие разные):



А теперь самое главное — как применяются, что обозначают и чему соответствует Модель Вид и Контроллер при написании приложений?

Тут можно выделить два кардинально отличающихся подхода, в каждом из которых Модель, Вид и Контроллер трактуются весьма различным образом.

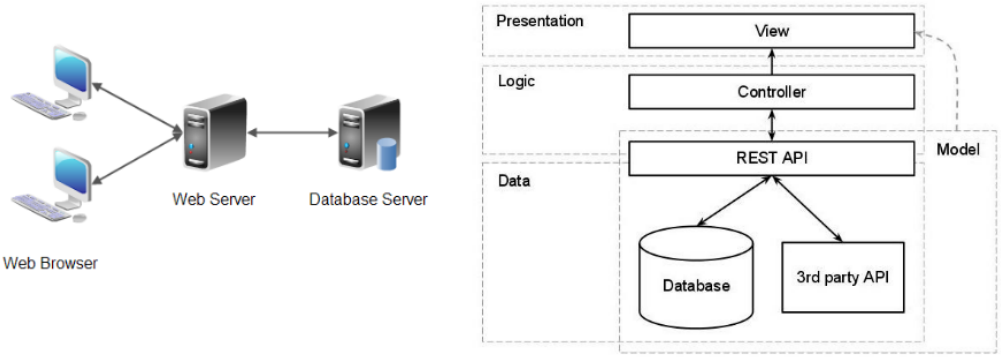
«Трехуровневый MVC» от веб

Первый подход идет из веб-программирования, где MVC получил самое широкое распространение, и поэтому в нем максимально отразились свойственные веб-программированию черты. А именно, привязка к трехуровневой архитектуре «клиент–сервер–база данных» и преобладание скриптовых языков. В результате компоненты MVC формально привязываются к трем слоям архитектуры и получается что:

1. **Модель = База Данных**
   Модель — это просто данные, которыми оперирует приложение

2. **Контроллер = Сервер**
   Контроллер — это бизнес-логика приложения. Иногда еще говорят что контроллер это центр обработки всех запросов и принятия решений, а также промежуточный слой обеспечивающий связь модели и представления.

3. **Вид = Клиент** (как правило тонкий)
   Вид — это пользовательский интерфейс. Причем интерфейс в этом случае, как правило, понимается в основном исключительно как «дизайн», просто набор графических элементов. Логика же работы этого интерфейса, как и логика работы с данными, выносится в Контроллер



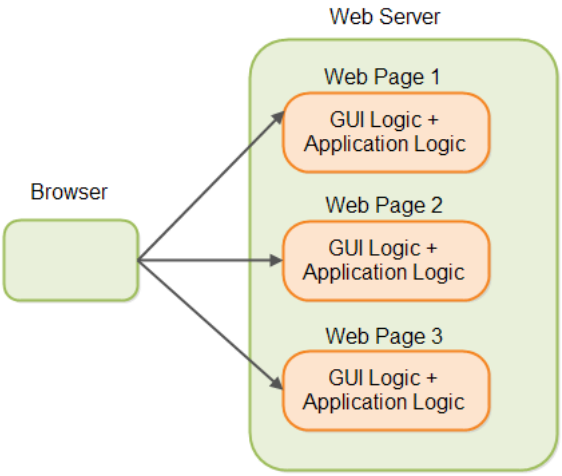
Про неадекватность этого подхода написано уже так много, что это вошло даже в википедию (MVC. Наиболее частые ошибки). Хорошо и подробно возникающие при этом проблемы рассматриваются в статье, ставшей своего рода классикой "M в MVC: почему модели непоняты и недооценены". Поэтому постараюсь просто кратко просуммировать:

1. Независимость Модели является главным в MVC. Если Модель тонкая, то есть содержит лишь данные, то возможность ее независимой разработки имеет мало смысла. Соответственно при таком подходе теряет смысл и сам MVC

2. Вся бизнес логика приложения, то есть большая часть кода, сосредотачивается в Контроллере и это при том что как раз Контроллер является самой зависимой частью в MVC – в общем случае он зависит и от Модели и от Вида. Вообще говоря в хорошо спроектированных приложениях стараются делать с точностью до наоборот – наиболее зависимые части должны быть минимальными, а не максимальными

3. На практике Контроллеру в веб-приложении обычно соответствует один скрипт и вынесение всей бизнес-логики в Контроллер фактически означает еще и то, что большая часть приложения оказывается в одном скрипте. Отсюда и появился термин ТТУК — толстый тупой уродливый контроллер

4. Поскольку, как правило, тонкой является не только Модель но также и Вид (тупой Вид или тупой интерфейс — Dumb GUI, Dumb View), то, как следствие, в Контроллер помимо всей бизнес-логики приложения помещается также еще и логика управления пользовательским интерфейсом. То есть, вместо разделения бизнес логики и логики представления при таком подходе получается их смешение.



Программа, конечно, разбивается на множество MVC, соответствующих страницам веб-приложения, и это спасает ситуацию но, увы, не меняет сути. Проблема эта известна, вот неплохая статья — "RIA Architecture".

> **Типичные ошибки: смешение в Контроллере бизнесс-логики и GUI-логики**

Хорошая новость заключается в том, что «веб-вариант MVC», всего несколько лет назад бывший самым распространенным, сейчас активно сдает позиции. Плохо то, что он по прежнему распространен, только теперь не в явном, а в замаскированном виде. Поскольку за фразы (цитирую): "*Модель это обмен данными с БД* и т.п. *Контроллер логика обработки этих данных и подготовка к View*" сейчас активно «минусуют», то стали писать:

- Модель — это данные и методы работы с ними
- Контроллер — обработка действий пользователя и вводимой им информации

Дело в том, что в объектно-ориентированном приложении нет данных, а есть множество объектов и каждый из них содержит какие-то данные и методы работы с ними. В том числе и объекы доступа к базе данных (если они имеются). Поэтому когда определение Модели начинается со слова «данные», то оно в сущности имеет мало смысла и нередко в завуалированной форме подразумевает все тот же самый доступ к базе данных. В **обработку** же **действий пользователя** нередко помещается львиная доля бизнес логики и в результате по прежнему вся, или почти вся, логика приложения часто оказывается в Контроллере.

«Архитектурный MVC»

Второй подход гораздо ближе к первоисточникам. Поэтому разберем его подробнее.

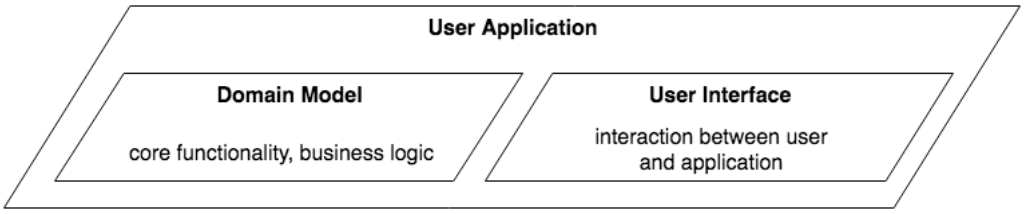
Мартин Фаулер абсолютно прав, когда говорит что MVC это не паттерн, а набор архитектурных принципов и идей, используемых при построении пользовательских информационных систем (как правило сложных).

Архитектурные принципы мы постарались собрать и описать в статье "Создание архитектуры программы или как проектировать табуретку". Если же говорить предельно кратко, то суть состоит в следующем: сложную систему нужно разбивать на модули. Причем декомпозицию желательно делать **иерархически**, а модули, на которые разбивается система, должны быть, по возможности, независимы или **слабо связаны** (**Low coupling**). Чем слабее связанность, тем легче писать/понимать/расширять/чинить программу. Поэтому одной из основных задач при декомпозиции является минимизация и ослабление связей между компонентами.

Давайте посмотрим, как эти принципы применяются в MVC для создания первичной архитектуры (декомпозиции) пользовательских приложений. По сути в основе MVC лежат три довольно простые идеи:

«1» Отделение модели предметной области (бизнес логики) приложения от пользовательского интерфейса

Первая и основная идея MVC заключается в том, что любое пользовательское приложение в первом приближении можно разделить на два модуля — один из которых обеспечивает основной функционал приложения, его бизнес логику, а второй отвечает за взаимодействие с пользователем:



Тем самым мы получаем возможность разрабатывать **модель предметной области**, содержащую бизнес-логику системы и составляющую функциональное ядро приложения, не думая о том как именно она будет взаимодействовать с пользователем.

Задача же взаимодействия с пользователем выносится в отдельный модуль – **пользовательский интерфейс** и тоже может решаться относительно независимо.

Именно модель предметной области (Доменная Модель от английского **domain model**) считается **Моделью** в «архитектурном MVC» (отсюда и термин). Поэтому так важно чтобы она была независимой и могла независимо разрабатываться и тестироваться.

"Сердцевиной идеей MVC, как и основной идеей для всех последующих каркасов, является то, что я называю «отделенное представление» (Separated Presentation). 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)." — Фаулер

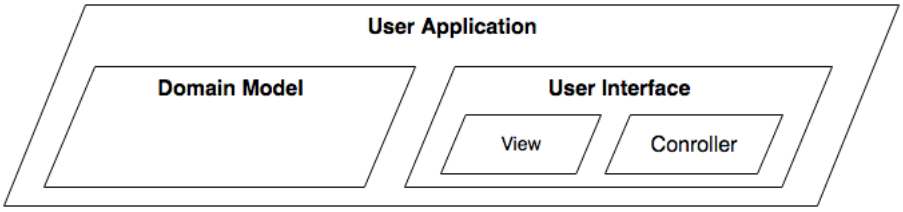
«2» Независимость Модели и синхронизация пользовательских интерфейсов за счет шаблона *Наблюдатель*

Вторая ключевая идея заключается в том, что для того, чтобы иметь возможность разрабатывать Модель **независимо**, необходимо ослабить ее зависимость от пользовательского интерфейса. И делается это, как уже упоминалось выше, за счет шаблона ***Наблюдатель***.

Модель **рассылает извещения об изменениях**. Интерфейс подписывается на эти оповещения и таким образом знает, когда нужно заново считать данные из модели и обновиться. Благодаря этому мы получаем практически **независимую Модель**, которая ничего не знает о связанных с ней пользовательских интерфейсах, кроме того что они реализуют интерфейс «наблюдателя».

«3» Разделение Пользовательского Интерфейса на Вид и Контроллер.

Третья идея это просто второй шаг иерархической декомпозиции. После первичного разделения приложения на бизнес модель и интерфейс, декомпозиция продолжается на следующем иерархическом уровне и уже пользовательский интерфейс, в свою очередь, делится на Вид и Контроллер.



У меня сложилось впечатление, что суть этого деления мало кто понимает и соответственно может объяснить. Обычно приводят лишь стандартную обтекаемую формулировку, что Контроллер как-то реагирует на действия пользователя, а Вид отображает Модель (поэтому в большинстве реализаций именно Вид подписывается на извещения об изменениях Модели. Хотя, как уже говорилось, подписчиком может быть и Контроллер, либо Вид и Контроллер вместе).

Поскольку деление пользовательского интерфейса на Вид и Контроллер относится ко **второму уровню иерархии**, оно гораздо менее значимо чем первичное разделения приложения на доменную модель и интерфейс. Очень часто (особенно когда дело касается простых виджетов) оно вообще не делается и используется «**упрощенный MVC**», в котором имеется только Модель и единый UI-компонент, представляющий собой объединенный ВидКонтроллер. Более подробно об этом речь пойдет чуть позже.

«Архитектурный MVC» на первый взгляд выглядит вполне разумно. Но как только мы попытаемся применить его не к учебному примеру из трех классов а к реальной программе, то столкнемся с целым рядом проблем и вопросов, о которых редко пишут, но которые чрезвычайно важны. И касаются они не только пользовательского интерфейса, но и самой Модели. Так что предлагаю таки попробовать с ними разобраться и, наконец-то, "послушать Карузо", то есть обратиться к первоисточникам.

«Original MVC»: Реенскауг и SmallTalk-80

Мы привыкли к тому, что MVC почти всегда рассматривается на примере создания какого нибудь простейшего графического компонента, вся «бизнес логика» которого помещается в один класс с данными и парой методов для их изменения. Но что делать, когда речь идет о **реальных** приложениях, ядро которых состоит из многих взаимосвязанных объектов работающих совместно?

В общем случае *Модель* это один объект или множество объектов? И на самом ли деле *Модель* в «MVC-схеме» тождественна доменной модели, описывающей предметную область и бизнес-логику приложения?

То, что Модель реализует шаблон *Наблюдатель* явно указывает на то, что Модель это именно один объект. На это же указывает и то, что Вид и Контроллер должны знать о Модели (для того чтобы брать из нее данные и вносить изменения) и следовательно они должны содержать на нее ссылку. Но тогда, если считать, что под Моделью подразумевается доменная модель, мы вновь приходим к тому что все ядро приложения оказывается в одном объекте. Только теперь вместо толстого уродливого Контроллера, у нас

появляется толстая Модель. Толстая Модель конечно лучше, поскольку она независима и в ней, по крайней мере, не смешивается бизнес логика с логикой GUI, но все равно такое решение сложно отнести к хорошей архитектуре.

Остается второй вариант — Модель это множество доменных объектов, совместно реализующих бизнес логику. Это предположение подтверждает и сам Реенскауг: "*A model could be a single object (rather uninteresting), or it could be some structure of objects.*" Но тогда остается открытым вопрос – кто реализует шаблон *Наблюдатель*, откуда берет данные Вид, куда передает команды пользователя Контроллер?

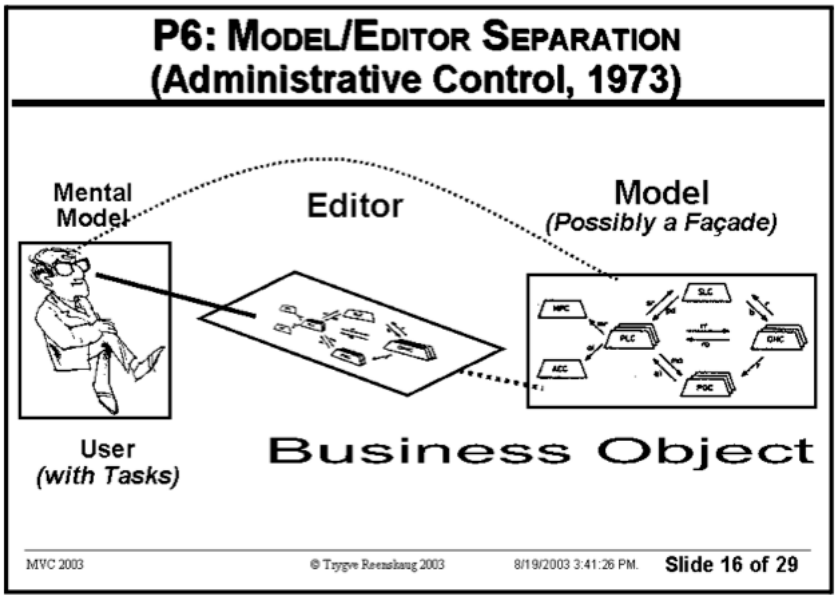
И вот здесь нередко встречается попытка обмануть самих себя путем примерно следующего рассуждения: "пусть Модель это множество доменных объектов, но… среди этого множества есть в том числе и «объект с данными», вот он-то и будет реализовывать шаблон Наблюдатель, а также служить источником данных для Вида." Эту уловку можно назвать «Модель в Модели». И по сути это еще один «завуалированный» вариант того, что «Модель это данные».

Тут можно сказать лишь одно: архитектура, в которой один модуль (Вид или Контроллер), должен «лезть» **внутрь** другого модуля (доменной модели) и искать там для себя данные или объекты для изменения очень нехорошо «пахнет». Получается что Вид и Контроллер зависят от деталей реализации доменной модели, и если структура этой самой модели изменится, то придется переделывать весь пользовательский интерфейс.

Для того же, чтобы понять «а как должно быть» предлагаю вновь обратится к «принципам». Когда говорилось о том, что систему надо разбивать на модули, **слабо связанные** друг с другом, мы не упомянули главное правило, позволяющее добится этой самой слабой связанности. А именно – модули друг для друга должны быть «черными ящиками». Ни при каких условиях один модуль не должен обращаться к объектам другого модуля напрямую и что либо знать о его внутренней структуре. Модули должны взаимодействовать друг с другом лишь на уровне абстрактных интерфейсов (Dependency Inversion Principle). А реализует интерфейс модуля как правило специальный объект — **Фасад**.

И если поискать какие же паттерны позволяют добится слабой связанности, то на первом месте будет находится именно паттерн *Фасад*, и только затем *Наблюдатель* и тд.

Ну а теперь схема из доклада Трюгве Реенскауга:



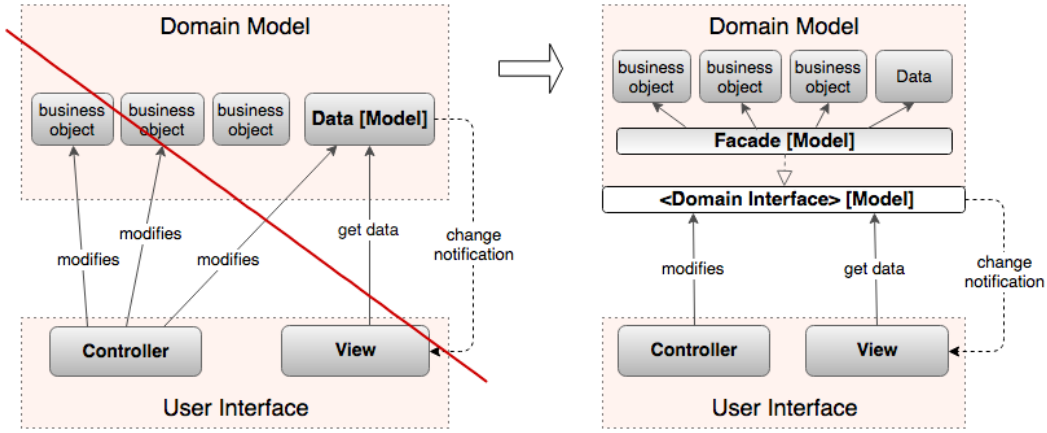
Пояснение: *Поскольку во времена создания MVC интерфейсы компьютерных программ были в основном текстовыми, то есть, по сути представляли собой простейший вид редактора, то вместо термина «Пользовательский Интерфейс», который появился позже, Трюгве Реенскауг использует термин «Editor» (редактор).*

Таким образом, ключевая идея MVC действительно состоит в том, что пользовательское приложение делится на два модуля – один из которых моделирует предметную область и реализует бизнес логику (*доменная модель*), а второй отвечает за взаимодействие с пользователем (*пользовательский интерфейс*). Но при этом **Модель** в «MVC схеме» вовсе **не тождественна доменной модели** (которая может быть

сколь угодно сложной и состоять из множества объектов), а является всего лишь ее **интерфейсом** и **фасадом**.

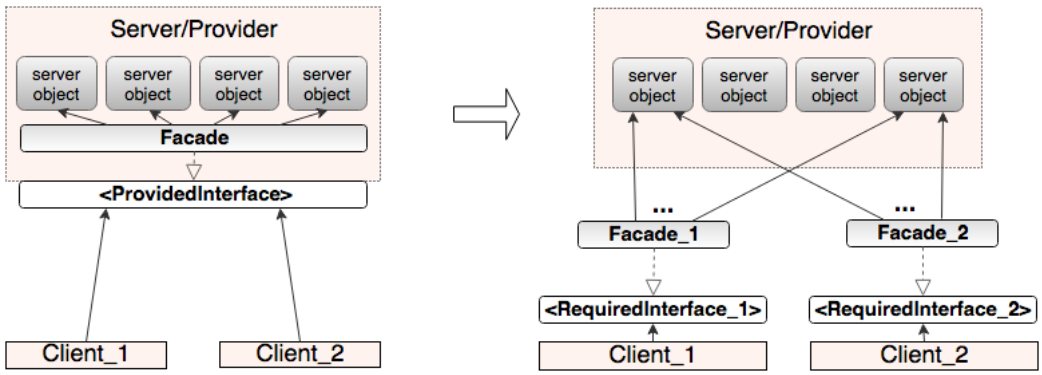
Так что ни Вид ни Контроллер разумеется не должны знать о том, как устроен модуль предметной области (доменная модель), где и в каком формате там храняться данные, и как именно осуществляется управление. Они взаимодействуют лишь с **интерфейсом** и реализующим его **объектом-фасадом**, который предоставляет все нужные данные в нужном формате и удобный набор высокоуровневых команд для управления подсистемой, а также реализует шаблон Наблюдатель, для извещения о **значимых** изменениях в подсистеме. И если мы захотим поменять базу данных, использовать облако, или вообще собирать нужные нам данные из различных источников в сети… если внесем какие угодно изменения в бизнес логику приложения, но при этом оставим неизменным интерфейс-фасад, то ни Вид, ни Контроллер это никак не затронет. Мы имеем архитектуру **устойчивую** к изменениям.

И если уж рисовать схему MVC, то выглядеть она должна следующим образом:

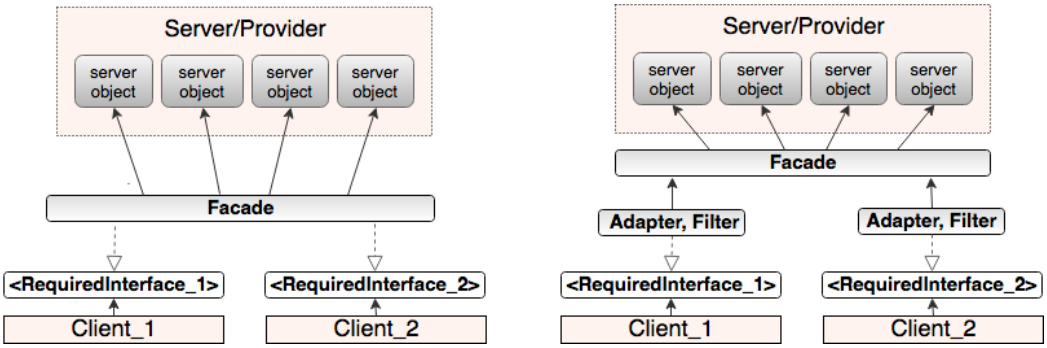


Давайте рассмотрим данную схему подробнее. Традиционно в клиент серверных приложениях главным считается сервер. Он предоставляет услуги/сервисы и решает в каком виде это должно быть реализовано. Соответственно интерфейс и фасад, как правило, определяются с точки зрения сервера. А клиенты под этот заданный формат подстраиваются.

На практике же более адекватной оказывается не сервер-ориентированная архитектура, а клиент-ориентированная. В ней фокус с сервера смещается в сторону клиента и интерфейс, вернее интерфейсы (и фасад или фасады), определяются исходя из потребностей клиентов. Вместо **Предоставляемого Интерфейса** (*Provided Interface*) используются **Требуемые Интерфейсы** (RequiredInterface).



Конкретные реализации могут варьироваться, но это не суть важно

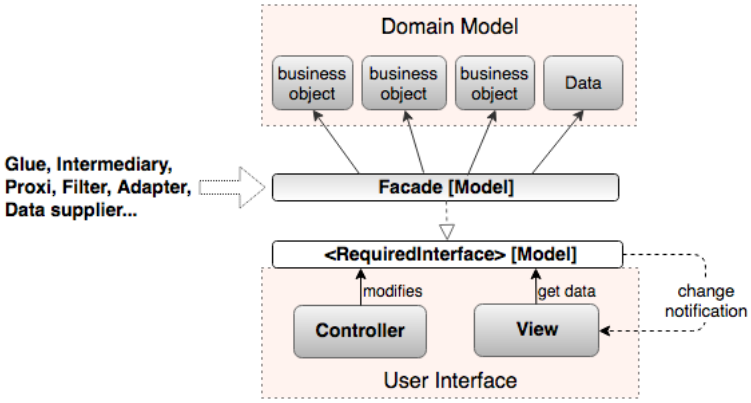


Клиент ориентированный подход гораздо лучше соответствует *Принципу разделения интерфейсов* (*Interface Segregation Principle*) поскольку в нем вместо единого для всех толстого **ProvidedInterface** используется множество тонких **RequiredInterface**.

Если я не ошибаюсь, именно такой подход используется в архитектуре микросервисов. Там для взаимодействия с множеством сервисов введено понятие шлюза, который является ни чем иным как фасадом — *"An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object-oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. "* Building Microservices: Using an API Gateway.

Причем шлюз этот "*вместо того чтобы обеспечивать общий единый для всех API, предоставляет различные API для каждого клиента (Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to it's requirements)*" API Gateway.

Как мы увидим дальше клиент-ориентированный подход применялся также и в SmallTalk-80. Но вначале давайте просто пере-рисуем схему MVC с учетом вышесказанного:



Смотрим на фасад… Вот он тот самый клей (glue), объект посредник, прокси, фильтр, адаптер… связывающий между собой доменную модель и пользовательский интерфейс и поставляющий нужные данные в нужном/удобном формате.

Удивительно то, что кроме Реенскауга об этом почти никто не пишет. Хотя некоторые пере-открывают эту идею самостоятельно (пример можно посмотреть тут или тут раздел "Interface-Based Programming Techniques").

Особенно хорошо тема Моделей-интерфейсов раскрыта в статье одного из JavaGuru — Advanced MVC Patterns. Автор подчеркивает, что Модели это не данные, а исключительно интерфейсы/объекты-посредники/фильтры (Models as Proxies, Models as Filters), обеспечивающие удобный доступ к данным, которые могут находится где угодно – на разных машинах, в разных форматах: *"О чем большинство программистов не думает, так это о том, что модели являются всего лишь интерфейсами. Они не должны содержать никаких данных!.. Модели-посредники расширяют охват и позволяют использовать уже* ***существующие данные*** *где бы они не находились"*.

Из-за того что фасад, присутствующий в original MVC, был «утерян», то его роль зачастую берет на себя Контроллер. Отсюда и проистекают представления что Контроллер находится «между Моделью и Видом», служит клеем между ними и обеспечивает нужные Виду данные.

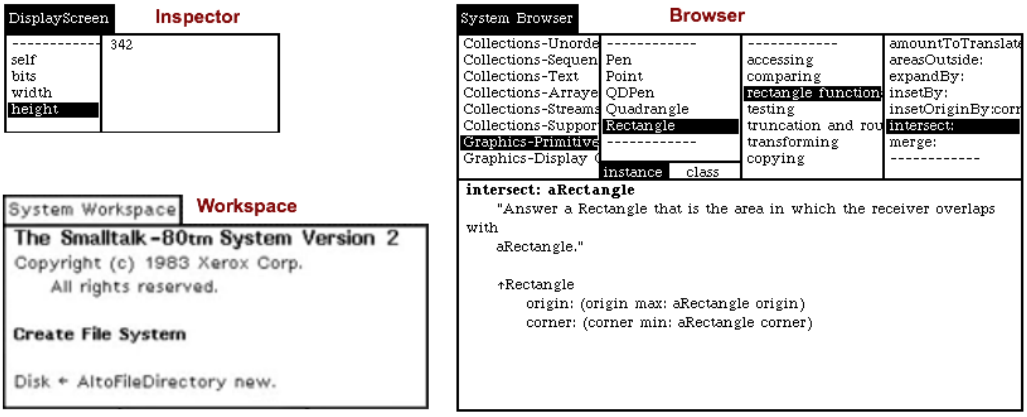
На форумах нередко встречается вопрос — "Чем контроллер отличается от фасада?". Не смотря на наивность этот вопрос вполне закономерен и на него сложно дать разумный ответ поскольку во многих MVC фреймворках Контроллер на самом деле фактически является фасадом — **«Фронт-Контроллер»**.

Чем плохо такое решение? Если оно граммотно реализовано, то ничем. Но это в теории. А на практике нередко происходит путаница концепций и понятий и в результате Фронт-Контроллер с одной стороны злоупотребляет своими полномочиями и вместо **делегирования команд** начинает включать в себя **реализацию** бизнес логики. А с другой – продолжает одновременно выполнять функции пользовательского интерфейса и в результате в нем происходит уже упоминавшееся смешение «бизнес логики» и «GUI логики» (что собственно и делает его код похожим на огромную свалку).

Думаю, что пришло время перейти к Smalltalk. Smalltalk-80 создавался очень талантливыми людьми. С документацией в нем действительно имелись проблемы (тем более что «шаблонов проектирования» тогда еще не существовало) но вот с реализацией в основном все было хорошо и пользовательские интерфейсы, конечно же, не взаимодействовали с доменной моделью напрямую.

Между интерфейсом и доменной моделью (объектами языка SmallTalk) всегда располагался некий промежуточный класс/объект, который обеспечивал удобный интегральный доступ к доменным объектам их данным и методам. Вот эти-то промежуточные объекты (по сути выполняющие роль фасадов) и были в действительности Моделями в SmallTalk-80.

Например, для работы с кодом в Smalltalk использовались следующие GUI интерфейсы: Inspector, Browser, Workspace,…



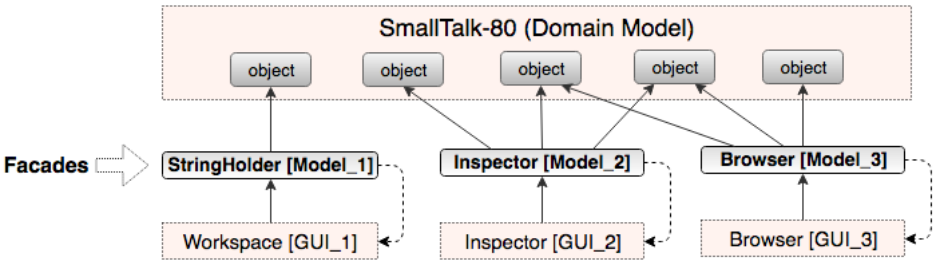
Вот что пишет об их устройстве Glenn Krasner:

*"Inspector в системе состоит из двух видов. ListView отображает список переменных (слева), а TextView показывает значение выбранной переменной (справа)… Моделью для этих видов служит экземпляр класса «**Inspector**»… Отдельный класс «**Inspector**» является **посредником** или **фильтром** для того чтобы обеспечивать доступ к любому свойству любого объекта. Использование промежуточных объектов между View и "**actual**" models является типичным способом изолировать поведение отображения от модели приложения...*

*Как и в случае Inspector, промежуточные объекты использовались также в качестве моделей для системных браузеров. Экземпляр класса «**Browser**» является моделью-посредником для каждого системного браузера..."*

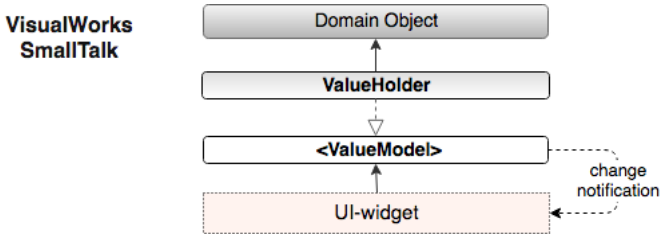
Замечание: название класса-посредника, описывающего промежуточный объект-фасад, обычно совпадало с названием отображающего его виджета. У Inspector промежуточная модель так и называлась «*Inspector*», а у Browser соответственно – «*Browser*».

В случае Workspace, который был одним из простейших интерфейсов "*моделью служил экземпляр StringHolder, который просто предоставлял текст, то есть строку с информацией о форматировании".*



В конце своей статьи Krasner приводит список использовавшихся в SmallTalk Моделей (наследников базового класса Model): **StringHolder**, **Browser**, **Inspector**, **FileModel**, **Icon**… А также отмечает что "*the models were almost always some sort of filter class*".

Позже в VisualWorks Smalltalk идея промежуточных Holder-ов была развита и реализована в полной мере. Там для доступа к каждой переменной, принадлежащей доменным объектам, используется свой интерфейс и фасад – ValueModel и ValueHolder. И, как не трудно догадаться, именно ValueModel реализует шаблон *Наблюдатель*, извещая GUI о происходящих «в домене» изменениях.



> **Типичные ошибки: обращение к доменным объектам напрямую**

Поскольку на практике в любом сколько нибудь серьезном приложении сложно обойтись без фасадов, то не удивительно что во многих фреймворках и "модификациях MVC" аналоги фасада или объекта-посредника между GUI и доменной моделью пере-изобретаются под самыми разными именами. Помимо Front-Controller здесь можно упомянуть также ApplicationModel, ViewModel (подробнее см. дискуссию Model-ModelView-Controller) и Proxy-model.

Из-за того, что разработчики не всегда хорошо понимают что стоит за всеми этими «моделями», а сами модели привыкли воспринимать как данные а не интерфейс, то это становится источником еще одной весьма распространенной и ресурсоемкой ошибки. Вместо того чтобы нужным образом всего лишь **интерпретировать** и адаптировать имеющиеся доменные данные с помощью моделей-посредников их начинают **копировать** в эти модели-посредники.

Например, ValueHolder это, как правило, всего лишь обертка вокруг уже существующей доменной переменной, он не должен содержать данные, он содержит ссылку на данные. Вот что пишут: "*ValueModel does not need to actually store the value because it is already being stored by another model*" (Understanding and Using ValueModels).

А вот цитата из статьи Advanced MVC Patterns: "*Одна из самых распространенных ошибок, которую совершают люди когда используют Swing компоненты, заключается в копировании данных в модели этих Swing компонент. Правильный же способ состоит в том чтобы использовать уже существующие данные, адаптируя их при помощи фильтра… Запомните: никогда не копируйте данные которые можно просто интерпретировать!*".

Рассмотрим следующий простой пример. Если последовать интернет советам и для "добавления элементов в список" использовать код подобный этому (взято со StackOverflow и Adding and Removing an Item in a JList), то будет происходить как раз то самое **копирование** данных в модель списка:

```
Object[] items; // Доменный объект

DefaultListModel model = new DefaultListModel();
JList list = new JList(model);
for (int i = 0; i < items.length; i++){
     // КОПИРОВАНИЕ доменных данных в модель списка!
     model.addElement(items[i]);
}
```

Правильнее, конечно же, использовать данные массива просто обернув их в интерфейс ListModel (тем более что для этих целей создана уже почти готовая AbstractListModel):

```
// создаем фасад-адаптер к доменным данным,
// который просто интерпретирует их нужным образом
ListModel model = new AbstractListModel() {
    public int getSize() {return items.length;}
    public Object getElementAt(int index) {return  items[index];}
};
// передаем созданный фасад списку в качестве модели
JList list = new JList(model);
```

И если надо объединить данные, отфильтровать или преобразовать каким-нибудь образом, то совершенно не нужны промежуточные массивы. Все делается непосредственно в модели-фасаде

```
Object[] items1;  Object[] items2; // Доменные объекты

// модель-фасад которая объединяет массивы
ListModel model = new AbstractListModel() {
    public int getSize() { return items1.length + items2.length;}
    public Object getElementAt(int index) {
        return index<items1.length ? items1[index] : items2[index-items1.length];
    }
};
JList list = new JList(model);
```

В случае небольших статических массивов преимущества не очевидны. Но в общем случае такой подход позволяет не только избегать копирования, но прежде всего защищает от проблем связанных с рассинхронизацией данных (если не полениться прописать методы извещающие слушателей об изменениях).

Ну а если хочется краткости, то тогда уж лучше так:

```
JList list = new JList(items);
```

В этом случае Джава сама сделает обертку-адаптер вместо копирования.

> **Типичные ошибки: копирование доменных данных в модели GUI-компонент**

Ну и наконец мы может развеять главный миф, являющийся источником наибольшего количества проблем и ошибок.

> **Мифы: Модель в «MVC схеме» тождественна доменной модели и данным**

Путаница возникает из-за того что одно и то же слово «Модель» используется в разных контекстах. Когда речь идет о декомпозиции и отделении *бизнес-логики* от *пользовательского интерфейса*, то под Моделью действительно понимается именно **доменная модель**, содержащая данные и логику работы с ними и обеспечивающая основной функционал приложения.

Но в контексте шаблонов и схем Модель это прежде всего **интерфейс** и реализующий его **объект-посредник** (фасад, адаптер, прокси) обеспечивающие удобный и безопасный доступ к доменным данным, которые могут находится где угодно. Реенскауг так и писал: "*model object with a façade that reflects the user's mental model*".

Когда MVC преподносится исключительно как «схема», то наличие «промежуточных моделей» кажется сложным и запутанным. Появляются вопросы ("Чем эти модели отличаются друг от друга?", "Как их правильно использовать?"), неоднозначные трактовки и множество возможностей сделать ошибку.

Но если понимать заложенные в MVC архитектурные идеи, то все становится предельно ясным: пользовательский интерфейс не имеет права обращаться к объектам доменной модели напрямую. А

значит между доменной моделью и пользовательским интерфейсом должнен находиться фасад/посредник/адаптер..., и взаимодействовать пользовательский интерфейс (Вид и Контроллер) может только с ним. Возможностей сделать ошибку – ноль.

И по большому счету становится все равно каким термином этот объект-посредник называется и какая именно Model-View-Whatever разновидность MVC используется… Начинаешь видеть: какая задача решается, с помощью каких шаблонов и то, насколько хорошо или плохо это делается

В принципе на этом статью можно и закончить. Как уже упоминалось деление пользовательского интерфейса на Вид и Контроллер является наименее значимым, даже вспомогательным моментом. Но с другой стороны пользовательский интерфейс присутствует в каждом пользовательском приложении и иметь представление о подходах и идеях наработанных в этой области часто бывает полезно. К тому же именно вокруг Контроллера ведутся основные споры. Поэтому – вторая часть, полностью посвященная именно этой теме.

MVC, архитектура приложений