

# Martin Fowler — GUI Architectures. Часть 1. :: РУССКИЕ ДОКУМЕНТЫ

---

Раздел: [Programming<sup>\[1\]</sup>](#) / [Теория разработки<sup>\[2\]</sup>](#) @ 04.03.2009 | Ключевые слова: [gui<sup>\[3\]</sup>](#) [юзабилити<sup>\[4\]</sup>](#)



[mvc<sup>\[5\]</sup>](#) [mvp<sup>\[6\]</sup>](#)

Автор: aserv

Источник: [habrahabr<sup>\[7\]</sup>](#)

Перевод материала Мартина Фаулера. В статье обсуждается общий подход к архитектуре UI и приводятся подробные описания таких шаблонов проектирования, как MVC, MVP, Presentation Model, Forms and Controls, Humble View, Passive View. Статья неплохо прочищает мозг. Для того, чтобы не упустить ни единого нюанса, решил заняться переводом.

Вообще говоря, приседал долго, хотел сделать все сразу и быстро. Пальцем в небо. Иногда ~~меня разбивал радикулит~~ подступали майлстоуны по проекту и я откладывал перевод в долгий ящик. Или еще что-нибудь мешало. Короче, я все сразу не осилил и, чтоб добру не пропадать, решил выкладывать перевод по параграфам. Сейчас перевел половину, половина же осталась.

Я не профессиональный переводчик и мог что-то неправильно понять (и даже кое-где сделал пометки в скобках), но вы в любом случае обладаете возможностью прочитать статью в [оригинале<sup>\[8\]</sup>](#). Надеюсь, что перевод такой интересной статьи поможет кому-то улучшить свои навыки и расширит кругозор.

Графические пользовательские интерфейсы стали привычной частью пейзажа разработки ПО, как для пользователей, так и для разработчиков. Если взглянуть на них с точки зрения дизайна, они представляют с собой особый набор проблем системного проектирования — проблем, у которых есть различные, но похожие решения.

Мой интерес в том, чтобы обозначить общие и полезные шаблонные решения, которые могут быть использованы программистами при разработке толстых (rich-client) клиентов. Я видел различные архитектуры как при осуществлении проектных инспекций, так и написанные в более «постоянном» стиле. В этих архитектурах есть полезные решения, однако, описывать их — непростое дело. Возьмем, к примеру, Model-View-Controller. Это решение часто воспринимается как шаблон, но я не вижу особой пользы воспринимать его, как шаблон, хотя бы потому что идеи, в него заложенные, различны по своей сути. Разные люди читают про MVC в различных источниках и воспринимают его идеи по-разному, но называют эти идеи одинаково — «MVC». Это приводит к замешательству и непониманию MVC, будто бы люди узнавали про него через «сломанный телефон».

В этой работе я хочу привести некоторое количество интересных архитектурных решений и описать свое видение их самых интересных свойств. Надеюсь, что объясню все в понятном стиле.

В некотором роде, вы можете воспринять эту работу как экскурс в историю различных решений программирования UI в течение нескольких лет. Однако, я должен вас предостеречь. Понимание архитектур не простое дело, особенно когда многие из них изменяются или умирают. Проследить некоторые идеи еще сложнее, потому что разные люди видят в них свое, отличное от того, что видят другие. В частности, я не провел глубокого и тщательного анализа тех архитектур, которые буду описывать. То, что я сделал, можно обозначить как общее описание дизайнов систем. Если я что-то пропустил — это значит, что я об этом просто не знаю. Поэтому, не воспринимайте мои описания, как руководство. Более того, некоторые вещи я специально оставляю упрощенными, если они незначительны. Помните, что моя основная задача — описать решения задачи, а не историю этих решений.

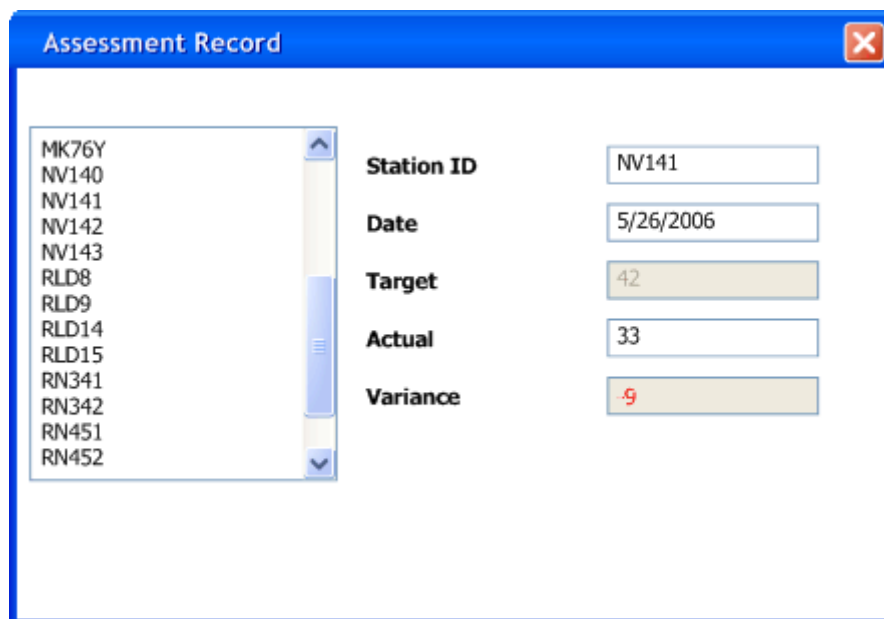
(Однако, я немного себе противоречу — в плане того, что у меня был доступ к работающей версии Smalltalk-80, чтобы я мог изучить MVC. Опять же, я не могу сказать, что это исследование было тщательным, но оно позволило мне понять некоторые аспекты решения, которые другие описания объяснить не смогли — что, в свою очередь, отталкивает меня от намерений преподносить свое описание других архитектур как единственно верное. Если вы хорошо знакомы с какой-либо из таких архитектур и видите, что я в чем-то важном не прав или что-то упустил, то я бы хотел об этом знать. Еще я думаю, что более тщательный анализ того, про что я буду говорить, послужит хорошим предметом изучения в ВУЗах.)

*(Фаулер очень рьяно оперирует понятием control, что можно оставить как «контроль» или перевести как «элемент управления». Увы, в русском языке нет слова «контроль» (даже несмотря на то, что оно повсеместно используется в сленге), и, чтобы не мучить вам глаза, далее я буду называть элементы управления просто «элементами». — прим. перев.)*

Я начну это исследование с архитектуры, которая сама по себе проста и хорошо знакома для большинства разработчиков. У нее нет общего обозначения, поэтому в целях общих правил этой работы, я буду называть ее «Формы и элементы». Эта архитектура хорошо известна прежде всего потому, что она предоставлялась средами разработки клиент-серверных приложений, вроде Visual Basic, Delphi, Powerbuilder. Несмотря на то, что она отвергается такими фанатами крутых архитектурных решений, как я, ее все равно продолжают повсеместно использовать.

Для того, чтобы ее понять (и понять другие архитектурные решения), я буду использовать общий для всех пример. В Новой Англии, где я живу, существует государственная программа слежения за количеством мороженого в воздухе. Если концентрация слишком мала, это означает, что мы едим слишком мало мороженого, что, в свою очередь, представляет большой риск для нашей экономики и общественного порядка. (Мне нравится использовать примеры не более реальные, чем вы находите в других книгах, подобной этой).

Чтобы следить за нашим мороженым, во всех штатах Новой Англии правительство построило специальные станции слежения. Используя сложную атмосферную модель, департамент назначает целевое (target) значение для каждой станции. После этого персонал отправляется на станции производить оценку концентрации мороженого. UI ниже позволяет им выбрать станцию и ввести дату замера (date) и фактическое значение (actual) концентрации. Система высчитывает отклонение (variance) между целевой и фактической величиной. Более того, если фактическое значение меньше, чем 90% от целевого, отклонение подсвечивается красным, а если больше, чем 105%, то зеленым.



Station ID	Date	Target	Actual	Variance
MK76Y				
NV140				
NV141	5/26/2006	42	33	-9
NV142				
NV143				
RLD8				
RLD9				
RLD14				
RLD15				
RN341				
RN342				
RN451				
RN452				

Рисунок 1: UI, который я буду использовать для примера

Из этого представления можно выделить две важных детали. Форма (form) является специфичной для приложения, но она использует элементы (controls), которые являются общими для других форм. Большинство сред разработок GUI обладают неким набором общих элементов, которые можно использовать при разработке приложений. Мы имеем возможность создавать свои элементы (и это хорошая идея), однако, формы приложения и повторно используемые элементы все равно друг от друга отличаются. Другими словами, созданные элементы могут быть использованы во множестве различных форм.

Форма обладает двумя основными обязанностями:

- Планировка экрана (Screen layout): форма определяет расположение элементов на экране, объединяя их в иерархическую структуру.
- Логика формы (Form logic): форма определяет поведение, которое не может быть запрограммировано в элементах

Большинство сред разработки GUI предоставляет разработчику средства определить планировку экрана, например, графический редактор, который позволяет перетаскивать и бросать (drag and drop) элементы на форму. Этого, в принципе, достаточно, чтобы создать планировку и легко настроить расположение элементов на форме (хотя, этот способ не всегда самый лучший — вернемся к этому позже.)

Элементы отображают данные — в нашем случае, это чтение. Данные, весьма вероятно, приходят откуда-то из другого места. Для нашего случая предположим, что из БД SQL, поскольку она является основным источником данных для упомянутых сред разработки. В большинстве случаев, в приложении существует три копии используемых данных:

- Первой копией является сама база данных. Эта копия отображает записи данных в фактическом хранилище данных — БД, поэтому я буду называть их (данные) состоянием записи (**record state**). Обычно, используя различные механизмы, к состоянию записи могут получить доступ сразу несколько человек.
- Вторая копия хранится в памяти приложения в виде набора записей (Record Set<sup>[9]</sup>). Большинство сред разработки предоставляют легкие способы получить набор записей. Данные в них актуальны в течение одной конкретной сессии между приложением и базой данных, поэтому я назову набор записей состоянием сессии (**session state**). Существенной характеристикой состояния сессии является то, что оно создает временную локальную копию данных, с которой работает пользователь до тех пор, пока он не сохранит или сделает commit в БД — при этом, состояние сессии объединяется (merge) с состоянием записи. Я не хочу подробно разбирать способы, позволяющие произвести объединение: я уже сделал это в [P of EAA].<sup>[10]</sup>
- Третья копия хранится в самих GUI-компонентах. Строго говоря, это данные, которые пользователь видит на экране, так что я буду называть их состоянием экрана (**screen state**). Очень важным является то, как UI производит синхронизацию между состоянием сессии и состоянием экрана.

Синхронизация состояния экрана и состояния сессии — это очень важная задача. Средством, которое может ее решить, является привязка данных (Data Binding<sup>[11]</sup>). Идея привязки данных состоит в том, что изменение, произошедшее в данных элемента или данных записи в наборе записей незамедлительно распространяется друг на друга. То есть, если я ввожу текущую концентрацию на экран, элемент «текстовое поле» обновляет значение в колонке для соответствующей записи данных в наборе записей (record set).

Реализация привязки данных является достаточно сложным занятием, потому что вам нужно избегать таких циклов, например, где изменение элемента изменяет набор данных, который обновляет элемент, который обратно обновляет набор данных, который вновь обновляет элемент... Поток использования (flow of usage) помогает избежать таких циклов — мы

загружаем данные из состояния сессии в состояние экрана только тогда, когда экран открывается, после этого любое изменение экрана распространяется обратно в состояние сессии. Обновлять состояние сессии тогда, когда открывается экран — необычно и не нужно. В результате, привязка данных может быть не полностью в обе стороны — она может быть ограничена начальной загрузкой данных и обработкой изменений экрана в состояние сессии.

Привязка данных, вообще говоря, реализует большинство функциональности клиент-серверного приложения. Если я изменю значение фактической концентрации, значение в колонке обновится. Даже выбор другой станции вызовет изменение выбранной записи в наборе записей, что, в свою очередь, заставит другие элементы обновиться.

В основном, описанное поведение уже реализовано разработчиками каркасов приложений, которые предусмотрели и предоставили средства быстро и несложно удовлетворить основные нужды программистов. В частности, одним из способов является явная установка некоторых характеристик элементов управления, которые обычно называют «свойствами». Элемент читает строковое название колонки набора данных, которое было указано разработчиком через простой редактор свойств и привязывается к ней.

Использование привязки данных, с нужной параметризацией, может помочь вам реализовать почти всю функциональность приложения. Однако, привязка данных не может реализовать ВСЮ функциональность — почти всегда в приложении существует логика, которая не попадает под опции параметризации. В нашем случае, расчет отклонения представляет собой пример того, что не может быть реализовано через привязку данных — из-за того, что такая функциональность специфична для приложения и реализуется в самой форме.

Для того, чтобы ее реализовать, форма должна знать, когда пользователь закончил ввод значения фактической концентрации, а это, в свою очередь, значит, что элемент управления должен вызвать какое-то поведение формы. Такая логика подразумевает инверсию управления (*Inversion of Control*). (в оригинале «*This is a bit more involved than taking a class library and using it through calling it as Inversion of Control is involved.*» — прим. перев.)

Существуют различные методы решения указанной проблемы. Обычно, среды разработки клиент-серверных приложений обладают системой представления событий. Каждый элемент управления имеет список событий, которые он может запускать. Любой внешний объект обладает возможностью сказать элементу, что его интересует событие. В этом случае, когда событие элемента произойдет, элемент передаст управление такому объекту. По существу, данное описание всего лишь переформулировка шаблона [Observer<sup>\[12\]</sup>](#), где форма следит за элементом. Каркас приложения позволяет разработчику формы написать код в методе, который будет исполнен после возникновения события. То, каким образом возникает связь между этим методом и событием, зависит от конкретной платформы и не важно в данном случае. Важно то, что такой механизм существует и он работает.

Когда форма получила управление, она может делать все, что ей нужно. Она может выполнить какие-то действия, изменить элементы управления, и, основываясь на привязке данных, провести сделанные изменения обратно в состояние сессии.

Такой механизм нужен еще и потому, что привязка данных не всегда доступна. В мире очень много элементов управления окнами и не все из них поддерживают привязку. Если привязки нет, то заниматься синхронизацией данных между состояниями должна форма. В частности, она может сама вытащить данные из набора данных и уложить их в элементы управления. Когда требуется сохранить данные обратно, она может сделать это таким же образом по событию нажатия кнопки «Сохранить».

Давайте посмотрим, каким образом будет происходить редактирование текущего значения концентрации, в случае, когда доступна привязка данных. Форма хранит ссылки на элементы управления. Для каждого элемента будет своя ссылка, однако мне интересны только текстовые поля для фактического значения, значения отклонения и целевого значения.

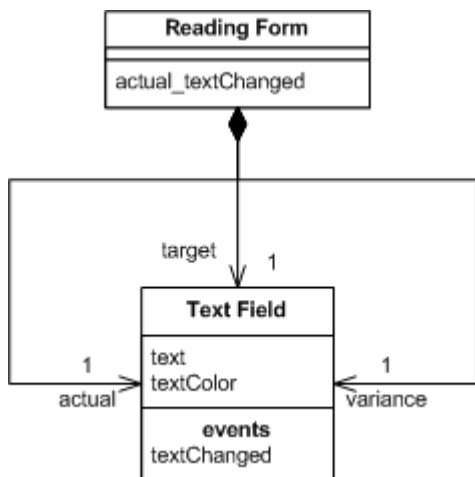


Рис. 2: Диаграмма классов для форм и элементов

Текстовое поле объявляет событие, когда изменился его текст. На этапе инициализации форма собирает себя и подписывается на событие, привязывает свой метод к нему — вот, `actual_textChanged`.

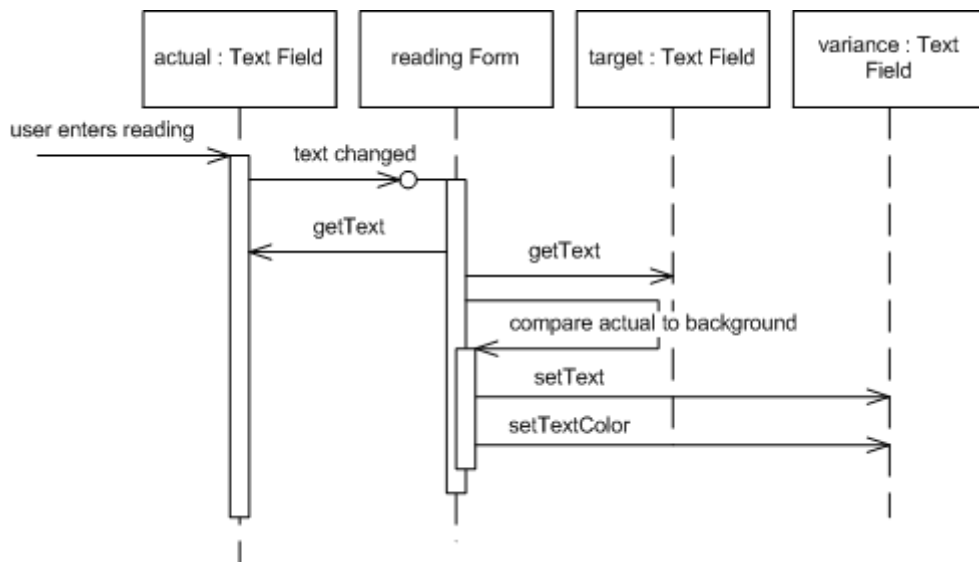


Рисунок 3: Диаграмма последовательности изменения текста.

Когда пользователь изменяет значение фактической концентрации, текстовое поле запускает свое событие и, используя особую магию каркаса приложения, выполняет метод формы `actual_textChanged`. Этот метод получает фактическое и целевое значение из элементов формы, производит вычисления и устанавливает отклонения в текстовое поле разницы. Так же он вычисляет, каким цветом нужно подсветить полученное значение и подсвечивает его.

Теперь можно подвести итоги:

- Программисты разрабатывают формы, которые своей функциональностью зависят от специфики приложения, но используют общие для всех форм элементы управления.
- Форма описывает расположение своих элементов управления.
- Форма следит за событиями элементов управления и обрабатывает их.
- Простые изменения данных синхронизируются при помощи привязки данных.
- Сложные изменения производятся в методах — обработчиках событий.

Следующая часть [тут](#)<sup>[13]</sup>.



[Распечатать статью](#)<sup>[14]</sup>

[Вернуться в раздел: Programming](#)<sup>[15]</sup> / [Теория разработки](#)<sup>[16]</sup>

Реклама:

## References

1. [↑ Programming](#)  
( <http://www.rusdoc.ru/reviews/programming> )
2. [↑ Теория разработки](#)  
( <http://www.rusdoc.ru/reviews/programming/mobile/theory/> )
3. [↑ gui](#)  
( <http://www.rusdoc.ru/tags/gui> )
4. [↑ юзабилити](#)  
( <http://www.rusdoc.ru/tags/%FE%E7%E0%E1%E8%EB%E8%F2%E8> )
5. [↑ mvc](#)  
( <http://www.rusdoc.ru/tags/mvc> )
6. [↑ mvp](#)  
( <http://www.rusdoc.ru/tags/mvp> )
7. [↑ habrahabr](#)  
( <http://habrahabr.ru/> )
8. [↑ оригинале](#)  
( <http://www.rusdoc.ru/go.php> )
9. [↑ Record Set](#)  
( <http://www.rusdoc.ru/go.php> )
10. [↑ \[P of EAA\].](#)  
( <http://www.martinfowler.com/books.html> )
11. [↑ Data Binding](#)  
( <http://www.rusdoc.ru/go.php> )
12. [↑ Observer](#)  
( <http://www.amazon.com/exec/obidos/tg/detail/-/0201633612> )
13. [↑ тут](#)  
( <http://www.rusdoc.ru/go.php> )
14. [↑ Распечатать статью](#)  
( <http://www.rusdoc.ru/articles/18358/print/> )
15. [↑ Programming](#)  
( <http://www.rusdoc.ru/reviews/programming> )
16. [↑ Теория разработки](#)  
( <http://www.rusdoc.ru/reviews/programming/mobile/theory/> )

Отрывок из веб-страницы: *Martin Fowler — GUI Architectures. Часть 1. :: РУССКИЕ ДОКУМЕНТЫ*  
<http://www.rusdoc.ru/articles/18358/>