GUI 架构

Definition of *Software Architecture*

网上查阅资料，搜到了**这篇文档(Software Architecture Guide)**，文档作者对软件架构说明了 3 个定义：

> - a better view of architecture was **the shared understanding that the expert developers have of the system design.**
> - “the design decisions that need to be made early in a project”, but Ralph complained about this too, saying that it was more like **the decisions you wish you could get right early in a project**.
> - His conclusion was that **“Architecture is about the important stuff. Whatever that is”**.

也就是说软件架构没有太明确的定义，有的人认为是专家对整个系统的设计，也有观点认为就是整个项目开始前这些设计理念的设计者认为的该有的“打开方式”，结论就是：不管是啥，就是跟重要的事情有关的东西。

我也就理解为 something important you should decide and design before the whole project, whatever comes to your mind.

在文档下面有个关于 **GUI Architectures** 的链接，虽然说 2006 年的可能有点老，毕竟 User Interface 的设计和审美随时间会不断变换，但是这篇文章不讲美观，只讲架构，还是值得深入学习的。

GUI Architectures

文章提到了几种不同的 UI 架构，最简单的先从表格入手。

表单/表格

结构

> The form contains two main responsibilities:
>
> - Screen layout: defining the arrangement of the controls on the screen, together with their hierarchic structure with one other.
> - Form logic: behavior that cannot be easily programmed into the controls themselves.

表格包含2个部分：屏幕布局和表格逻辑。

数据备份

而控件中显示了数据。然后讲到了 SQL 数据备份：

> In most situations there are three copies of the data involved:
>
> - One copy of data lies in the database itself. This copy is the lasting record of the data, so I call it the **record state**. The record state is usually shared and visible to multiple people via various mechanisms.

> - A further copy lies inside in-memory **Record Sets** within the application. Most client-server environments provided tools which made this easy to do. This data was only relevant for one particular session between the application and the database, so I call it **session state**. Essentially this provides a temporary local version of the data that the user works on until they save, or commit it, back to the database - at which point it merges with the record state. I won't worry about the issues around coordinating record state and session state here: I did go into various techniques in [**P of EAA**].
> - The final copy lies inside the GUI components themselves. This, strictly, is the data they see on the screen, hence I call it the **screen state**. It is important to the UI how screen state and session state are kept synchronized.

数据绑定

然后提到了保持前端界面状态和同步更新，以此用到的工具就是 数据绑定(**Data Binding**)。就是说任何更新都应该立即同步。

然后就开始讲数据绑定的细节：

> In general data binding gets tricky because if you have to avoid cycles where a change to the control, changes the record set, which updates the control, which updates the record set.... The flow of usage helps avoid these - we load from the session state to the screen when the screen is opened, after that any changes to the screen state propagate back to the session state. It's unusual for the session state to be updated directly once the screen is up. As a result data binding might not be entirely bi-directional - just confined to initial upload and then propagating changes from the controls to the session state.

客户端服务器工具包

表格的任一字段的改变都应该被提醒，然后就提到了 client-server toolkits 。读完之后就是在说 event listener。

Diagram

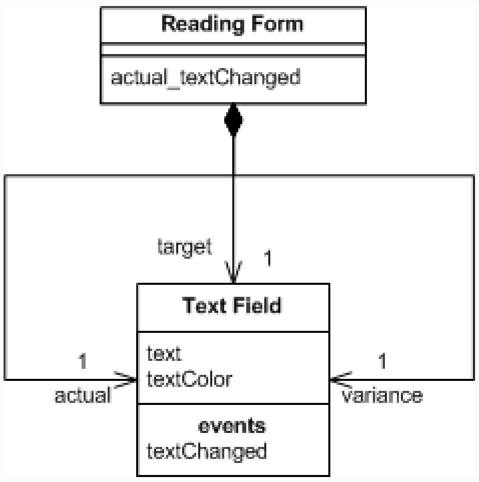

Figure 2: Class diagram for forms and controls

> The text field declares an event for text changed, when the form assembles the screen during initialization it subscribes itself to that event, binding it a method on itself - here `actual_textChanged`.

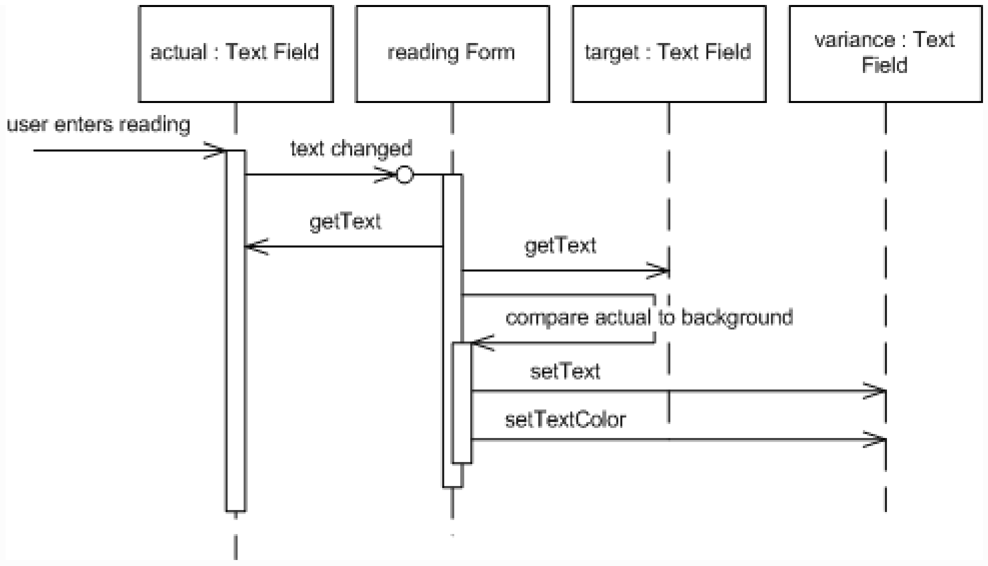

Figure 3: Sequence diagram for changing a genre with forms and controls.

When the user changes the actual value, the text field control raises its event and through the magic of framework binding the `actual_textChanged` is run. This method gets the text from the actual and target text fields, does the subtraction, and puts the value into the variance field. It also figures out what color the value should be displayed with and adjusts the text color appropriately.

发生事件后的各种细节，图表显示更清晰。

总结

- Developers write application specific forms that use generic controls.
- The form describes the layout of controls on it.
- The form observes the controls and has handler methods to react to interesting events raised by the controls.
- Simple data edits are handled through data binding.
- Complex changes are done in the form's event handling methods.

模型视图控制器 Model View Controller (MVC)

开头说了许多自认为是 MVC 但事实上不是的。

分离演示

MVC 中最具有影响力的框架，分离演示 (Separated Presentation)，明确划分了模拟我们对真实世界感知的**域对象**(Domain Objects)和我们在屏幕上看到的 GUI 元素的**演示对象**(Presentation Objects)。

域对象也称之为模型(Model)对象，完全忽视 UI 。

The presentation part of MVC is made of the two remaining elements: view and controller.

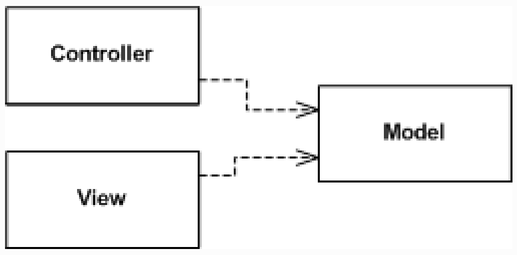

Figure 4: Essential dependencies between model, view, and controller. (I call this essential because in fact the view and controller do link to each other directly, but developers mostly don't use this fact.)

流同步和观察者同步

> These two styles I describe as patterns: **Flow Synchronization** and **Observer Synchronization**. These two patterns describe alternative ways of handling the triggering of synchronization between screen state and session state. Forms and Controls do it through the flow of the application manipulating the various controls that need to be updated directly. MVC does it by making updates on the model and then relying of the observer relationship to update the views that are observing that model.

总结

> - Make a strong separation between presentation (view & controller) and domain (model) - **Separated Presentation**.
> - Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model.
> - Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - **Observer Synchronization**.

视觉工作应用模型 VisualWorks Application Model

VisualWorks 提出了一种应用模型的结构，一种类似于演示模型的结构，以此解决 MVC 难以应对视图逻辑和视图状态的难题。

关键点是把属性转变成对象。然后就一直在讲这种映射关系。

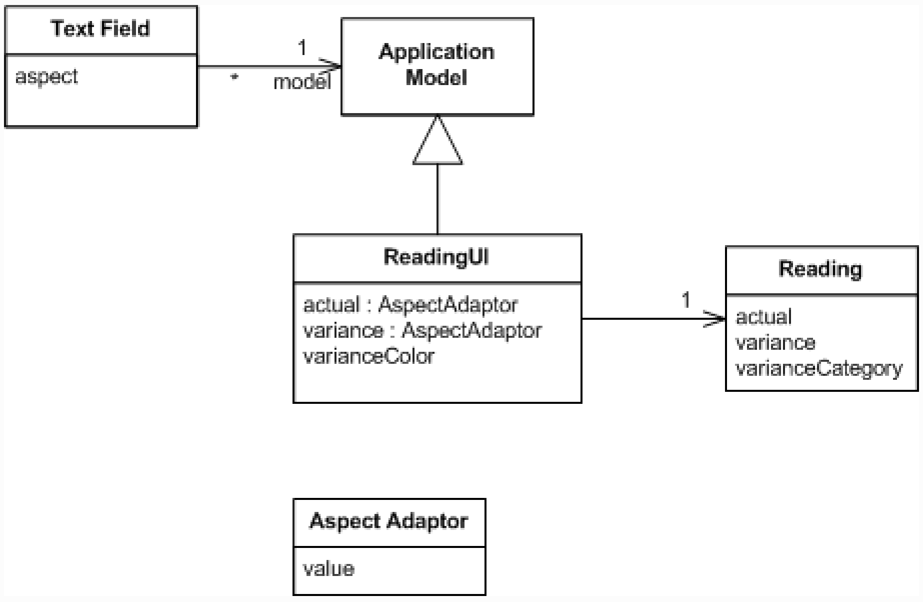

Figure 9: Class diagram for visual works application model on the running example

总结

> - Followed MVC in using **Separated Presentation** and **Observer Synchronization**.
> - Introduced an intermediate application model as a home for presentation logic and state - a partial development of **Presentation Model**.
> - Widgets do not observe domain objects directly, instead they observe the application model.

> - Made extensive use of Property Objects to help connect the various layers and to support the fine grained synchronization using observers.
> - It wasn't the default behavior for the application model to manipulate widgets, but it was commonly done for complicated cases.

模型视图演示者 Model-View-Presenter (MVP)

对比 MVP 和 MVC：

> - Forms and Controls: MVP has a model and the presenter is expected to manipulate this model with **Observer Synchronization** then updating the view. Although direct access to the widgets is allowed, this should be in addition to using the model not the first choice.
> - MVC: MVP uses a **Supervising Controller** to manipulate the model. Widgets hand off user gestures to the **Supervising Controller**. Widgets aren't separated into views and controllers. You can think of presenters as being like controllers but without the initial handling of the user gesture. However it's also important to note that presenters are typically at the form level, rather than the widget level - this is perhaps an even bigger difference.
> - Application Model: Views hand off events to the presenter as they do to the application model. However the view may update itself directly from the domain model, the presenter doesn't act as a **Presentation Model**. Furthermore the presenter is welcome to directly access widgets for behaviors that don't fit into the **Observer Synchronization**.

总结

> - User gestures are handed off by the widgets to a **Supervising Controller**.
> - The presenter coordinates changes in a domain model.
> - Different variants of MVP handle view updates differently. These vary from using **Observer Synchronization** to having the presenter doing all the updates with a lot of ground in-between.

Humble View

> **The Humble Dialog Box** paper uses a presenter, but in a much deeper way than the original MVP. Not just does the presenter decide how to react to user events, it also handles the population of data in the UI widgets themselves. As a result the widgets no longer have, nor need, visibility to the model; they form a **Passive View**, manipulated by the presenter.
>
> This isn't the only way to make the UI humble. Another approach is to use **Presentation Model**, although then you do need a bit more behavior in the widgets, enough for the widgets to know how to map themselves to the **Presentation Model**.