GUI ARCHITECTURES

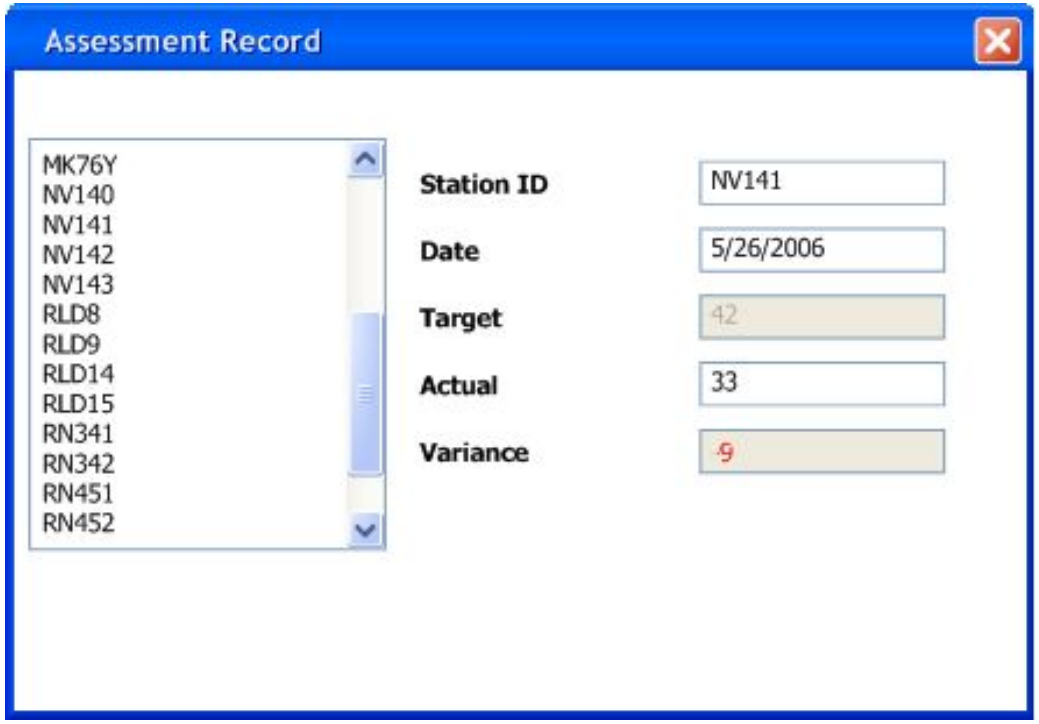
SWE 4601

https://martinfowler.com/eaaDev/uiArchs.html

FORMS AND CONTROLS

- It is encouraged by client-server development environments in the 90's - tools like Visual Basic, Delphi, and Powerbuilder.

- Scenario:
  - monitors the amount of ice-cream particulate in the atmosphere.
  - staffers go out on an assessment where they go to various stations and note the actual ice-cream particulate concentrations.
  - The system highlights the variance in red when it is 10% or more below the target, or in green when 5% or more above the target.



FORMS AND CONTROLS

- The form is specific to our application, but it uses controls that are generic.
- 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.
- Most GUI development environments allow the developer to define screen layout with a graphical editor that allows you to drag and drop the controls onto a space in the form.

FORMS AND CONTROLS

- The controls display data - in this case about the reading.
- let's assume a SQL database as data source
- In most situations there are three copies of the data involved:
  - **record state:** One copy of data lies in the database itself. This copy is the lasting record of the data.
  - **session state:** The data relevant for one particular session between the application and the database. Essentially this provides a temporary local version of the data that the user works on until they save,
  - **screen state:** The final copy lies inside the GUI components themselves.
- It is important to the UI how screen state and session state are kept synchronized.

FORMS AND CONTROLS

- Data Binding: The idea was that any change to either the control data, or the underlying record set was immediately propagated to the other.
- Data Binding handles much of the functionality of a client-sever application pretty nicely.
- In the example: changing the selected station alters the currently selected row in the record set, which causes the other controls to refresh.
- Using data binding, with the right kind of parameterization, can take you a long way. However it can't take you all the way - there's almost always some logic that won't fit with the parameterization options. In this case calculating the variance is an example of something that doesn't fit in this built in behavior - since it's application specific it usually lies in the form.

FORMS AND CONTROLS

- Each control had a list of events it could raise. Any external object could tell a control that it was interested in an event.
- Essentially this is just a rephrasing of the Observer pattern where the form is observing the control.
- If data binding isn't present then it's up to the form to carry out the synchronization. This could work by pulling data out of the record set into the widgets initially, and copying the changed data back to the record set when the save button was pressed.

FORMS AND CONTROLS

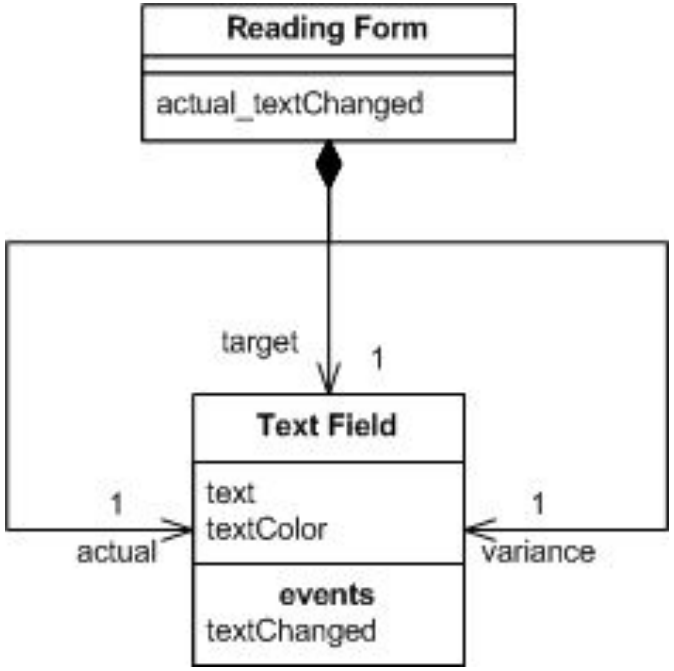


Figure 2: Class diagram for forms and controls

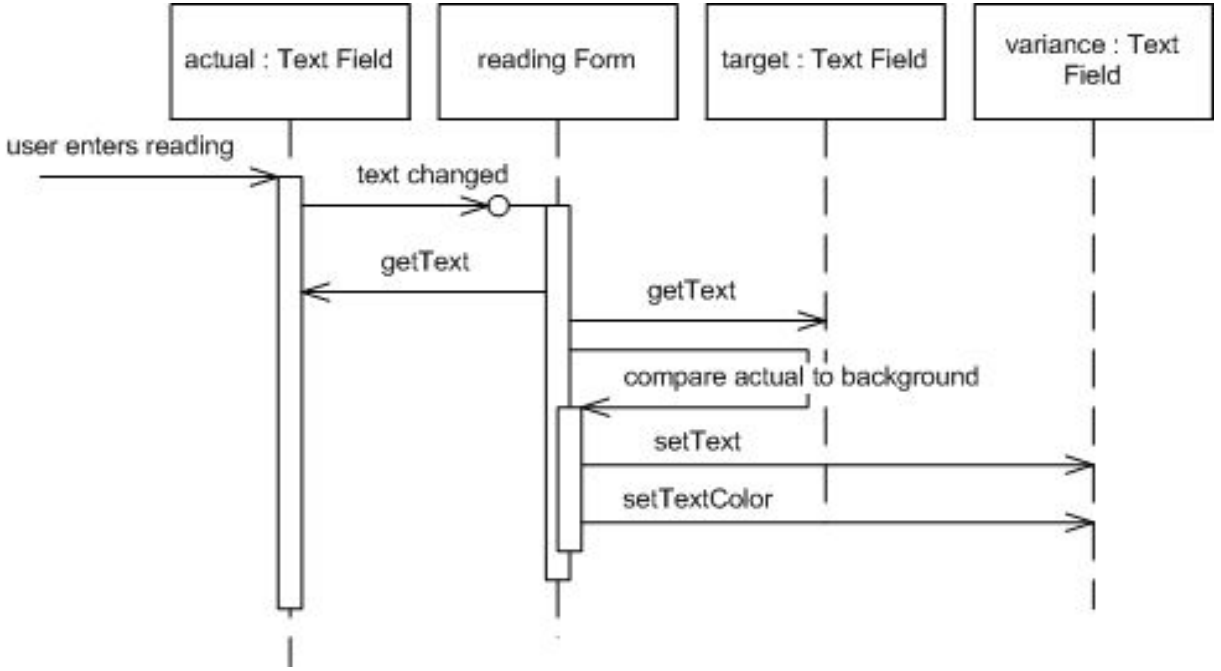


Figure 3: Sequence diagram for changing a genre with forms and controls.

FORMS AND CONTROLS

We can summarize the architecture with a few soundbites:

- 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.

MODEL VIEW CONTROLL

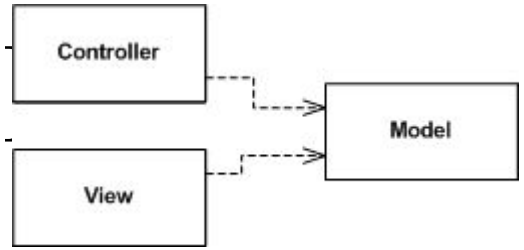


Figure 4: Essential dependencies between model, view, and controller.

- At the heart of MVC, and the idea that was the most influential is separated presentation
- **Separated Presentation**
  - Ensure that any code that manipulates presentation only manipulates presentation, pushing all domain and data source logic into clearly separated areas of the program.
  - The idea behind **Separated Presentation** is to make a clear division between
    - domain objects that model our perception of the real world, and
    - presentation objects that are the GUI elements we see on the screen
  - allowing many applications to be manipulated through both a graphical and command-line interface.
  - The controller's job is to take the user's input and figure out what to do with it.

MODEL VIEW CONTROLL

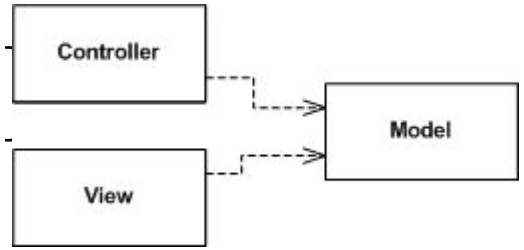


Figure 4: Essential dependencies between model, view, and controller.

- our assessment UI example
  - Model: Reading
  - View: Text Field
  - Controller: Text Field Controller
- Forms and Controls assumed that most people wanted to easily manipulate data from a relational database, MVC assumes we are manipulating regular objects.

MODEL VIEW CONTROLLER

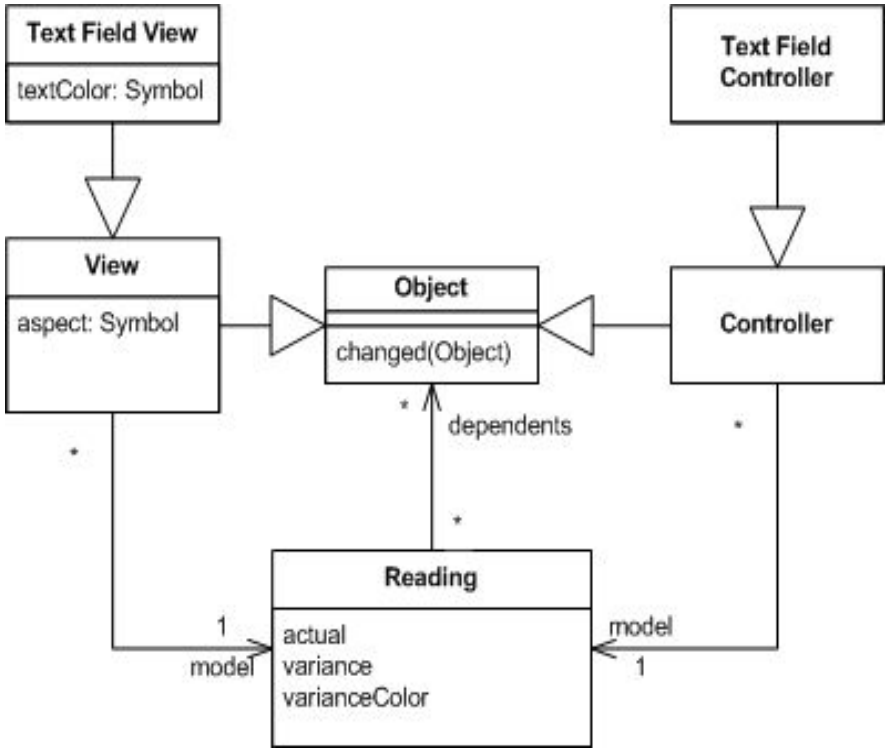


Figure 5: Classes for an MVC version of an ice-cream monitor display

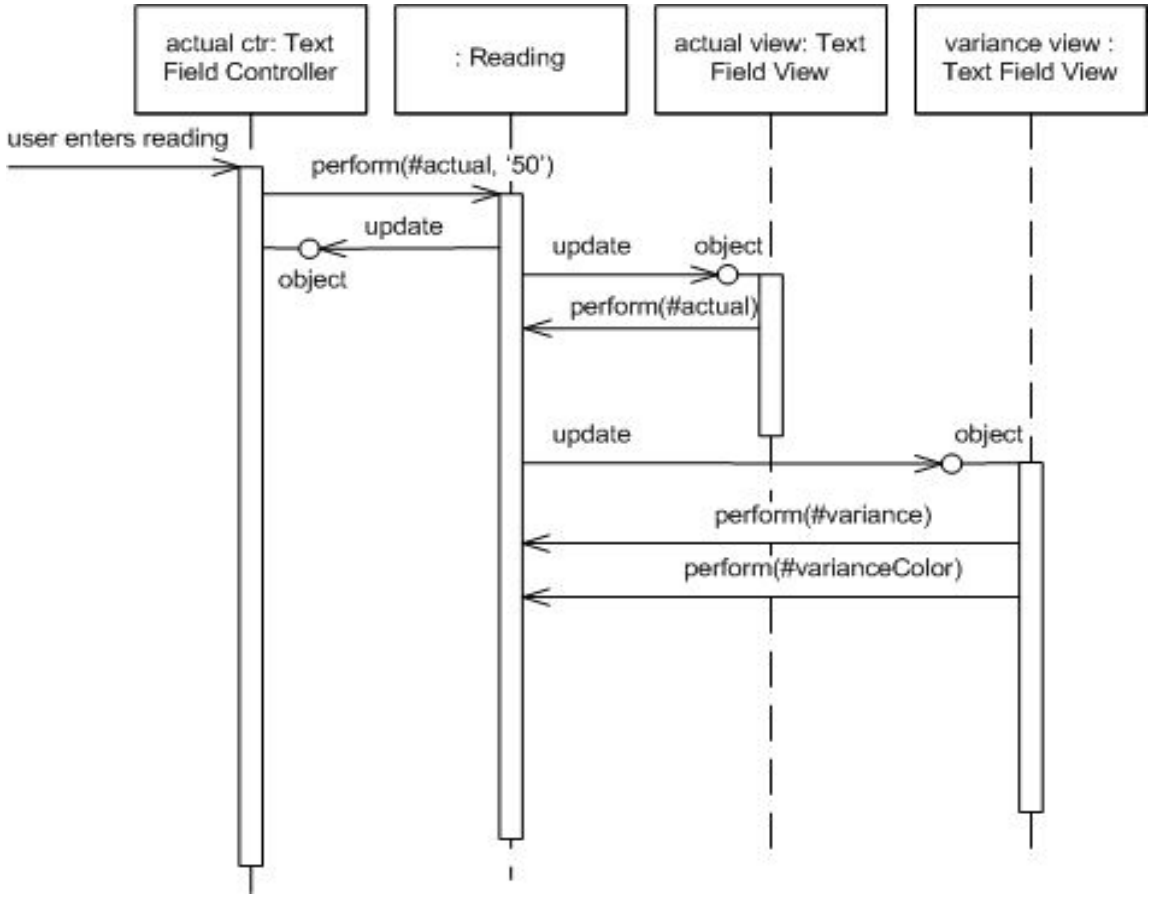


Figure 6: Changing the actual value for MVC.

MODEL VIEW CONTROLLER

- Observers do occur in MVC, indeed it's one of the ideas credited to MVC. In this case all the views and controllers observe the model. When the model changes, the views react.
- In this case the actual text field view is notified that the reading object has changed, and invokes the method defined as the aspect for that text field - in this case #actual - and sets its value to the result.

MODEL VIEW CONTROLLER

- Flow Synchronization and Observer Synchronization. These two patterns describe alternative ways of handling the triggering of synchronization between screen state and session state.
- Flow Synchronization is even more apparent when data binding isn't present. If the application needs to do synchronization itself, then it was typically done at important point in the application flow - such as when opening a screen or hitting the save button.
- Observer Synchronization Synchronize multiple screens by having them all be observers to a shared area of domain data.

MODEL VIEW CONTROLLER

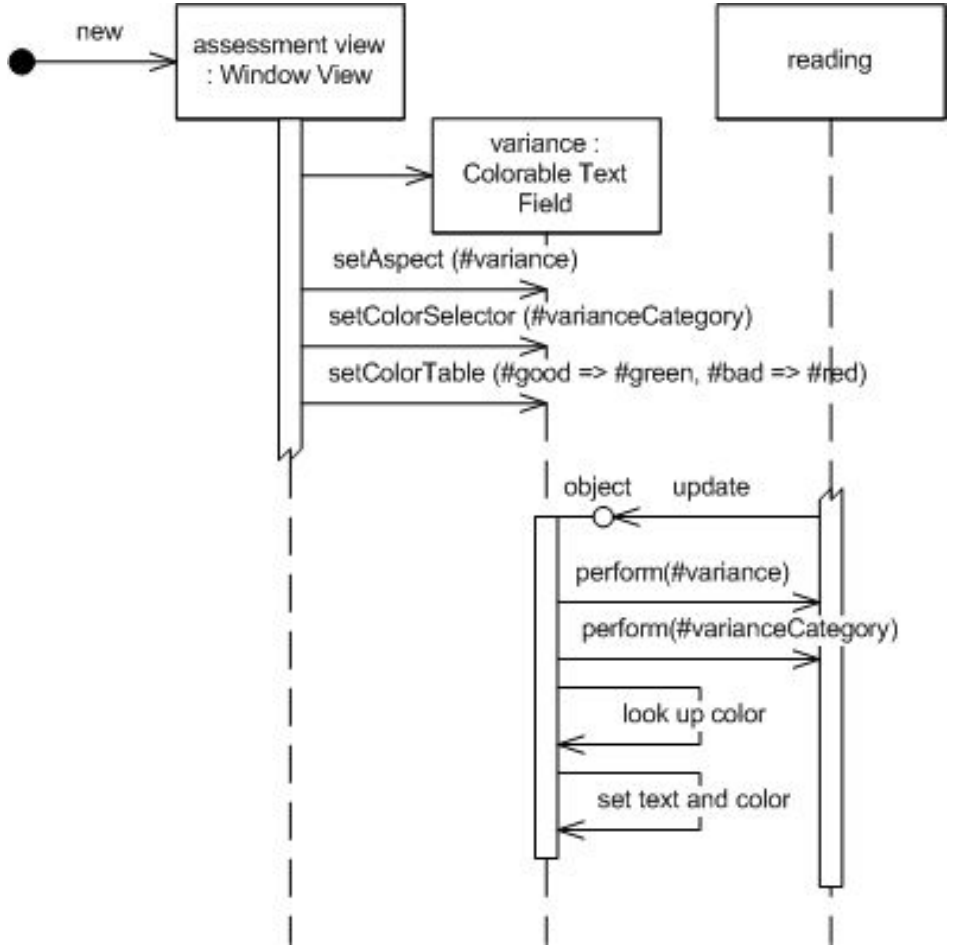


Figure 7: Using a special subclass of text field that can be configured to determine the color.

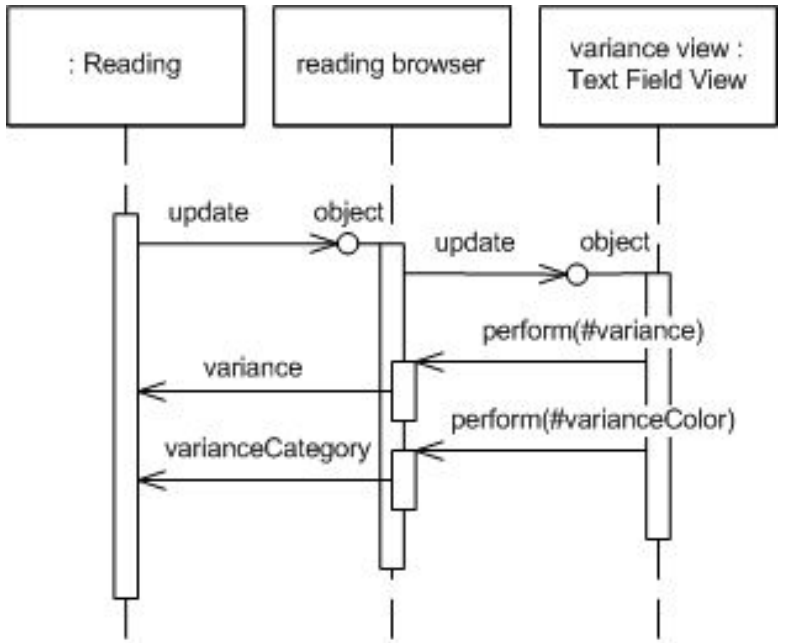


Figure 8: Using an intermediate Presentation Model to handle view logic

MODEL VIEW CONTROLLER

Simply MVC.

- 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.

MODEL-VIEW-PRESENTER (MVP)

- MVP is an architecture that first appeared in IBM and more visibly at Taligent during the 1990's. It's most commonly referred via the Potel paper.
- To approach MVP
  - On the one hand is the Forms and Controller architecture which was the mainstream approach to UI design,
  - on the other is MVC.
  - The Forms and Controls model provides a design that is easy to understand and makes a good separation between reusable widgets and application specific code.
  - What it lacks, and MVC has so strongly, is Separated Presentation and indeed the context of programming using a Domain Model.
- MVP as a step towards uniting these streams, trying to take the best from each.

MODEL-VIEW-PRESENTER (MVP)

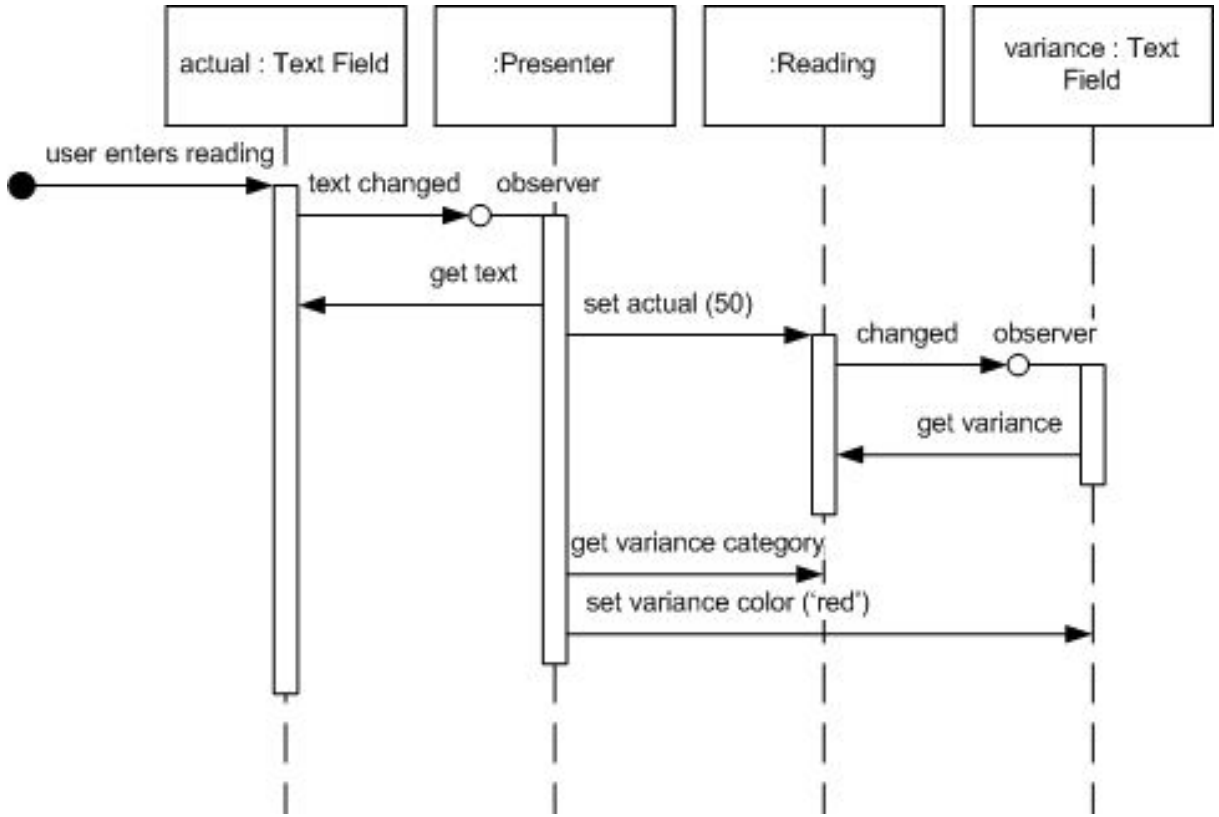


Figure 12: Sequence diagram of the actual reading update in MVP.