Subject: Re: MVC example
Posted by kohait00 on Thu, 04 Aug 2011 22:29:12 GMT
View Forum Message <> Reply to Message

MVC in upp is actually a very interesting topic to talk about, because it leads to proper structurizing of your app very quick. since i'm currently dealing with some related stuff in my project, i've been messing around a bit and will rewarm the thread  i dont plan to introduce MVC in upp, but to see and maybe show some hints on how/why upp is still able to be used with MVC patterns to a certain degree, though it is not quite easy. and we can't deny that sometimes kind-a-MVC is at least a bit appealing.

first things first: everything dependands on *what* you define to be your model. then, it points you what you define to be the view and last, what logic to lay as base (control).

if you take your whole upp application base to be the model (SQL backend, some sort of state data, which doesnt know anything of it beeing used or edited, etc) then it will logically draw you to the point to see the upp layout of the app as the view, beeing the Ctrl's the processors of user interaction (forwarded to the control to perform the logic on the model). MVC states, that a view is exchangeable, which is kind a irrelevant here, since it's your single application instance, except for if you want your app be a chameleon and *totaly* change its appearance. additionally, you normally got one single control in this case, which is, according to MVC pattern, your application logic, your current handling context logic. so far, upp quite well complies to MVC, except for one thing: the general notification mechanism from the model, which instructs one or multiple views to perform a refresh of the whole representation. in case of an upp application, this is sort of done with arbitrary means, and no more than one observer (the app view) is needed to be listening. so somehow, the view is triggered to perform a redraw. that's fine.

things significantly change, when you reduce your model, or split your app to be comprised of several models, with it's independant views. now the lack of notification means becomes evident, since more than one view could be rendering the same model and need to keep track of changes.

let's look at Display: it's not a true view according to MVC. it should have means of receive and forward userinput in some fashion to the controller. Display is view-only.
also, Display is rendering Value as 'Model', which lacks notification mechanism, though it refers to the same 'Data' internally, beeing sort of a shared ptr. a change to the data does not trigger all pointing-to instances of Value to signal a Display to redraw itself.. and how should 'control' come into play here

this quickly leads to looking at the Upp Ctrl's, since they are MVC all in one, just as koldo pointed out. the draw mechanism is the view, also processing the user input, which is used in a cotnrol mechanism internally. and a upp Ctrl renders and controls an internal form of Value as model, very generally speking. it can be accessesed with GetData/SetData. noone else acutally 'views' the same internal model of the Ctrl. the Ctrl can be notified of its internal model's change (after manually calling SetData, i.e.) from outside and have it redraw itself, by calling Refresh() (or UpdateRefresh() if more internal calculations are needed). on the other hand, an Upp Ctrl offers means of notification of change of the interlan model due to user handling. CtrlCore::WhenAction is a settable Callback, called each time some user input causes a change of the internal model. so the Ctrl itself, not the model, is notifying. it can be used to notify others based on that model. it

wont notify you of changes made through code api though (after SetData directly, i.e). you need to call Action() if you want so. but it will save you from a lot of loop call headache. see Ctrl design aspects in srcdoc of CtrlCore. we see: Ctrls are a sort of MVC in this sense: single-internal-M + single-parametrizable-V + single-internal-C.

now, using Ctrls as MVC has caveats. each Ctrl can store its own model (Value), and it notifies you of its own models' changes only. and the internal model is not neccessarily a Value, it can merely be set/read using Value, but it might map to other things internally, totally unrelated to Value that just happen to be exposed through the GetData/SetData gateway (but not neccessarily only through it). so best way is, to see upp::Ctrl's VC only, since they render a meta-model..which is stored internally and not fully known to anyone else, who also maybe wants render it.
also, the notification mechanism is very uncomfortable here. Callback usually only has a one-destination-call, so only one observer could be notified. one can add more observers with 'WhenAction << THISBACK(notifymecb);', but in therms of subscription handling, this is totally unsatisfactory. how to unsubscribe now? WhenAction.Clear() would flush other recepients as well. so definitely no notification mechanism here, suited for MVC.

*IF* using Value as the model of choice, which is a good idea, since it is a common data base in upp for all sorts of data, then one needs to have a change of perspective here. see the Ctrl internal Value as 'copy' of your model, which the Ctrl needs to have to be able to show something. the real model lies somewhere outside and needs to provide for the notification mechanism. now, the Ctrl becomes a View with means of receiving user input, which needs to have mapped its actions to some sort of control/logic, which in turn would cause the model change and would notify all other managed views (Ctrls) of that model of the changes and cause their redraw.

i came up with the Dispatcher package in bazaar, exactly due to such a problem. use case: an audio controlling software, operating on a database of objects (each object is a seperate model), which represent some parameters of an audio device (gain, mute, etc). each object can be represented in various ways, i.e mute object can be shown as a LED (readonly V, C is empty) on the main view of the device page, and have a button set on it, which can control it's state. (here the button is VC). so the need of a round-robin notification mechanism becomes evident. and upp does not provide for theese use cases.

but it provides a lot of very interesting helpers to accomplish even such a task, i.e. Value class, Callback class..doing some extra work, some extra template mapping classes lets you have such a use case completed. i am currently reworking this thing to be more general. so maybe it can be used by others in some way.

generally speaking, mirek is right. MVC is to unspecific for real world problems. that's why it's a pattern and not a library . the pattern applys well if the conditions are met (the need of a central model with many views). but the solution for this is oftentimes tied to the things you have to deal with anyway, API, class base, etc. so a general answer or even implementation is dificult here anyway. nevertheless, i will share the object implementation soon. maybe it's general enough for the crowd.. the base ide is this:

object is a Value based model, with a Dispatcher to notify observers. observers can be other controls (upp controls with extended capabilities) or some actors (mappers to actions or callbacks). an application, that has a portion of MVC-like problems, can use a dictionary of objects

and set up some VC parts on it, and wire up some actions to the objects. it's still not 100% MVC, but it's relatively general and flexible. this also enables to have a live work environment where one can select / create / move arbitrary Ctrl's and wire them with some objects to monitor and edit parameters/actions. the application internally can pre-wire some action to the objects' cataloge.

with this long post i hope to have condensed some thoughts on MVC and UPP, since it might be an ever regressing question. maybe some parts of the stuff discussed in this thread could go to 'wetting your appetite' or general questions.

sidenote:

android is MVC as well. the model is even stored in the cloud . one can obtain some providers to access the information needed (not all apps really do need such stuff). the control is carried out in the app itself, using the android controls as views..the dispatcher/notifier here is google itself