Hans Karlsen

MDriven The book

Doing effective Business – by taking control of Information

Hans Karlsen, Stockholm Sweden
[Datum]

Seeking the database with OCLps using ViewModels

To get anything done you need to find things. The normal software system has the same need. So how do we go about to declare a non-limiting multi variable user friendly seeker into a generic model driven system like the ones we build with MDriven?

This is what we need:

1. We need unrooted (as in not having anything to start with), persistent storage evaluated OCL expressions in order to execute a search

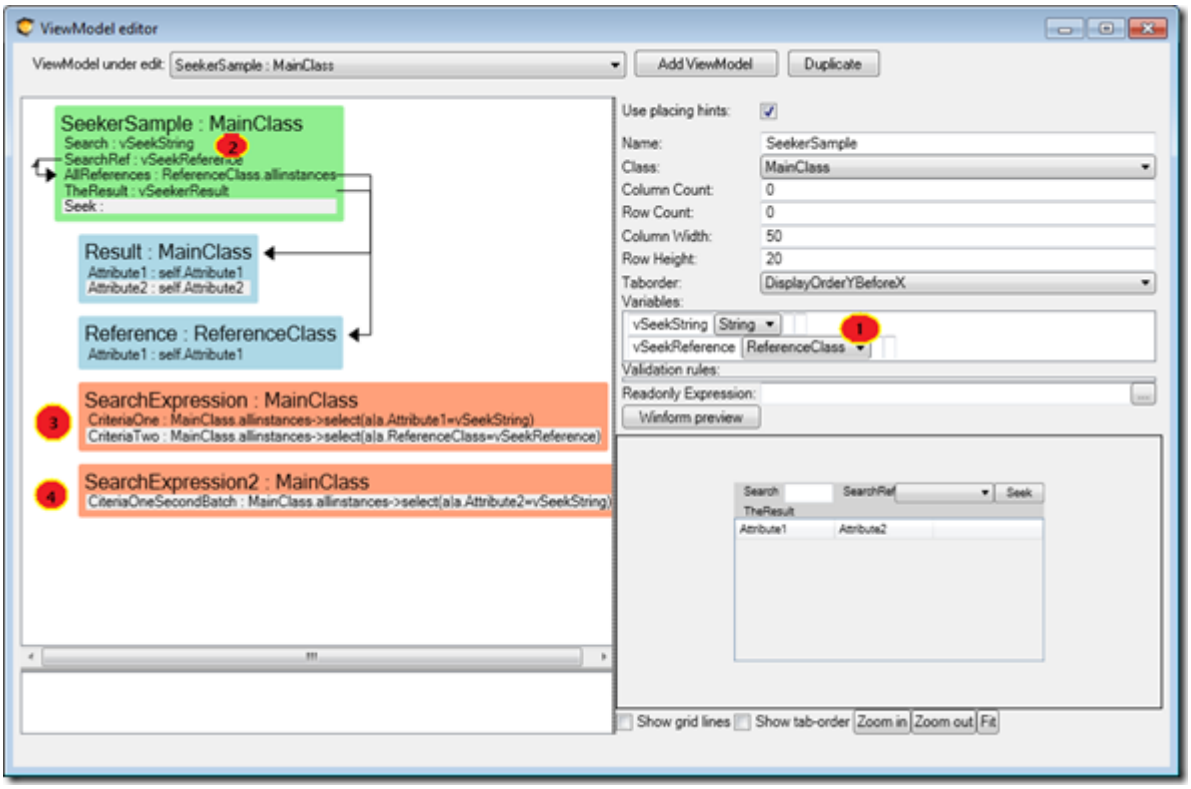
2. We need user input on how to limit the search

3. We need to allow for the user to use different limiting criteria's as he or she see fit ; after all the One-Field-Matches-Everything tactic that Google use does not really cut the mustard in enterprise applications. Users will want to limit the search on "Only this department", "Only things from yesterday" etc, and even in google you need to use an extended syntax to get this done.

4. We want to allow for multiple sets of search definitions per seeker interface – if the user does not get a hit using the filters with the first set, it is natural for the user to "try again" and then we want to use another set of search criteria's; this has been tested on real users and many find it intuitive and obvious that it should work this way.

Consider this model:

And this is how we can define a seeker:



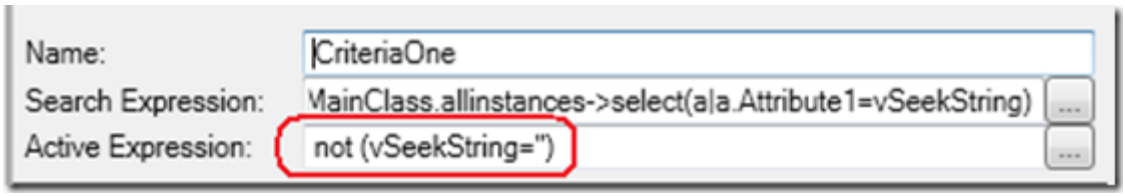
#1 We declare variables that holds the user input – one variable is a string, and the other is an Object reference of type ReferenceClass (from our model).

#2 We add Columns that use these variables so that we get some UI for the user to enter the criteria's into. The Reference value we set up as a PickList.

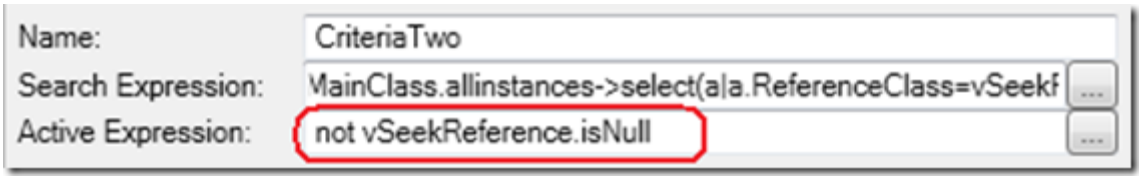
We create a SearchExpression - right click menu on view model column - Add nested – Add search expr. When adding the first search expression to viewmodel the designer will also add the default implementation details with a vSeekerResult variable and a SeekString. This is intended as a help and there is nothing magical about the added widgets. The only important thing in a seeker is that there is a variable named vSeekerResult and that it is of type collection<AnyClass> - this where the result of your search will go.

#3 Add two criterias – the result of the two criteria's will be intersected (on the server side). And here is an important fact: Since the result of the criteria's will be intersected we need some way to say if a criteria is Active or not – after all it is up to the user to limit the search on either or both of the two criteria's.

This is how the activation of a Criteria is done:



and



The Active Expression is optional, if you leave it blank it defaults to true – always on.

#4 The second batch of search expression is executed the second time the user clicks the search button BUT only if the search variables has not changed. You can have as many search batches as you need, and they are round-robin-used whenever the user clicks the search button with untouched variables. Whenever a variable is changed the Round-robin is reset and the first batch is used again.

Even if these rules may seem complex they are intuitive for the user – especially if you use the search batches to filter for the same data as the resulting columns show. For example; the user enters someone's first name, but your first batch filter on last name – the wrong people comes up for the first search, the user hits search again – now we use the second batch where you filter on first name – voila. This was just an example – you can just as well create a filter expression that unions the first name and the last name results ;*Person.allinstances->select(a|a.FirstName.SqlLike(vSeekString+'%'))->union(Person.allinstances->select(a|a.LastName.SqlLike(vSeekString+'%')))*

Another example might be that the users enters a number – first we try to match it with a product code, user hits search again, we try to match it with the order number. The user is still not happy so he hit search again – now we match it with the phone number of the customer – user happy. In this example we could have chosen to create a detailed search interface with 3 text boxes – one for product code, one for order number and one for customer phone number – just as valid. Or we could do a union expression as above – just as valid. Choose the strategy that sits best with your users. But I urge to you to test the simple interface with a single or only few input boxes - it is user friendly.

Databases use SQL

The search expressions for ocl criterias are different than other than viewmodel column expressions in one important aspect. They are executed against persistent storage (your database). If your database is an SQL-Server these expressions will be translated to SQL and sent to the database for evaluation and execution. The database is much better and faster to handle huge volumes of data than MDriven. This means that we are now able to filter out specific objects in our model from a nearly unlimited sized database. The multi variable seekers described in this chapter will be the natural starting point for your users work in your system. They search – they find – and then they work. The work part is often rooted in a specific found object where your other viewmodels expand the neighboring information.

Efficient fetch – real case (advanced – skip until you have the need)

ViewModels are good for efficient fetching of data since they declaratively explain what data that will be used. This enables MDriven to scan thru the expressions and fetch data with a lot fewer queries.

If you have a root of Something that has a list of details and the detail in turn fetch even more details you could easily end up with x*y*z queries to the database.

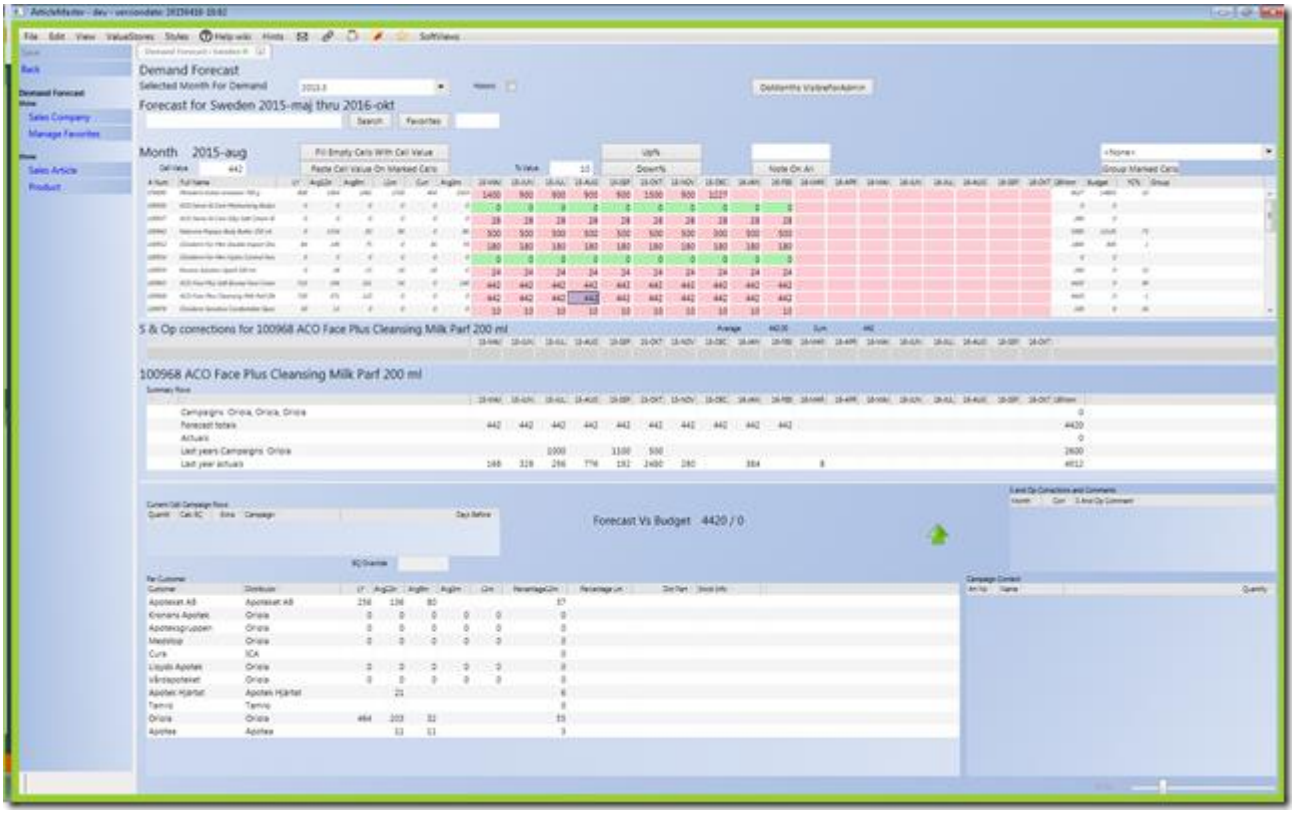
But since the ViewModel use expressions that may be stackable we can rather easily fetch all x, all x.y and all x.y.z with 3 queries.

This is the standard behavior when using the MDriven ViewModels.

When you use seekers the result is collected in a collection variable commonly named vSeekerResult.

Search result is not covered by the standard efficient fetch algorithm because this data is not available at load time. This may in certain situations be less than ideal so we have added strategies to improve the efficient fetching in ViewModels using seekers.

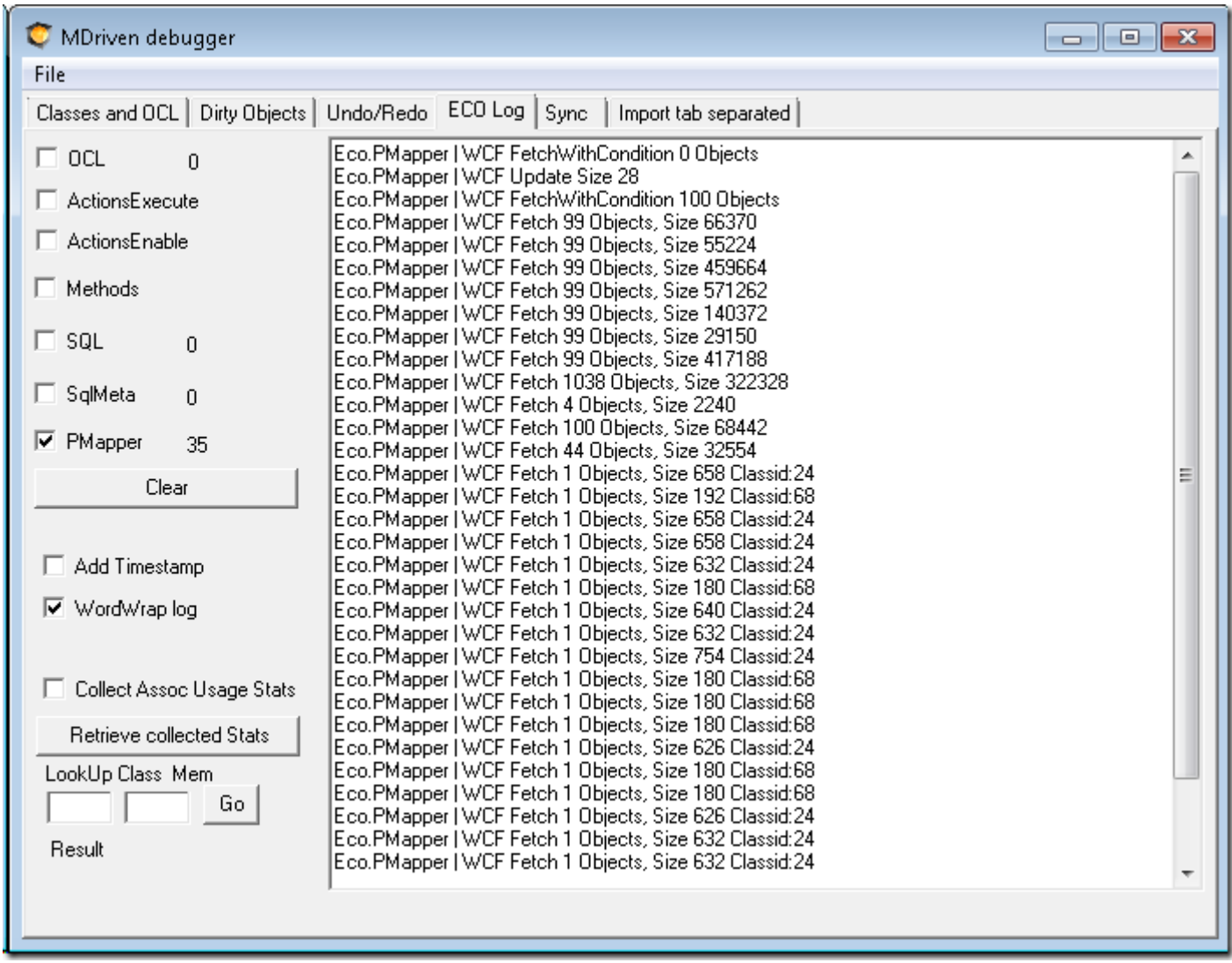
All the ideas implemented and explained below came from a real case where I had a fairly complex UI to visualize and manage demand forecast for articles



There are thousands of objects shown in this one single image – it is a seeker, but it can also get data from a list tied to current user that is called Favorites.

Even with the normal efficient fetch turned on this UI typically spewed out up to 900 questions to the persistence server.

To fully understand where the questions came from I used the runtime/prototype-time debugger:

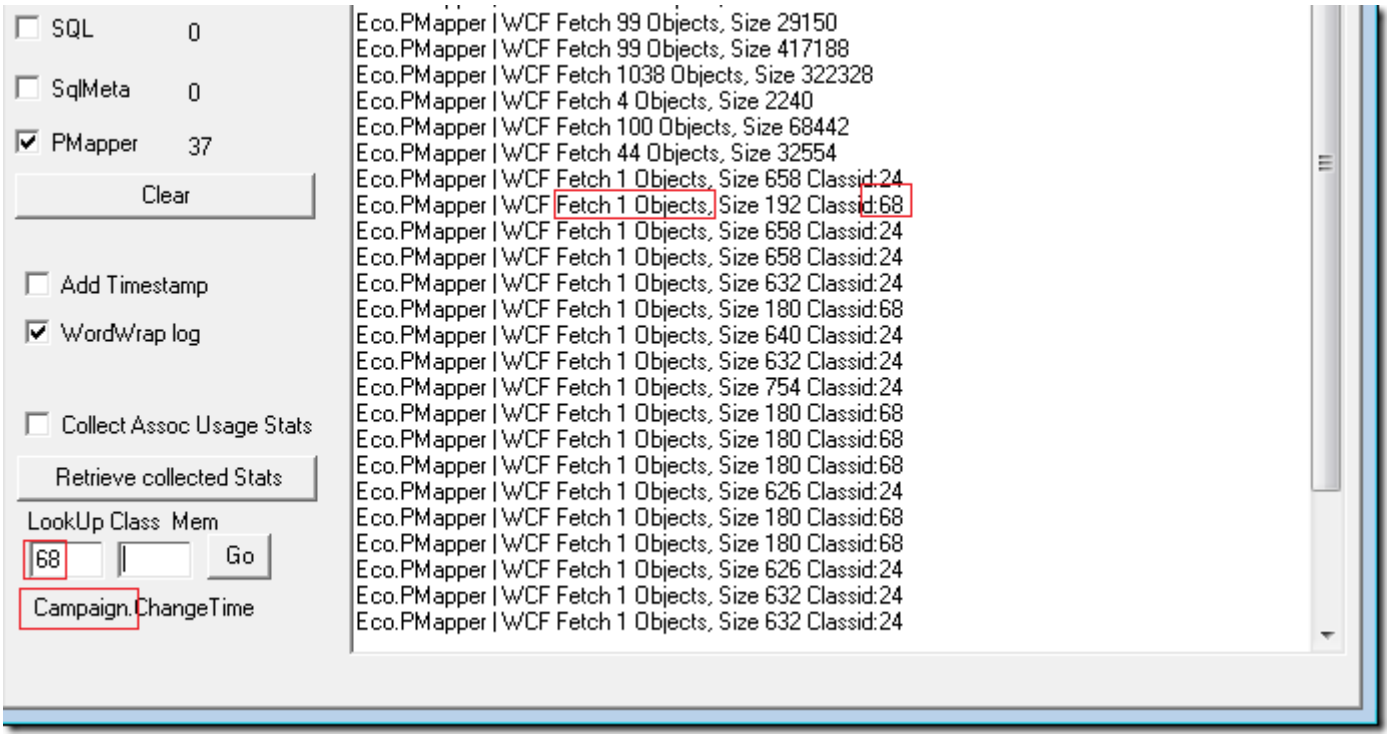


Check the PMapper and you will see all the PMapper calls.

The ones that we look for are the ones that fetch only 1 object – these are candidates for improvement.

When the log sees that only 1 object is fetched it also shows the ClassId.

You can look up classid to name in the debugger:



Once we know what it is that create the queries we can think of ways to fix it.

The new strategy to further improve the efficient fetch is to add a ViewModel Nesting with fetching hints. Any ViewModel nesting with a name that starts with “FetchHints” will be found and the columns within will be executed on any list that seeker logic delivers.

Like this:

```
FetchHints_NestingStartingWithFetchHintsWillExecuteExprOnSearchRes : SalesArticle
  Fetcher1 : self.Article.MonthDemandSupplyCorrections
  Fetcher2 : self.ForecastRowTransient.SalesArticle
  Fetcher3 : self.ForecastRowTransient.TwelveMonths.SalesArticle
  Fetcher4 : self.MonthDemandForCountrys
  Fetcher5 : self.MonthSalesFromDWs
  Fetcher6 : self.Article.MarketedBy
  Fetcher7 : self.CustomerArticles.CampaignRows
```

Working like this I could reduce the load from this UI from 900 queries to 30 something.

Now the problem with search was solved – but when the data came from the other source – the user favorites rather than the search logic – the fetch hints were not applied.

We had to devise a way to allow for me to tell the ViewModel that this logic should execute on a list of objects.

To handle this I make use of the standard variable in ViewModels: selfVM

The selfVM variable is a reference to the ViewModel we work on – self is the object context as before – but selfVM is the ViewModel holding the objects.

The selfVM introduce other operations: ExecuteFetchHints, Save, ExecuteAction.

The ExecuteFetchHints is the one I was after in this case. My code for showing the users favorites in the UI above was extended like this:

```
vSeekerResult->Clear;
Singleton.oclSingleton.User. FavoriteArticles->collect(a|      vSeekerResult.Add(a)       );
selfVM.ExecuteFetchHints(vSeekerResult)
```

And then the same efficient fetching could be applied to the data coming this way as well.