# Validation

Note: This are some thought that proceeded the time the main documentation was written. They are probably not correct anymore.

1. Execute all view model validations that are defined for the current VM.
2. If the validation state has changed raise a validation changed event.

### 

### Collection validation

Validation process for leanPlan:

1. Validate all children of leanPlan. The only child is the Tasks collection. Validate collections like follows:
   1. Create a new validation state holder.
   2. Call validate for each TaskVM (createDal, developSetup) with the state holder.
   3. For the TaskVM do the following:
      1. Validate the children of createDal (none in this case).
      2. Execute the property validations of createDal:
         1. Call PropertyValidating of the IPropertyBehaviorContext. All VMs up the hierarchy have the opportunity to handle the validation event and add errors.
         2. If the validation state has changed (valid/invalid, collection of errors), raise a notification.
      3. Execute the view model validations of createDal.
         1. Call ViewModelValidating of the IViewModelBehaviorContext. All the VMs up the hierarchy have the opportunity to handle the validation event and add errors.
         2. If the validation state has changed, raise a notification.
      4. If the previous steps raised a validation state changed notification, all ancestor VMs handle it. Do the following in their event handler:
         1. Start a view model validation process.
         2. Execute all view model validations for the VM.
         3. If the validation state has changed, raise a notification (which again bubbles up).
2. Execute the property validations of leanPlan.
3. Execute the view model validations of leanPlan.

#### Using the custom state to optimize collection validators for full validations of collections

Some collection validators may need to consider all collection items to determine if a single item is valid. A prominent example is the unique validation. Consider the following steps:

1. Validate the Tasks collection:
   1. Execute the unique validation of Tasks for each TaskVM:
      1. Search all items of Tasks if there is another task that has the same name as the current but that is not the current task. If there is such a task, add a validation error.

But for large collections this may result in a bad performance because for every item in the list we have to iterate over all other items, which results in n² operations.

To optimize this, a validator may get or set a custom state object. The state object set by a validator is passed to every validate operation that is called for the remaining items of collection. With this technique we implement an optimized version of the unique validation:

1. Validate the task collection:
   1. Execute the unique validation of each TaskVM:
      1. If the custom state object is null (this is case for the first item), create