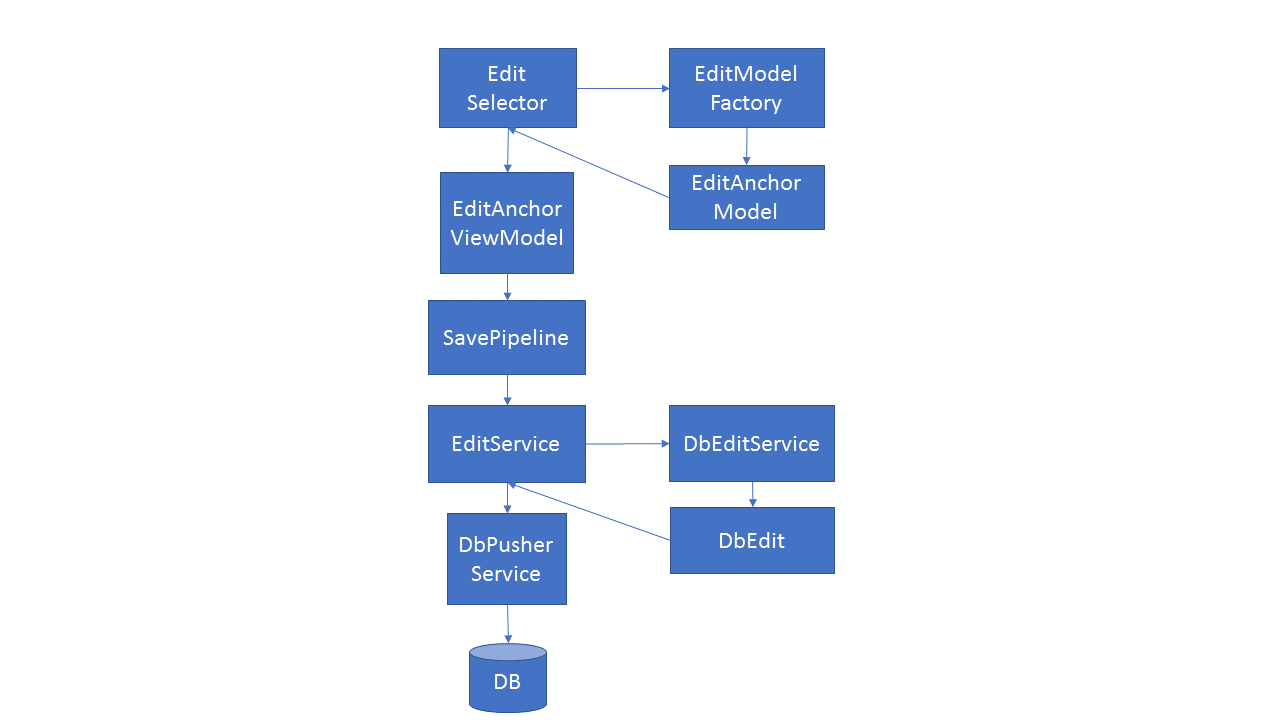
Edit in SesammTool2

# Edit instance



# EditSelector

This is the entry point for the Presentation Layer when performing edits.  
The class handles creating the required models and displaying the edit dialog.

## EditModelFactory

This class handles the creation of the EditAnchorModel to be used in the editing.  
It takes an instance or an entity type and returns the correct model based on the input.

# EditAnchorModel

The EditAnchorModel holds all the data and which properties that are able to be changed by the user. It also handles the creation of anchors/versions that is needed to do the editing and handles validation and null-propagation.

# EditAnchorViewModel

The view model used by the “Edit Instance dialog”. The view model takes an EditAnchorModel as a constructor parameter.

When the user clicks Ok in the dialog the “AcceptCommand” is fired and the view model takes the anchor from the EditAnchorModel and sends it to the SavePipeline.

# SavePipeline

SavePipeline forwards the edited anchors/versions to the Edit Service and adds the possibility to handle PreSave and PostSave functions.

# EditService

The Edit Service determines which objects has changed (Anchor or version) and if they are new or if it’s an update of an existing object.

It then retrieves the correct “EditAction” from DbEditService and sends the changes to DbPusherService as DbEdit objects to be saved in the database.

After the save is completed the DataModel is updated.

# DbEditService

Handles the different EditActions that exists such as “Add”, “Edit”, “Commit” etc.  
These actions are represented as delegates.

The service returns a DbEdit object that holds the Action and the Entity to perform it on.  
Right now they are executed by DbPusherService.

# DbEdit

Holds the delegate with the action to be performed and the entity on which to perform it.

# DbChangeList

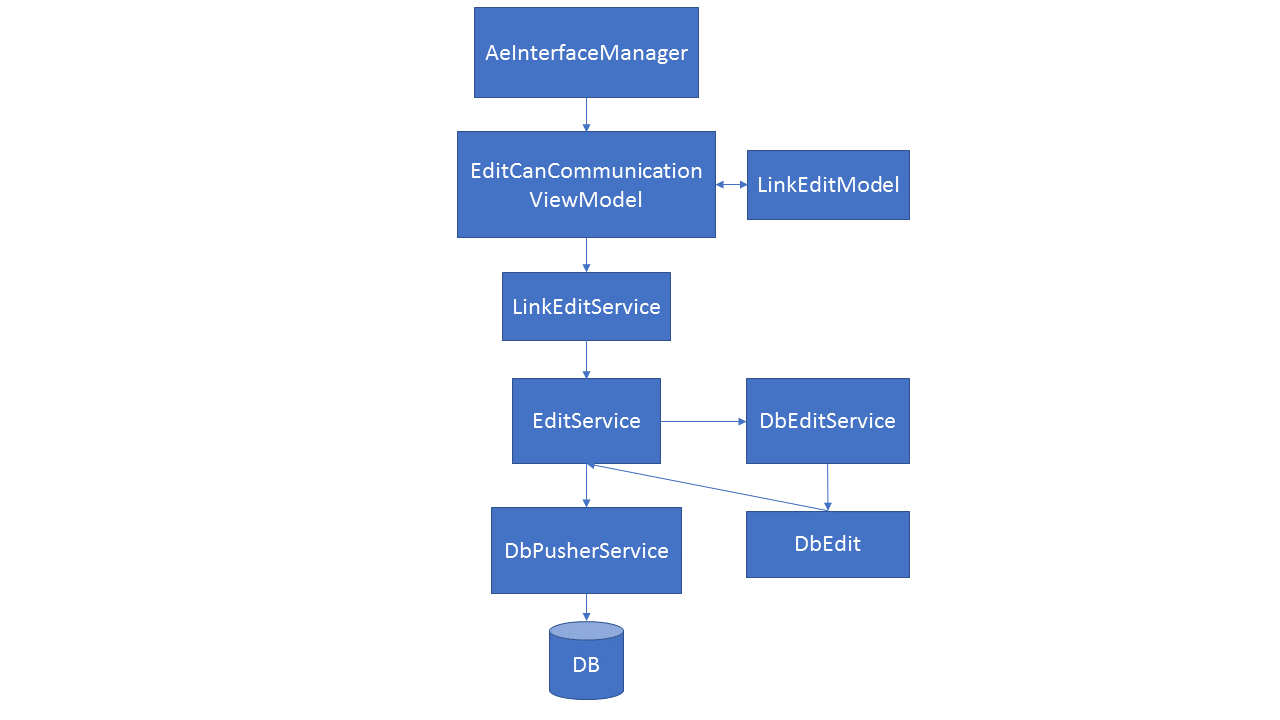
A list of DbEdits that can be sent to the DbPusherService to be executed.

# DbPusherService

Handles the DbEdits and executes the action assigned to them.  
Also sets the correct state of new objects so that they have an initial version (CR0).

The service also saves the objects to the database by calling context.SaveChangesAsync.

# Edit Can Link



# AeInterfaceManager

This is the entry point for the Presentation Layer when performing edits of links and gateways.  
The class creates the models required and displays the dialogs.

# EditCanCommunicationViewModel

The main view model for the “Edit Can Communication” dialog. The view model holds the data, handles the filtering of items, validation and conversion to edit models.

When the user presses for example “Create Tx-pair” the view model converts the data to a CanPortLinkEditModel that is then passed to the LinkEditService to be processed.

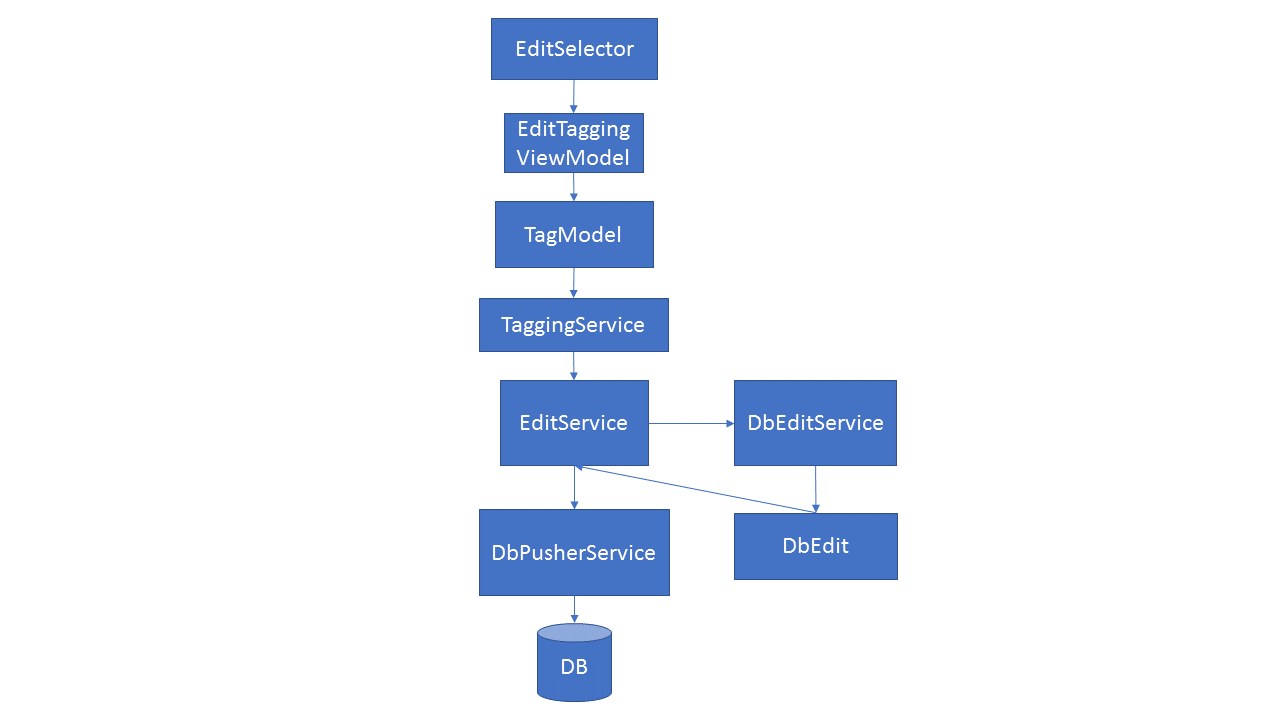
# LinkEditService

LinkEditService converts the CanPortLinkEditModel to EF-objects.

The service also sets any related user functions and creates gateways based on the path that was supplied in the edit models.

The service then forwards the EF-objects to the Edit Service to be saved. This flow uses a different method in EditService because we are sending in EF-objects but the method should do the same thing as the other flows.

# Tagging



# EditSelector

Same as “Edit Instance”, entry point for the presentation layer and handles the creation of view models and showing the dialog.

# EditTaggingViewModel

Handles the tag state and which instances are possible to tag.  
When the Ok-button is pressed in the dialog the view models “AcceptCommand” is fired which will convert the view model to a TagModel and send it to the TaggingService to be processed.

# TagModel

Holds the tags to be saved and which CR they are connected to.

# TaggingService

Takes a tag and handles the change (Inactivate/Activate/Create).  
The service then forwards the anchors to the EditService to be saved.

The rest of the flow is the same as the “Edit Instance”-flow.

# TODO list after talking with Elias with suggested prioritization order

1. Enable and fix integration tests in EditServiceTests before other refactorings. Crash in runtime
2. DbContextProvider::CreateContext() should return SesammToolContext. Done
3. Create a base class for the integration tests that creates a fresh database via Entity Framework (DropCreateDatabaseAlways).
4. Remove the class SavePipeline (see comments above) Done
5. Merge the two AddOrUpdate functions in EditService (see comments above) Done
6. Move functionality in DBPusherService inte EditService Done
7. Merge the two Tag-functions in TaggingService Done
8. Remove SetUserFunctionRelation in LinkEditService.
9. Remove DbEditService and collaborating classes and replace the “Action”-logic with something easier to follow and debug (see comments above).
10. Look at changelist 2267387 in Perforce to see TODO’s added by Elias.
11. Add end-to-end (E2E) tests from EditSelector and down to database.
12. Rename EditAnchorViewModel to EditDialogViewModel.
13. Merge EditAnchorModel2.cs (non-template version) and EditAnchorModel.cs (template version.
14. Rename EditAnchorModel to EditAnchorViewModel and inherit from EditDialogViewModel
15. Move business-logic from EditDialogViewModel to EditService (see appendix A).
16. Refactor DatabaseChangeModel according to figure 1.

# Appendix A – från Elias möteskallelse 2018-10-26

**Problem med nuvarande lösning:**

**Editering av instanser:**

* Svårt att överblicka, vad sker vars?
* Ganska oflexibelt i dagsläget.
* Anchor används som input/modell men vi har knappt någon stödfunktionalitet för dessa objekt.
* Edit metoderna är lambdas som exekveras långt ifrån där de deklareras vilket gör det svårt att debugga när/om det blir fel.
* EditAnchorViewModel innehåller mycket businesslogik som kanske borde vara mer intern till editeringen.

**Editering av länkar:**

Dåliga modeller i dagsläget. För att skapa en länk behövs 1-2x AePortar, 1-\* CanPortData, x UFRel, x Gateways.

Blir rätt många objekt att hålla ordning på och se till att dom är i synk. Föreslagsvis så skapar vi bättre abstraktioner för detta (Finns en idag som heter CanLink men den används ej).

Borde i så fall skapa/använda abstraktioner för:

* Can länk
* Internal Länk
* Direct Wire Länk
* Gateway path

Idag har editering av länkar egna servicar vilket mest beror på avsaknad av abstraktionerna som vi pratat om tidigare.

Dessa borde alignas mer med resterande delar av editeringen. Här används även EfAnchors för att vi ska spara relationer,

detta behövs inte längre då Anchor kan hantera detta men om vi ska bli av med Anchor så måste vi hitta på något annat.

Vi borde ha något snyggare sätt att bygga upp en Gateway path på, typ en builder eller liknande.

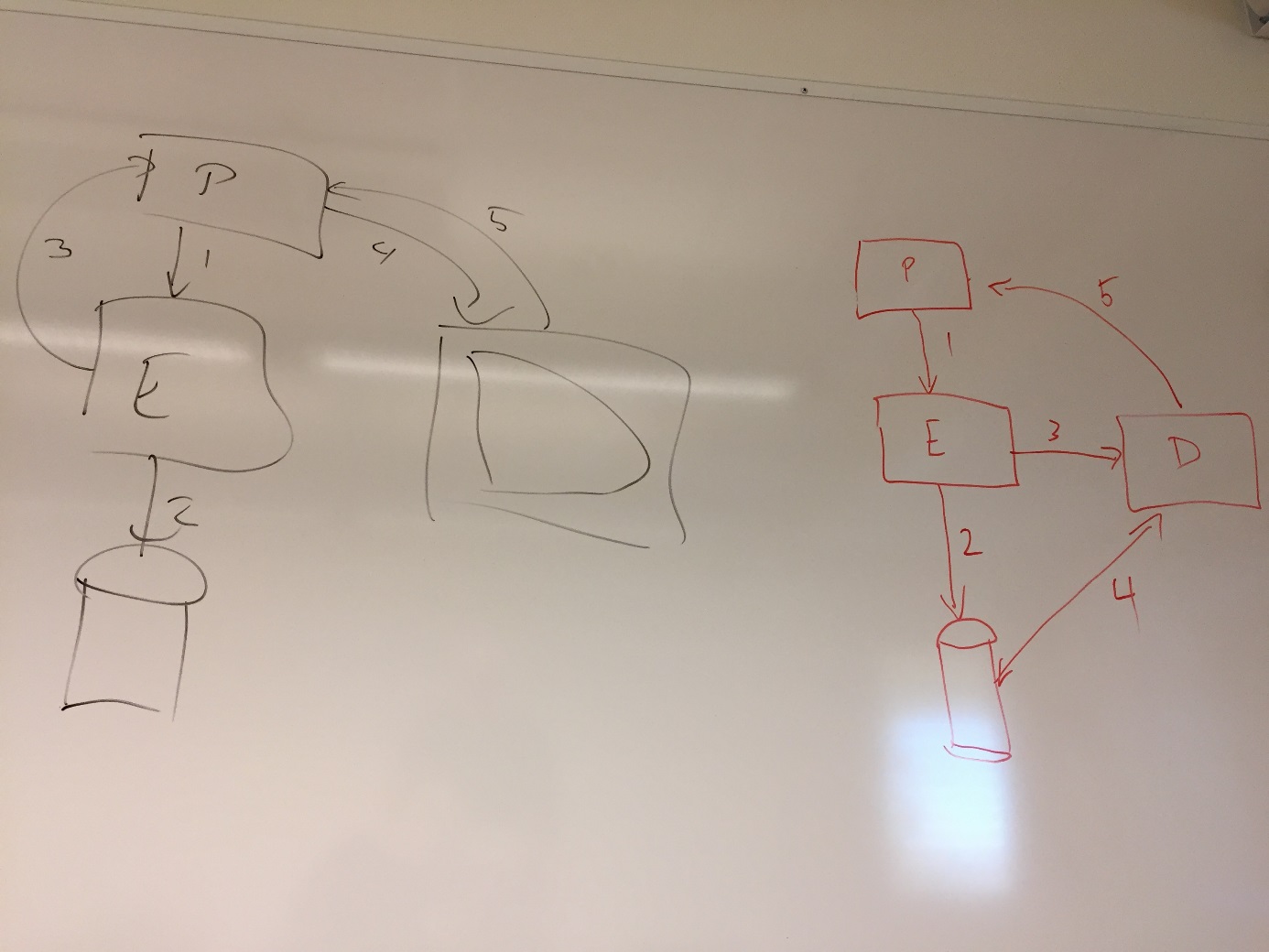
Saknar ett sätt att enkelt se om ett objekt finns sedan tidigare. Vi har ingen definierad Query kapabilitet utan använder Instansmodellen för att göra "Queries", vilket gör att all sådan logik dupliceras och sprids genom kodbasen.

**Generellt:**

* Tror inte på att hantera versioner i editeringen på presentationssidan. Det känns som att vi läcker intern komplexitet ut mot denna sida

som egentligen inte behöver vara där. Versionshanteringen kan skötas av backend/businesslagret och sen har man en mer simplifierad modell för editeringen i frontend.

* Behöver ta höjd för det faktum att man kan göra editeringar utan CR.
* Behöver kolla på hur vi ska göra med delete och resolve eftersom dom är lite special.
* Behöver en bättre hantering av refresh efter en sparning. Den som är i dagsläget är ganska dålig och kan leda till att felaktig data presenteras tills omstart av applikationen.
* Sparning av objekt borde ske i DataLayer, inte i EditLayer.
* Borde eventuellt ha en egen hantering för DeepCopy sparning så vi kan göra vissa optimeringar och sätta CR, IsActive etc korrekt. Den har inte exakt samma behov som editering.



**Fig 1** DatabaseChangeModel

**The left part in fig 1 describes the current implementation.**

There is a hook event defined in MainWindow. PresentationLayer pushes changes to DB (2) through EditLayer (1). EditLayer sends the event (3) to Presentationlayer that updates DataModelManager (4) which notifies PresentationLayer (5).

**The right part in fig 1 describes the proposed implementation.**

PresentationLayer pushes changes to DB (2) through EditLayer (1). EditLayer notifies DataModelManagar that the DB is updated (3). DataModelManager fetches data from DB (4) and notifies the PresentationLayer (5).