Author: J.F.Benckhuijsen (jfbenck@users.sourceforge.net)
Version: 1.0

Kiss Models Library

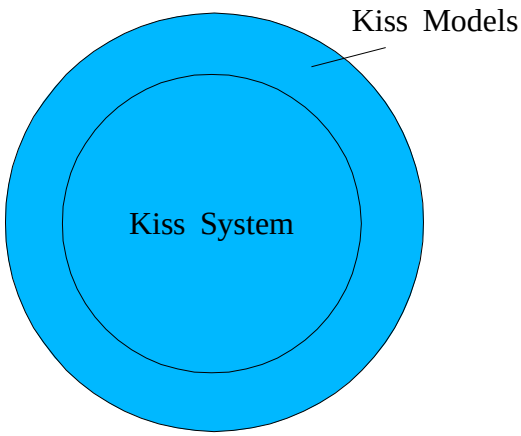
1. Introduction

The whole MDK Application is centered around the Kiss Modeling Language. This means the application uses models to design software. In MDK we designed the Kiss System library to store information about a system. The Kiss Models library is a library of classes, which purpose is to translate this information from the Kiss System to one of the models of Kiss and show it to the users. It also translates the actions of the user to the changes which have to be made to the Kiss System.

This design allows us to alter or enchance the information stored in the system, without the user knowing it. The Kiss Models Library merly provides a view on a (small) part of the system as a whole.

2. Overall design issues

The kiss models library is thus nothing more than some shells around the Kiss System. This implies some limitations to the way the models are implemented.



2.1 Data storage

First of all, all information should be stored in the Kiss System, not in any of the Kiss Models. The responsibility of the Kiss System is to store the information of a system. This is not the responsibility of the Kiss Models.

In some early prototypes of MDK, all information was stored in the models themselves. This resulted in a inconsistent models, hard to interpret data. Also it was difficult to keep the models in sync. Every one of them had to know all the others, so adding a new model meant all models had to be reimplemented.

In the new design, models aren't allowed to store any information. In implementation terms this means there can't be any private members, except for a pointer to the used Kiss System, which is provided by the base class TAbstractModel.

Some models can be implemented without any problems using these constraints. The Information Quadrant for example has a near 1-to-1 equivalent in the Kiss System. Other models however, like the Sequence Rules model and the Object Interaction Model don't have such a direct 1-to1 implementation.

Implementing these models without using any private members proved to be highly inefficient or even impossible. For these models it's allowed to store a cache, a translated version of the Kiss System, which is used for easier implementation.

However usage of these caches do have some consequences. First of all, all changes made to the model will result in changes to these cache. However, to allow consistency between models, these changes have to be stored in the Kiss System also. Besides that, it's possible for the System to change, without the model knowing. So the model has to be aware of the changes made to the System, and translate these changes to changes in its cache.

2.2 Communications with the Kiss System and the Kiss Model Components

As said, there has to be some communications with the Kiss System to store the changes to the model in the cache (if the model has one). There also has to be some communications with the Kiss System to notify the User Interface of external changes to the system (for example changes created by another model).

However we don't want to let the Kiss System know there are models. Informing the System of the existence of models would imply we had to change the System every time a new model is added. We'll have to do that probably anyway, because a new model stores some new information (in most cases anyway). However we'll only want to add some features to the System, we don't want to search through all the code to implement all the changes the new model has to be notified of. Besides that, it isn't even the responsibility of the System to notify the models.

The sollution to implement these communications is the usage of the Observer Pattern (see Design Patterns by E. Gamma). Whenever the user changes a model using the User Interface, a method of the appropiate model is called. Any method of a Model which changes the System, translates this call to call to change the System. The system handles these changes and does some checking to see wheter these changes can be executed (see the description of the Kiss Rules Library). When these changes are processed, the System , which is the Subject of the Observer pattern, notifies it's Observers (or Listeners) of these changes. In this callback method the model applies these changes to it's cache (if it has one). After that, it notifies the User Interface of the changes. The User Interface then shows these changes.

Be aware that the real changes in the cache of the models and the changes to the User Interface are made in the callback methods. They are not processed in the initial call to the model. There are two reasons, why we implement the models this way:

First of all, the change excecuted by the user are not real changes, these are just change requests. This means, the change request could fail, which results in an error message being shown to the user. Using the callback methods we seperate the change requests and it's error processing from the actual changes to the User Interface or the cache. Off course, this results in easier code to maintain.

Second, changes can be made by some other source than the model itself (for example another model). This means, the changes have to be made in the callback methods anyway, because we have to update the model whenever the system has changed. Implementing the change logic in both the request methods and the callback methods is off course nonsense, because the callback methods are called anyway.