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

Version: 1.0

Kiss Consistency Checks

1. Problem description

What are the kiss consistency checks? In every program the application programmer has to implement two types of preconditions for each piece of code. There are technical preconditions and logical preconditions. An example of a technical precondition is a check wheter an array bound is crossed. Technical preconditions are very implementation specific. An example of a logicla precondition is a check wheter an account on a bankprogram isn't locked before allowing a withdraw. These preconditions are more problem specific. In MDK we also had to implement both these types of preconditions.

2. Sollution

In the first version of MDK they were both implemented side by side in the same piece of code. This seem to be the most logical sollution to the problem. Soon however, we realized we could implement the technical preconditions this way, but not all of the logical preconditions we needed, so we had to redesign this part of the application.

The second version of MDK has a different approach. Technical preconditions are still implemented before the actual code. However previously these preconditions generated exceptions that were passed on to the user. In our opinion however a user doesn't know how to handle a technical error. He doesn't know anything about array bounds. In fact, a crossing of array bounds (or any other technical precondition for that matter) is a programming error, and should be acted upon as such. Because of this, we replaced all technical exceptions with assertions. So whenever a technical precondition fails, the program is simply aborted. This seem rather radical, however technical preconditions that fail are simply bugs that can't be recoverd from.

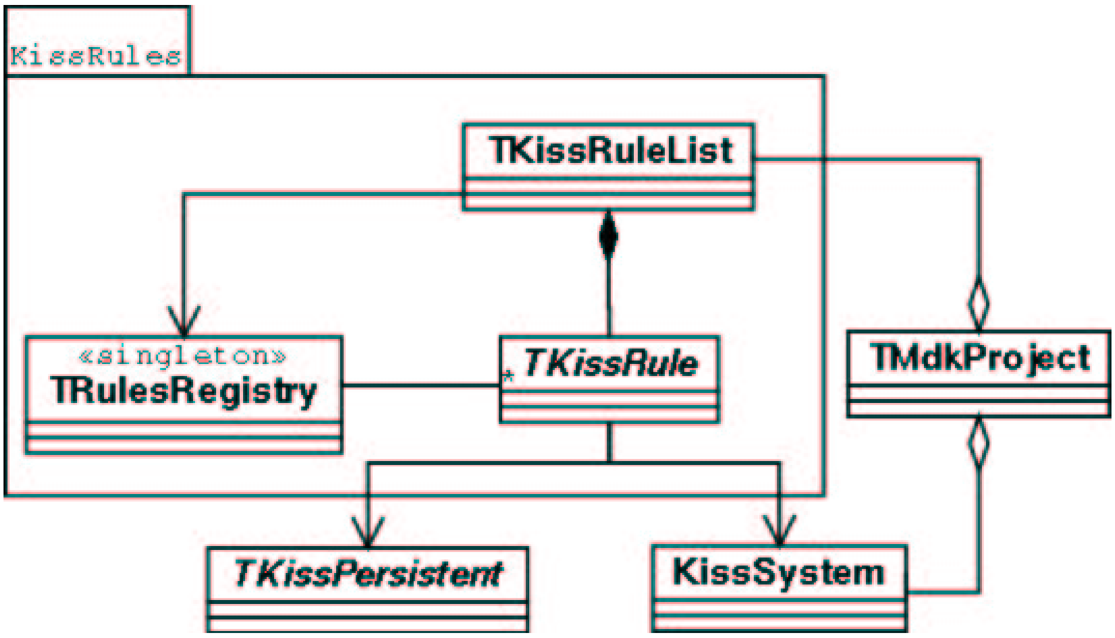
With the technical preconditions solved we still need a more flexibel, extendible way to implement the logical preconditions. It's clear logical preconditions have to be implemented with exceptions, because the user can take action to correct the problem. Aborting the program leaves him with very little options to do so...

In the new version of MDK this problem was solved with a new library: the KissRules library (namespace: “KissRules”). This library stores all the logical rules of the Kiss Modeling language. The UML diagram below shows the design of this library.

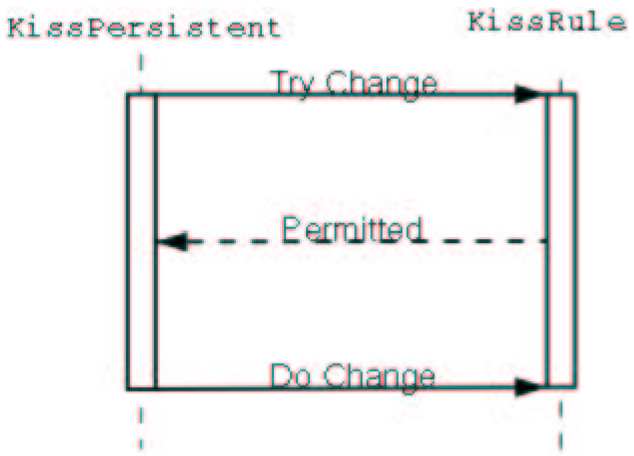
The most important class in the whole library is the TKissRule class. This is the abstract base class of all rule classes in the library. The attach method of the KissRule class creates a new KissRule and attaches this rule to the KissSystem passed as a parameter and will thus check this system.

The TRulesRegistry is a singleton class to which rules have to register themselves. This is done by declaring a static variable in each cpp file of a rule. The constructor of the base TKissRule checks if this rule was already registerd with the registry and if not does so. This way, we can add new rules to the system, without having to change any of the other classes.

The TKissRuleList acts as a container class for all rules checking a certain KissSystem. Upon creation the KissRuleList asks the rules registry for all available rules, attaches each of them to the system and stores the newly created rule in an internal list. The KissRuleList manages the whole lifetime of the attached KissRules.



Each implemented kiss rules is an Observer (Design Patterns page 294), with the KissSystem and it's subclasses being the subjects. Before changing anything each KissPersistent (or descendant classes) first emits a signal asking wheter a certain change is allowed. This signal is caught by zero or more rules. Each of these rules do their own checking wheter this change may be excecuted. Whenever one fails, the change can't be excecuted and an exception(created by the rule) is thrown. The remaining rules after the failing one will not be checked. If none of the rules fail, the KissPersistent excecutes its change. After that a signal will be emitted to inform everybody the change was succesful. This is needed for example to inform a rule which monitors KissObjects of the fact that a new object was added.



The MdkProject class (which doesn't belong to the KissRules library) manages both the KissSystem itself and it's rules using the KissRuleList. This way the whole logical checks are insulated from the rest of the program.

This way of implementing the logical checks (that is, seperated from the actual code they check) may seem illogical. However this way, we've created a flexibel way to implement these checks: we can easily add or modify the rules a system has to obey to. Esspecially the flexibel adding is important because it gives us room to partially implement the rules for the time being and implement the remaining ones later.

Besides that, there are some rules which can't be checked along with the actual code. An example of this is the setName method of a KissObject. A name has to be unique in the system, so before setting the name of an object, it must be checked if another object with that name already exists. An individual object however doesn't have any knowledge of the rest of the system. So this rule has to be implemented in an external class. Using the old way, this resulted in the logical checks being implemented in two places. This way, it's harder to check if all rules are implemented, and it's harder to debug.

There is a problem however in this approach. The Kiss Rules should enforce consistency on the models. However a model the user is working on probably isn't consistent until the user is finished working on the model (or even after multiple edit sessions). So we can't enforce models to be completely consistent all the time.

Instead we enforce this consistency only when it's needed. In the MDK application full consistency is only needed when the user wants to generate a the application from the designed model. For this reason the Kiss Rules library is divided into two parts. The first part is the already described set of classes inherited from TKissRule. These checks should always pass. The second part is a set of classes inherited from TConsistencyCheck. These checks are only applied whenever the user wants to generate the application (or whenever full consistency is requiered). If one of these checks fails, the user is informed about this fact, generation is aborted and the user should correct the problem.