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

Implementation Guideliness

In every project it's important to have some general implementation guideliness. These guideliness provide rules which each developer should obey. This provides a consistent view on the code, and thus makes it easier form different developers to work on the same project. This document describes the guideliness as used by the MDK Development team. Future developers of this project should follow these guideliness to ensure the consistency of the implementation.

1. Naming Conventions

The project is designed in an object-oriented manner. This means we have a lot of classes. Each of these classes is coded into a separate module. Thus we only have a single class in a .cpp and .h file. An exception to this rules is the case were a class is needed for implementation of the main class only. In this case it's allowed to code the whole class (definition and declaration) in the .cpp file.

A class name always starts with a T. For classnames consiting out of multiple words these words are concatenated and each first letter of a new word is capitelized. In this we follow the Borland coding convention. An example is: class TGeneratorContext {}.

Names of module files are the same as the class their implementing. So the files implementing the TGeneratorContext would be called TGeneratorContext.cpp and TGeneratorContext.h.

2. Layout & Comments

Using the same layout throughout a project increases the readability of the code. Using comments clarifies strange looking code. This project only has a few rules concerning the layout of the files:

- Indent your code using tabstops of 8 spaces;
- Use comment to clarify code which is really hard to understand. Generally it's not necessairy the add comments to your code. Good code is self explaining, so only when you have to write code which is hard to understand, for example because of some strange optimalisations, adding comments to the code is justified;

3. Portability

Portability is an important issue in a time where new operating systems arise. Thus we want our code to be as portable as possible. Off course, this can't be achieved for the whole program without spending a great deal of time on this aspect or by making some strange design choices. Because of this we've decided to make a part of the application portable.

The MDK Application is a very functional oriented application, meaning the User Interface isn't really that great. Most effort is spend in designing and implementing the low layers of the application.

Thus, it's a logical choice to make this part portable, and fortunally this is the easiest to do. All implementations, which have anything to do with functionality have to be written portable. This means, only standard c++ (home written classes) and the usage of the standard c++ library is permitted in these layers.

Because we're developing in Borland C++ Builder, the most non-portable part is the VCL,

we don't want to have to rewrite that. So when editing functional modules it's not allowed to include any of the VCL header files.

Besides that, certain parts of the program have been made easy to port. The best example is the Modelviewer component, which is largly prepared to be ported to any other platform. This is because the Modelviewer is such a complex component. Rewriting it for another platform would cost far more time than making it easy to port.

4. Insulation

Insulation is only used where it makes sense. We don't want to insulate the whole application, this would mean multiplying the development time by three or four. We do however won't some insulation to prevent design changes from cascading through the layers.

In practice we've insulated the whole Kiss System library using the Kiss Models library. Also we've insulated the