NANDRAD Developers Manual

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 1.0.0, July 2020

Table of Contents

1. Introduction

This document is for NANDRAD and SIM-VICUS developers. It discusses coding guidelines/rules and covers the underlying concepts of the solver and user interface implementation.

2. General Information

2.1. Building the code

Building the code is fairly easy and only requires two steps:

1. setup the development environment
2. run the build scripts

For actual development we recommend use of Qt Creator with the prepared qmake build system.

2.1.1. Setting up the build environment

Generally, you need a fairly up-to-date C/C++ compiler (that means: `c++11` features should be supported). Also, you need the Qt libraries. And you need cmake. That's it, nothing else!

Windows

Note: the build scripts are set up to work for Visual Studio 2015 x64 with Qt Version 5.11.3. You can, however, use any other combination of Visual Studio or even Intel compiler or MinGW and any other (newer) Qt version. It may be, though, that you have to tweak the build scripts or defined environment variables.

> Team development works best if all use the same compilers and library versions. If you change build scripts and commit them to the repository, always make sure that you do not distrupt other peoples work by requiring alternative installation directories/tool versions.

Here are the recommended steps:

1. Download and install Visual Studio 2015 x64 Express for Desktop. You can get it on https://visualstudio.microsoft.com/de/vs/older-downloads/
2. Download and install cmake. Either download from https://cmake.org or use *chocolatey* (https://chocolatey.org) and run

```
choco install cmake
```

3. Download and install Qt Version 5.11.3. Download the Qt Online Installer (https://www.qt.io/download-qt-installer) and in the installation program select the 5.11.3 variant instead of the default (newest).

Linux

On Linux it's a walk in the park. Just install the build-essential package (g++/clang and cmake and qt5default packages). On Ubuntu simply run:

```
> sudo apt install build-essential cmake qt5-default libqt5svg5-dev
```

MacOS

MacOS comples with Clang installed. I suggest installing *Homebrew* (see https://brew.sh). Then

```
> brew install cmake
```

Parallel gcc OpenMP code require a bit of extra work (to be documented later :-)

Then, download the Qt online installer (https://www.qt.io/download-qt-installer) and also select version 5.11.3.

2.1.2. Building

This works pretty much the same on all platforms. If you've successfully installed the development environment and can build basic Qt examples (open Qt Creator, pick an example, build it), you should be ok.

Go to the `build/cmake` subdirectory and run:

```
> ./build.sh
```

for Linux/MacOS or

```
> build_VC_x64.bat
```

for Windows.

On Linux/MacOS you can pass a few command line options to adjust the build, for example:

```
> ./build.sh 8 release omp
```

to compile in parallel with 8 CPUs and create a release build (optimized, no debug symbols) with OpenMP enabled. See the documentation in the `build.sh` script for more information.

Once the build has completed, the executables are copied into the `bin/release` (or `bin/release_x64` on Windows) directory.

On Windows, you may want to run `bin/release_x64/CreateDeploy_VC14.bat` batch file to fetch all required DLLs (so that you can start the application by double-clicking the executables).

2.1.3. Development with Qt Creator

Development is best done with Qt Creator (it is way more efficient to work with than Visual Studio or Emacs/VI). The source code is split into many different libraries and executables, so you best open the prepared session project file `build/Qt/SIM-VICUS.pro`.

If you start working with Qt Creator, please mind the configuration rules described in Qt Creator Configuration.

2.2. Coding Guidelines and Rules

Why coding guidelines and rules?

Well, even if you write software just for yourself, once the code base reaches a certain limit, you will appriciate well written code, so that you:

1. avoid wasting time looking for variables, types, functions etc.
2. avoid re-writing similar functionality, simply because you don't rememeber/find the already existing code pieces
3. avoid accidentilly breaking code because existing code is hard to read

Bottom line:

Write clean and easy to read and maintain code, both for yourself and others in the team.

The big question is: What is clean and easy to read/maintain?

Well, in my humble opinion this is mostly achieved by

- doing stuff (mostly) the same way as everywhere in the code
- using conventions that makes it easy to exchange code with others
- using conventions that save you the trouble of remembering each and every variable/function etc.
- write code that makes it easy for your development environment to assist **you** (see also section Qt Creator below)

Below I've collected a bunch of rules/guidelines that help us achieve this goal of getting nice code, without restricting the individual coding style of each developer too much.

2.2.1. Programming efficiency

We are a small team and need to get the most out of our programming time.

Basic rule: **Know your tools and choose the right tool for the job!**

For programmers, you need a good text edtor (for everything that's not actual code, or for quick hacks) and a decent development environment (IDE). I'd say *Qt Creator* wins big time against Visial Studio, XCode and any other stuff out there, but let's not start an **emacs vs. the world** flame war here :-).

Of course, you also need to handle svn/git, diff and merge tools etc. but text editor and IDE are the most important. I'd suggest *SmartGit* as git client - not because it is the best out there, but because most of the team members use it and can help you better with a problem.

Knowing the capabilities of your particular IDE you can write code such, that already while typing you can use auto-completion to its maximum. This will speed up your coding a lot and save you much unnecessary compilation time. With code-checking-while-typing (see clang checks in Qt Creator), you'll catch already 80% of typical compiler bugs, so we should use this.

Also, some of the naming conventions below help in an IDE to be fast, for example the `m_` prefix in member variable names, really speed up coding. You need to access a member variable: type `m_` and you'll get only the member variables in the auto-completion, no mistake with local variables is possible.

2.2.2. Indentation and line length limit

- only tabs for indentation, shown in display as 4 spaces - especially on larger monitors with higher resolutions this will allow you to see indentation levels easiy; and since we are using tabs, you may still switch your development environment to use 2 or 8 spaces, without interrupting other author's code look
- line length is not strictly limited, but keep it below 120 (good for most screens nowadays)

2.2.3. Character encoding and line endings

- Line endings LR (Unix/Linux) - see also git configuration below
- UTF-8 encoding

2.2.4. File naming and header guards

- File name pattern: `<lib>_<NameInCamelCase>.*`, for example: `IBK_ArgsParser.h` or `NANDRAD_Project.h`
- Header guards: `#ifndef <filenameWithoutExtension>H`, example: `#ifndef NANDRAD_ArgsParserH`

2.2.5. Namespaces

Each library has its own namespace, matching the file prefix. Example: `NANDRAD::Project` get `NANDRAD_Project.h`

> **Never ever** write `import namespace XXX`, not even for namespace `std` !!!
>
> This is mostly a precaution, as in larger projects with many team members it is very likely that function names are similar or even the same, if written by different authors. When typing in your favourite development environment with code completion you are forced to write the namespace and the auto-completion will now only offer those functions/variables that are defined in the respective namespace (making it much harder to mistakely call a function you didn't intend to call).

2.2.6. Class and variable naming

- camel case for variable/type names, example: `thisNiceVariable`
- type/class names start with capital letter, example: `MyClassType` (together with namespace prefix nice for auto-completion of type names)
- member variables start with `m_`, example: `m_myMemberVariableObject`(useful for auto-completion to get only member variables)
- getter/setter functions follow Qt-Pattern:

Example:

```
std::string m_myStringMember;

const std::string & myStringMember() const;
void setMyStringMember(const std::string & str);
```

Never ever write `getXXX` !!!

The reason for having strict rules for these access functions is two-fold:

1. you do not need to remember the actual names for the getter/setter functions or the variable itself - knowing one will give you the name of the others (less stuff to remember)
2. efficiency: you can use the Qt-Creator feature → Refactor→Add getter/setter function when right-clicking on the member variable declaration

2.2.7. Exception handling

Basic rule:

- during initialization, throw `IBK::Exception` objects (and **only** `IBK::Exception` objects in **all code that uses the IBK library**) : reason: cause of exception becomes reasonably clear from the exception message string and context and this makes catch-and-rethrow-code so much easier (see below).
- **during calculation** (in parallel code sections), **avoid throwing Exceptions** (i.e. write code that cannot throw); in error cases (like div by zero), test explicitely for such failure conditions and leave function with error codes

When throwing exceptions:

- use function identifier created with `FUNCID()` macro:

```
void SomeClass::myFunction() {
    FUNCID(SomeClass::myFunction);

    ...
    throw IBK::Exception("Something went wrong", FUNC_ID);
}
```

Do not include function arguments in `FUNCID()`, unless it is important to distinguish between overloaded functions.

When raising exceptions, try to be verbose about the source of the exception, i.e. use `IBK::FormatString`:

```
void SomeClass::myFunction() {
    FUNCID(SomeClass::myFunction);

    ...
    throw IBK::Exception( IBK::FormatString("I got an invalid parameter '%1' in object #%2")
        .arg(paraName).arg(objectIndex), FUNC_ID);
}
```

See documentaition of class `IBK::FormatString` (and existing examples in the code).

Exception hierarchies

To trace the source of an error, keeping an exception trace is imported. When during simulation init you get an exception "Invalid unit """ thrown from `IBK::Unit` somewhere, you'll have a hard time tracing the source (also, when this is reported as error by users and debugging isn't easily possible).

Hence, if you call a function that might throw, wrap it into a try-catch clause and throw on:

```
void SomeClass::myFunction() {
    FUNCID(SomeClass::myFunction);

    try {
        someOtherFunctionThatMightThrow(); // we might get an exception here
    }
    catch (IBK::Exception & ex) {          // we can rely on IBK::Exception here, since nothing else is allowed in our
code

        // rethrow exception, but mind the prepended ex argument!
        throw IBK::Exception(ex, IBK::FormatString("I got an invalid parameter '%1' in object #%2")
            .arg(paraName).arg(objectIndex), FUNC_ID);
    }
}
```

The error message stack will then look like:

```
SomeClass::someOtherFunctionThatMightThrow    [Error]          Something went terribly wrong.
SomeClass::myFunction                         [Error]          I got an invalid parameter 'some parameter' in object
#0815
```

That should narrow it down a bit.

2.2.8. Documentation

Doxygen-style, prefer:

```
/*! Brief description of function.
    Longer multi-line documentation of function.
    \param arg1 The first argument.
    \param temperature A temperature in [C]
*/
void setParams(int arg1, double temperature);

/*! Mean temperature in [K]. */
double m_meanTemperature;
```

Mind to specify **always** physical units for physical value parameters and member variables! Physical variables used for calculation should always be stored in base SI units.

2.2.9. Git Workflow

Since we are a small team, and we want to have close communication of new features/code changes, and also short code-review cycles, we use a single development branch **master** with the following rules:

- CI is set up and ensures that after each push to **origin/master** the entire code builds without errors - so before pushing your changes, make sure the stuff builds
- commit/push early and often, this will avoid getting weird merge conflicts and possibly breaking other peoples code
- when pulling, use **rebase** to get a nice clean commit history (just as with subversion) - makes it easier to track changes and resolve errors arising in a specific commit (see solver regression tests)

- before pulling (potentially conflicting) changes from **origin/master**, commit all your local changes and ideally get rid of temporary files → avoid stashing your files, since applying the stash may also give rise to conflicts and not everyone can handle this nicely
- resolve any conflicts locally in your working directory, and take care not to overwrite other people's code
- use different commits for different features so that later we can distinguish based on commit logs when a certain change was made
- **never ever commit generated binary files** (object code files, executables, binary files in general), as always, there are exceptions to this rule, for example PDFs for documentation etc, but keep in mind that all this stuff stays in the repository (eventually blowing it up to unreasonable sizes… no one wants to download gigabytes of reposity data)

For now, try to avoid (lengthy) feature branches. However, if you plan to do a larger change (which might break compilation for some time to come) and, possibly, work on the master at the same time, feature branches are a good choice.

2.2.10. Tips and tricks

Detecting uninitialized variable access during debugging

Accessing not initialized member variables or **even worse**, accessing member variables initialized with default values (hereby skipping over mandatory initialization steps), can be hard to track during development/debugging.

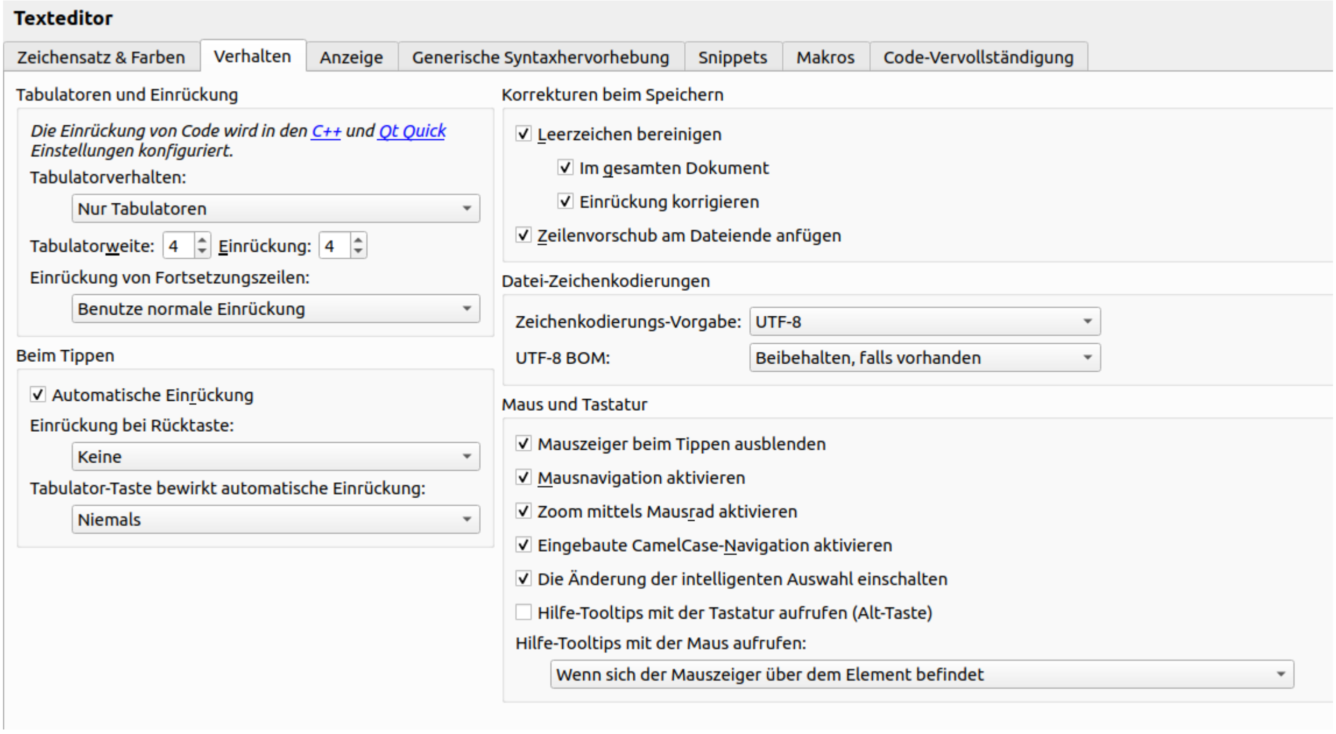
Hence initialize variables that **need to be initialized** with values you will recognized. Using C++11 features, you should write code like:

```
class SomeClass {
    ...

    // nullptr is good to recognize pointers as "not initialized"
    SomeType    *m_ptrToSomeType = nullptr;

    // use some unlikely "magic number" to see that a variable is not initialized (yet)
    double      m_cachedCalculationValue = 999;
};
```

2.3. Qt Creator Configuration

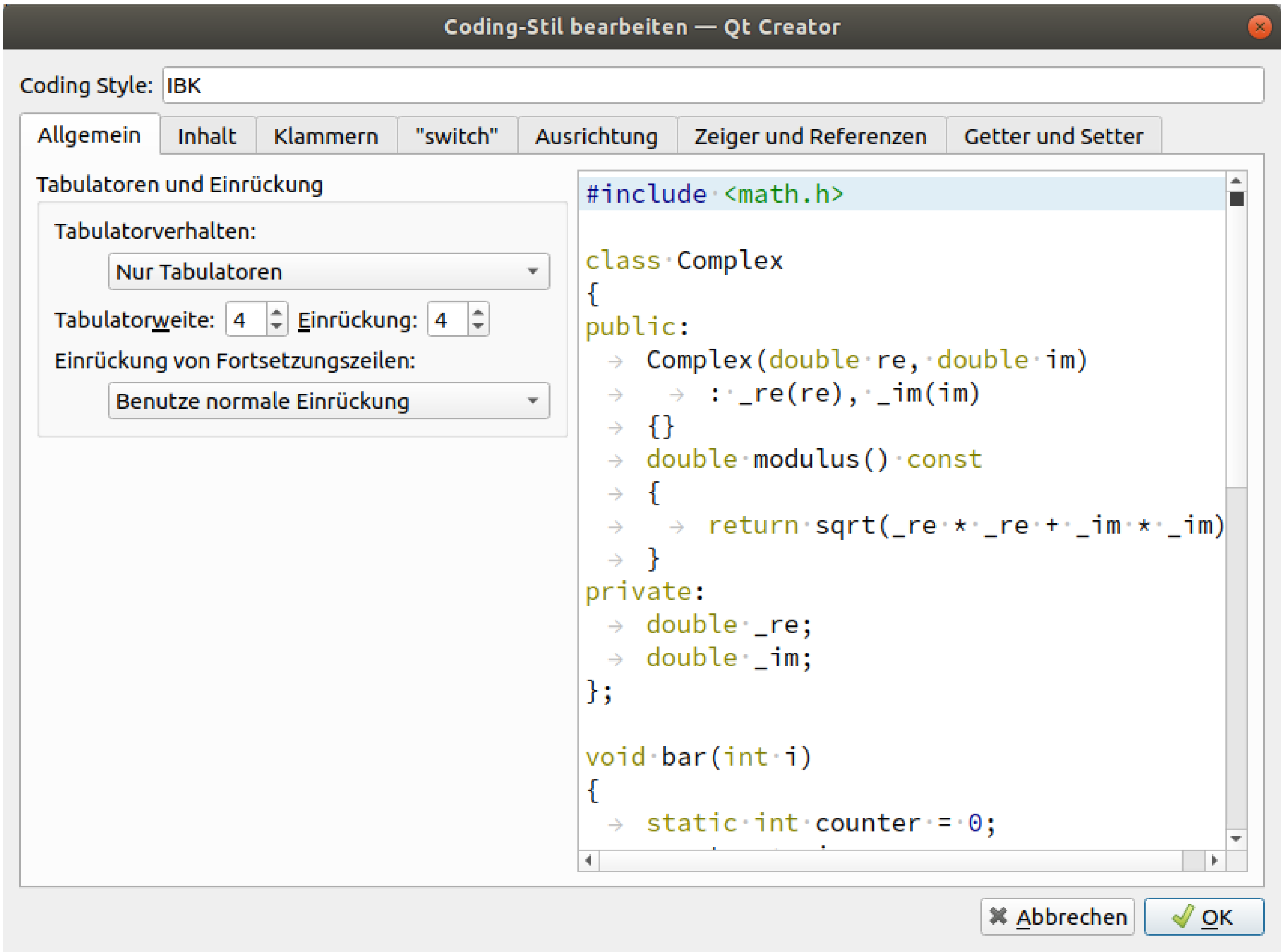
Please use the following Qt Creator text editor and coding style configuration. Some tipps on efficient Qt Creator use are given below.

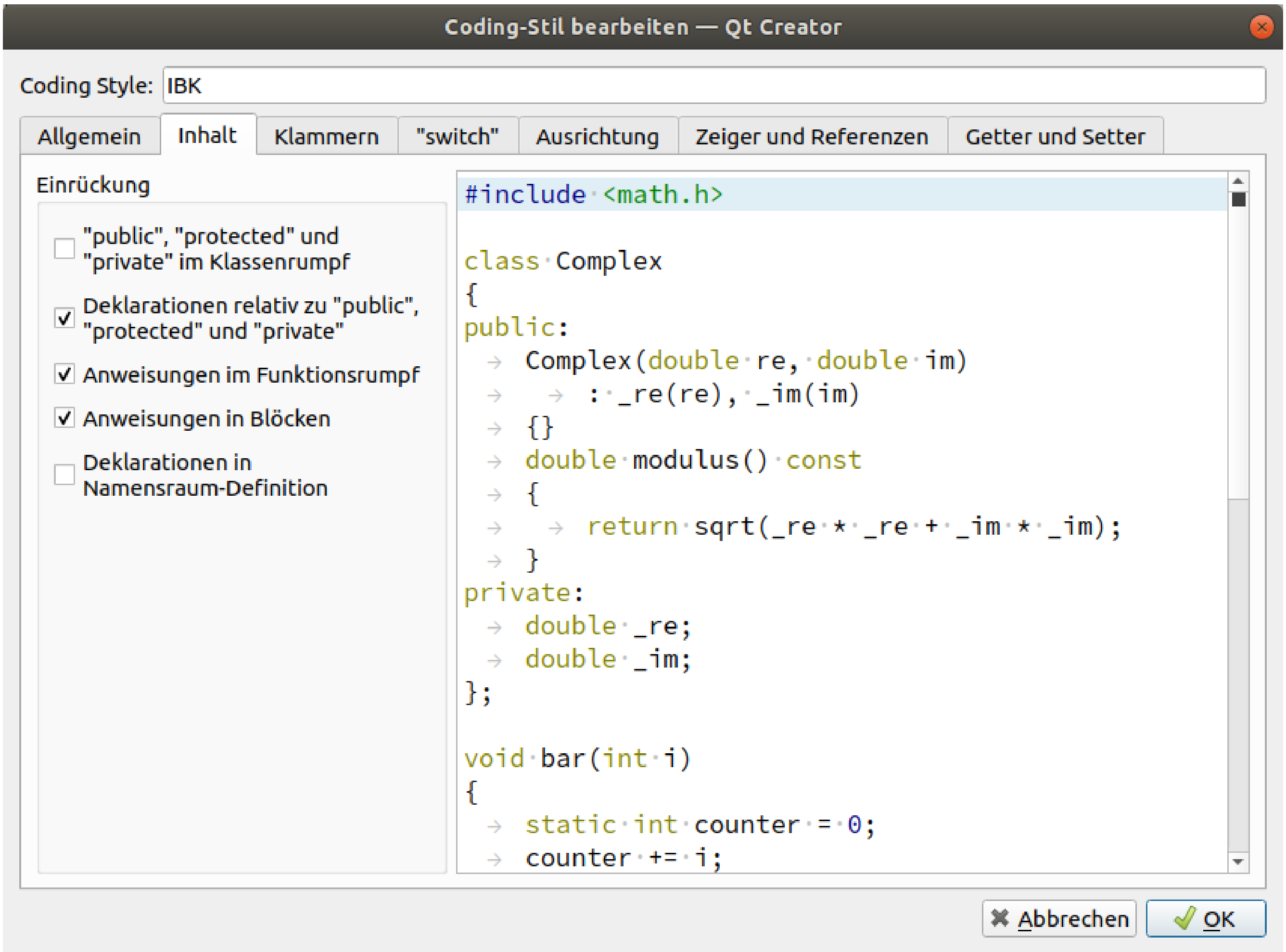
2.3.1. TextEditor settings

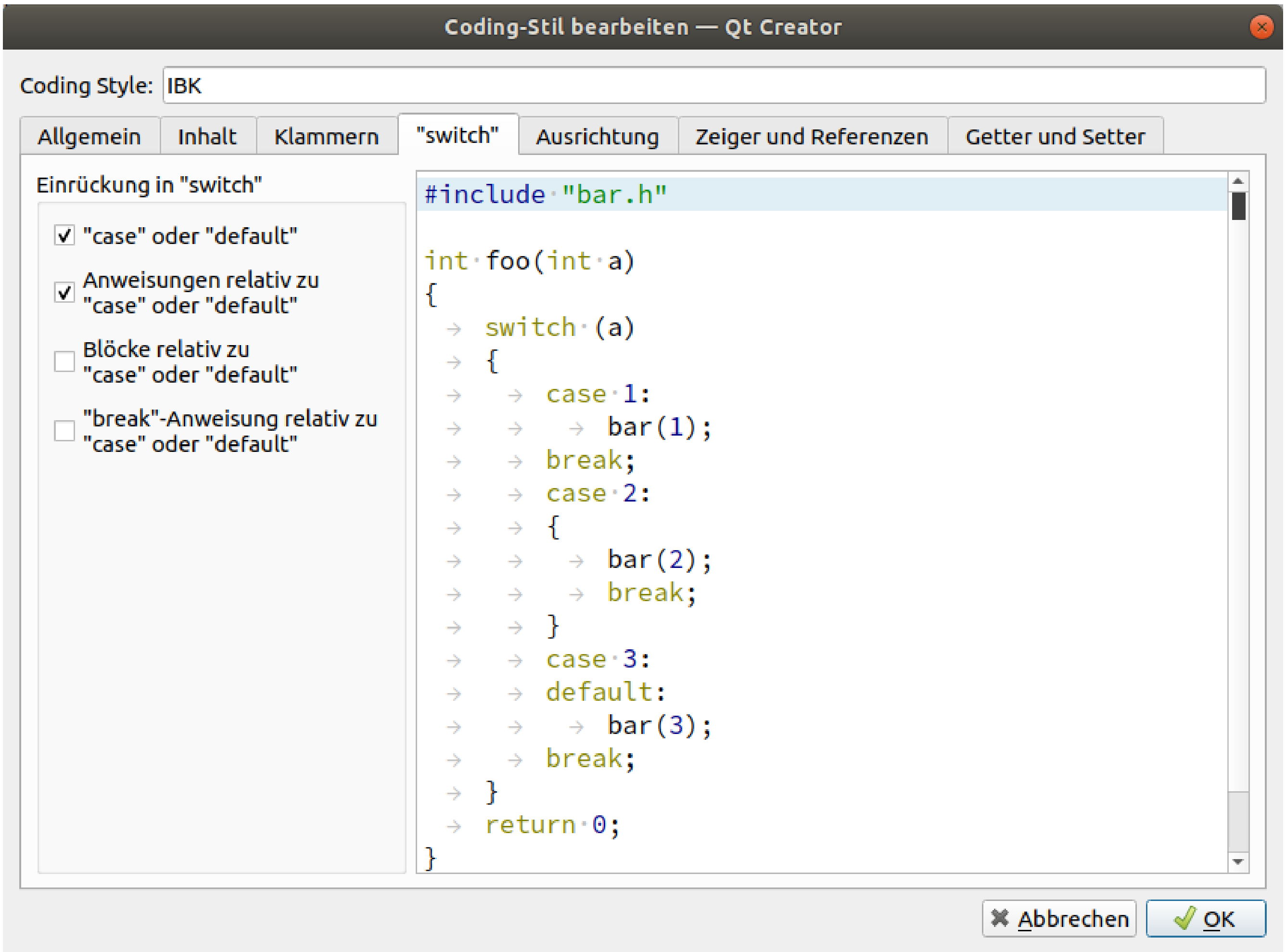


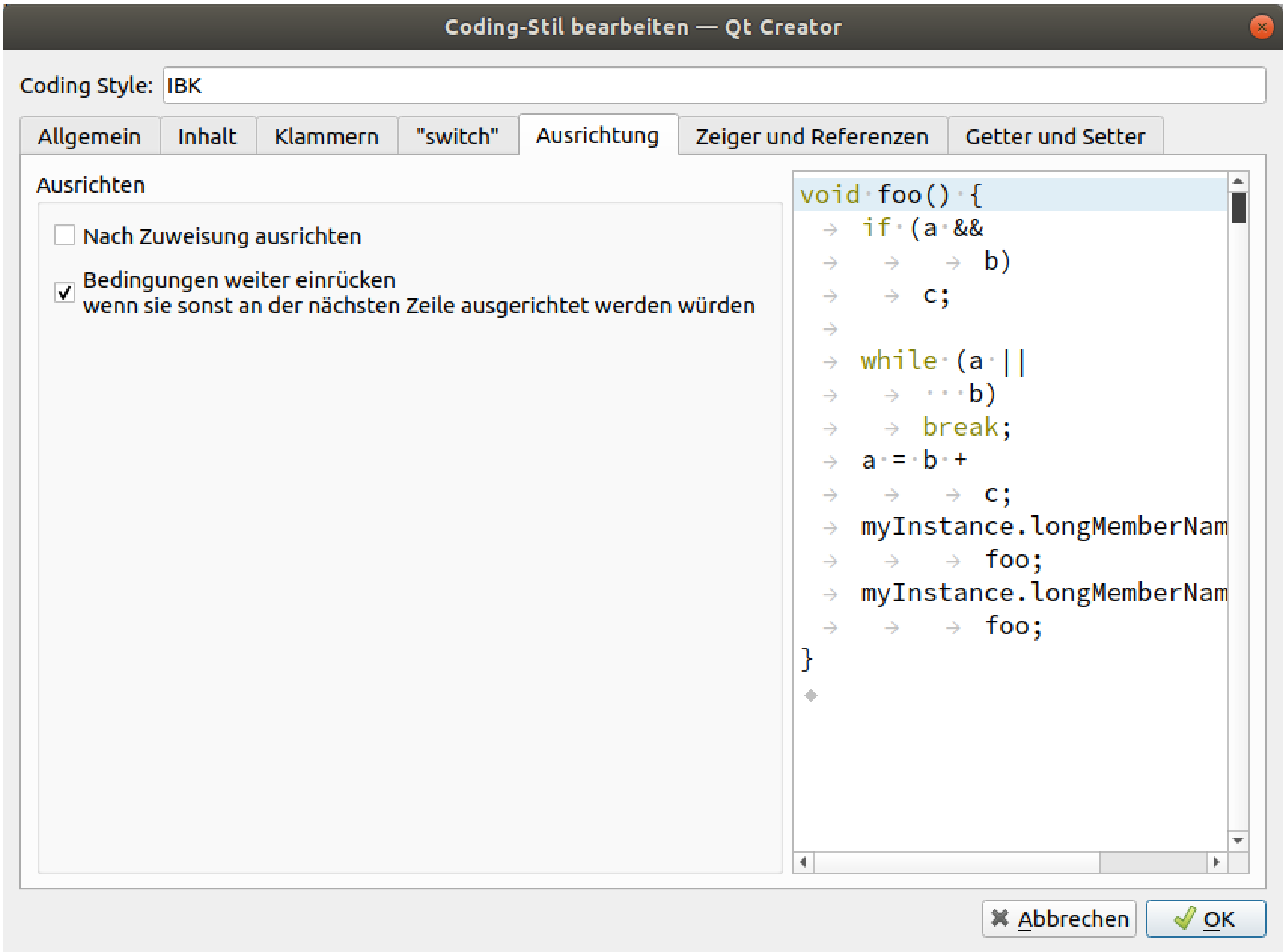
2.3.2. Coding style

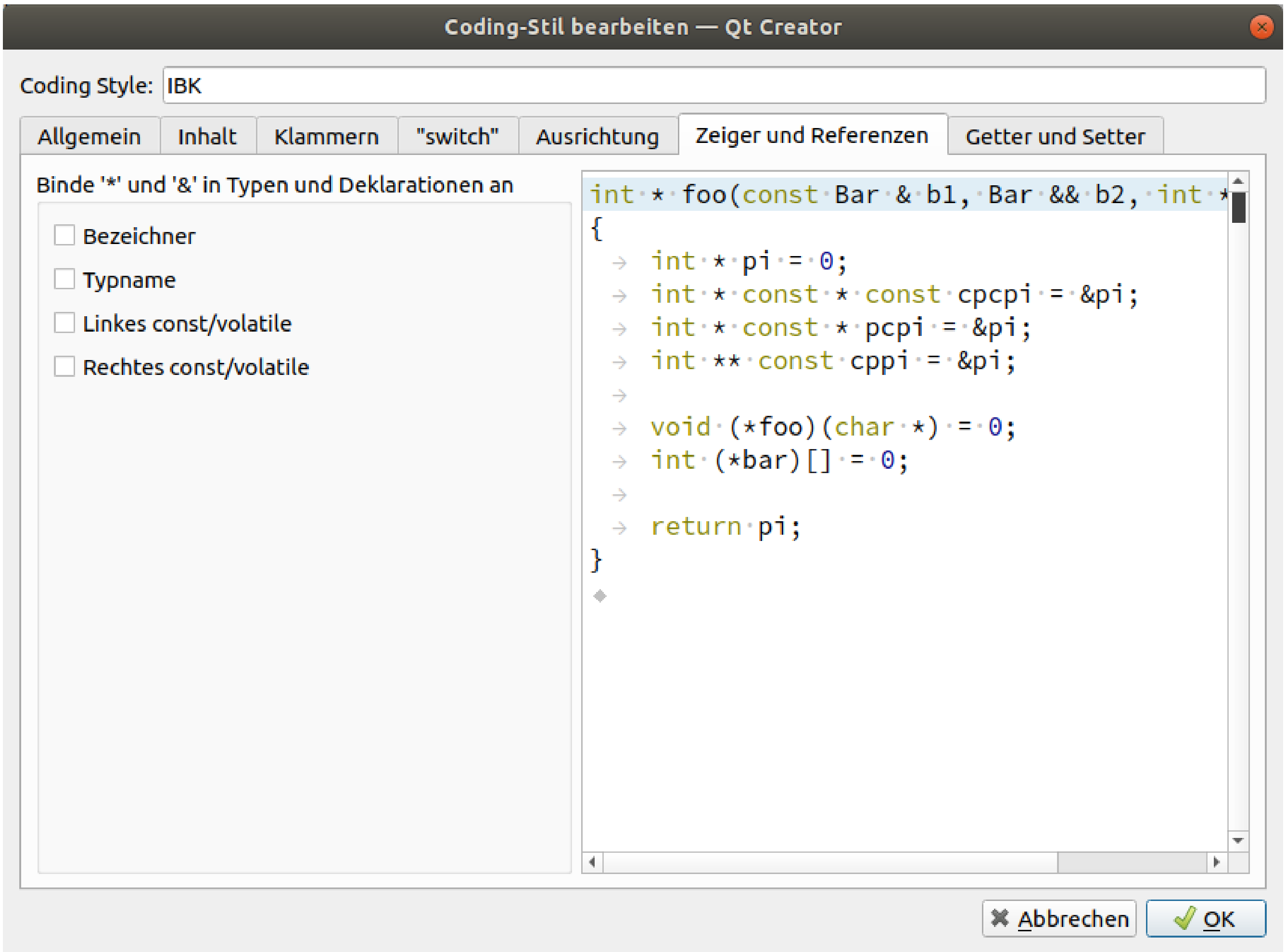
Create a custom coding style (copy from Qt-style), name it "IBK" and change it as follows (not shown configuration pages need not be changed):


Coding-Stil bearbeiten — Qt Creator
Coding Style: IBK
Allgemein
Inhalt
Klammern
"switch"
Ausrichtung
Zeiger und Referenzen
Getter und Setter
Tabulatoren und Einrückung
Tabulatorverhalten:
Nur Tabulatoren
Tabulatorweite: 4
Einrückung: 4
Einrückung von Fortsetzungszeilen:
Benutze normale Einrückung
#include <math.h>
class Complex
{
public:
Complex(double re, double im)
: _re(re), _im(im)
{}
double modulus() const
{
return sqrt(_re * _re + _im * _im)
}
private:
double _re;
double _im;
};
void bar(int i)
{
static int counter = 0;
Abbrechen
OK


Coding-Stil bearbeiten — Qt Creator
Coding Style: IBK
Allgemein
Inhalt
Klammern
"switch"
Ausrichtung
Zeiger und Referenzen
Getter und Setter
Einrückung
"public", "protected" und "private" im Klassenrumpf
Deklarationen relativ zu "public", "protected" und "private"
Anweisungen im Funktionsrumpf
Anweisungen in Blöcken
Deklarationen in Namensraum-Definition
#include <math.h>
class Complex
{
public:
Complex(double re, double im)
: _re(re), _im(im)
{}
double modulus() const
{
return sqrt(_re * _re + _im * _im);
}
private:
double _re;
double _im;
};
void bar(int i)
{
static int counter = 0;
counter += i;
Abbrechen
OK


Coding-Stil bearbeiten — Qt Creator
Coding Style: IBK
Allgemein
Inhalt
Klammern
"switch"
Ausrichtung
Zeiger und Referenzen
Getter und Setter
Einrückung in "switch"
"case" oder "default"
Anweisungen relativ zu "case" oder "default"
Blöcke relativ zu "case" oder "default"
"break"-Anweisung relativ zu "case" oder "default"
#include "bar.h"
int foo(int a)
{
switch (a)
{
case 1:
bar(1);
break;
case 2:
{
bar(2);
break;
}
case 3:
default:
bar(3);
break;
}
return 0;
}
Abbrechen
OK


Coding-Stil bearbeiten — Qt Creator
Coding Style: IBK
Allgemein
Inhalt
Klammern
"switch"
Ausrichtung
Zeiger und Referenzen
Getter und Setter
Ausrichten
Nach Zuweisung ausrichten
Bedingungen weiter einrücken wenn sie sonst an der nächsten Zeile ausgerichtet werden würden
void foo() {
if (a &&
b)
c;
while (a ||
b)
break;
a = b +
c;
myInstance.longMemberNam
foo;
myInstance.longMemberNam
foo;
}
Abbrechen
OK



2.3.3. Other coding style settings:

- C++ → Namenskonventionen für Dateien → Kleinbuchstaben für Dateinamen verwenden = off

2.3.4. Codemodel

The code model is responsible for checking the code while typing and can detect quite a few problems from mismatching types, misspelled variables, missing ; and basically everything a regular compiler can spot. In fact, the code model just runs the code through the first stages of the compiler - saving you quite a bit of compilation time.

The code model integration into Qt Creator is pretty nice, so you should activate it.

You can use one of the provided code model configurations, but that might lead to excessive number of errors/warnings. Rather configure the code model with the following parameters:

Codemodel Options for CLang on Linux

```
-Weverything -Wno-c++98-compat -Wno-c++98-compat-pedantic -Wno-unused-macros -Wno-newline-eof -Wno-exit-time
-destructors -Wno-global-constructors -Wno-gnu-zero-variadic-macro-arguments -Wno-documentation -Wno-shadow -Wno-switch
-enum -Wno-missing-prototypes -Wno-used-but-marked-unused -Wno-shorten-64-to-32 -Wno-old-style-cast
```

2.3.5. Efficient use of the Qt Creator IDE

i. TODO … (maybe a video will be better?)

2.4. Code Quality

This section discusses a few techniques that help you write/maintain high quality code.

2.4.1. Frequently check for potential memory leaks

Use valgrind to check for memory leaks in regular intervals:

First run only initialization with `--test-init` flag.

```
> valgrind --track-origins=yes --leak-check=full ./NandradSolver /path/to/project --test-init
```

You should get an output like:

```
...
Stopping after successful initialization of integrator.
Total initialization time: 802 ms
==15560==
==15560== HEAP SUMMARY:
==15560==     in use at exit: 0 bytes in 0 blocks
==15560==   total heap usage: 3,776 allocs, 3,776 frees, 1,101,523 bytes allocated
==15560==
==15560== All heap blocks were freed -- no leaks are possible
==15560==
==15560== For counts of detected and suppressed errors, rerun with: -v
==15560== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Do this check with:

- just the initialization part (i.e. with `--test-init`) parameter
- run the initialization with some error in the input file to check if temporary variables during initialization are cleaned up correctly
- also run a small part of the simulation, to check if something goes wrong during actual solver init and if tear-down is done correctly
- run a small part of the simulation, then break (`Ctrl+C`) and check if code cleanup after error abort is done correctly

> Of course, in very flexible code structures as in NANDRAD solver, where many code parts are only executed for certain parameter combinations, checking all code variables for consistent memory allocation/deallocation is nearly impossible. Hence, writing safe code in the first place should be highest priority.

Example: Avoiding memory leaks

NANDRAD creates model objects on the heap during initialization (never during solver runtime!). Since the model objects are first initialized before ownership is transferred, you should always ensure proper cleanup in case of init exceptions. Use code like:

```
ModelObject * modelObject = new ModelObject; // does not throw
m_modelContainer.push_back(modelObject); // transfer ownership, does not throw

modelObject->setup(...); // this may throw, but model object will be cleaned as part of m_modelContainer cleanup
```

If there is code between creation and ownership transfer, use code like:

```
std::unique_ptr<ModelObject> modelObject(new ModelObject);

modelObject->setup(...); // this may throw

m_modelContainer.push_back(modelObject.release()); // transfer ownership
```

3. NANDRAD Solver

This chapter discusses the underlying fundamentals and algorithms, as well as initialization and calculation procedures.

3.1. Model Initialization Procedure

3.1.1. Pre-Solver-Setup steps

The following steps are done when initializing the model:

- parsing command-line and handling early command line options
- setting up directory structure and message handler/log file
- creating `NANDRAD::Project` instance
- setting default values via call to `NANDRAD::Project::initDefaults()`
- reading the project file (only syntactical checks are done, and for IBK::Parameter static arrays with default units in keyword list, a check for compatible units is made), may overwrite defaults; correct units should be expected in the data model after reading the project file (maybe add suitable annotation to code generator?)
- merge similarly behaving construction instances (reduce data structure size)

> From now on the project data structure remains unmodified in memory until end of solver runtime and persistent pointers can be used to address parameter sets. This means no vector resizing is allowed, no more data members may be added/removed, because this would invalidate pointers/references to these vector elements.

Error checking in NANDRAD data model

Some basic error checking can be made in the NANDRAD Project data structure that is independent of the actual simulation-dependend model setup. These tests are done in functions called `checkParameters()`.

Basically, the NANDRAD Model calls these `checkParameters()` functions for all data objects. The functions should check for sane values (i.e. positive cross-section areas, non-negative coefficients, and generally all parameters with known limits. Also, the existence of parameters when required should be tested (in sofar this is independent of specific modelling options).

Error handling shall be done in such a way, that when parameter errors are found, the user can quickly identify the offending XML tag and fix the problem.

Parameter checking function of a vector element is wrapped in a try-catch clause with indication on where the error occurred.

```
for (unsigned int i=0; i<m_project->m_materials.size(); ++i) {
    NANDRAD::Material & mat = m_project->m_materials[i];
    try {
        mat.checkParameters();
    }
    catch (IBK::Exception & ex) {
        throw IBK::Exception(ex, IBK::FormatString("Error initializing material #%1 '%2' (id=%3).")
                             .arg(i).arg(mat.m_displayName).arg(mat.m_id), FUNC_ID);
    }
}
```

Note, that in the exception the index, optional display name and ID is printed - one of these should be sufficient to find the problematic XML block.

The location of error is reported by the calling function, inside the `checkParameters()` function it is sufficient to indicate the actual parametrization error:

Using consistent error checking functionality for IBK::Parameter data types.

```
void Material::checkParameters() {
    // check for mandatory and required parameters and
    // check for meaningful value ranges
    m_para[P_Density].checkedValue("kg/m3", "kg/m3", 0.01, false, std::numeric_limits<double>::max(), true,
                                   "Density must be > 0.01 kg/m3.");
    m_para[P_HeatCapacity].checkedValue("J/kgK", "J/kgK", 100, true, std::numeric_limits<double>::max(), true,
                                   "Heat capacity must be > 100 J/kgK.");
    m_para[P_Conductivity].checkedValue("W/mK", "W/mK", 1e-5, true, std::numeric_limits<double>::max(), true,
                                   "Thermal conductivity must be > 1e-5 W/mK.");
}
```

The exception hierarchy will be collected in the `IBK::Exception` objects and printed in the error stack (see also Exception handling).

For (the very frequently occurring) `IBK::Parameter` data type, the `checkedValue()` is most convenient to both check for existence and validity of an `IBK::Parameter` (including matching units). The function also returns the value in the requested target unit (see documentation for `IBK::Parameter::checkedValue()`).

Quick-access connections between data objects during runtime

Access of model parameters is very fast during simulation, if looping and searching through the data structure can be avoided. Pointers between data structures are an efficient way of relating data objects. For example, a construction layer references the respective material data via ID number of the material. During the initialization (specifically in `checkParameters()` a pointer is obtained to the material and stored in the object. This requires, of course, that the pointer may never become invalidated during the life-time of the simulation. Hence the **strict requirement** of not-changing vector sizes (see above).

If a references data element is missing, an error message is thrown (this is part of the reason, why this lookup of referenced data objects is done and checked in `checkParameters()`.

Summary

- implement `checkParameters()` functions in NANDRAD data model classes
- for cross references between data members (via ID numbers of ID names), create fast access pointer links in this function
- indicate the source of an error (i.e. the actual object) in the **calling** function

3.1.2. Model Setup

Now the actual model initialization starts.

TODO : model specific documentation

3.1.3. Climatic loads

Implementation

The `Loads` model is a pre-defined model that is always evaluated first whenever the time point has changed. It does not have any other dependencies.

It provides all resulting variables as `constant` (during iteration) result variables, which can be retrieved and utilized by any other model.

With respect to solar radiation calculation, during initialization it registers all surfaces (with different orientation/inclination) and provides an ID for each surface. Then, models can request direct and diffuse radiation data, as well as incidence angle for each of the registered surfaces.

Registering surfaces

Each construction surface (interface) with outside radiation loads registers itself with the Loads object, hereby passing the interface object ID as argument and orientation/inclination of the surface. The loads object itself registers this surface with the climate calculation module (CCM) and retrieves a surface ID. This surface ID may be the same for many interface IDs.

The Loads object stores a mapping of all interface IDs to the respective surface IDs in the CCM. When requesting the result variable’s memory location, this mapping is used to deliver the correct input variable reference/memory location to the interface-specific solar radiation calculation object.

3.2. Model objects, published variables and variable look-up

Physical models are implemented in model objects. That means the code lines/equations compute results based on input values.

Example 1. Model object

A heating model takes room air temperature (state dependency) and a scheduled setpoint (time dependency) and computes a heating load as result.

The result is stored in a **persistent memory location** where dependent models can directly access this value.

3.2.1. Model instances

There can be several model instances - the actual object code resides only once in solver memory, but the functions are executed several times for each individual object (=model instance).

Example 2. Model instance

You may have a model that computes heat flux between walls and zones. For each wall-zone interface, an object is instantiated.

Each model instance stores its result values in own memory.

3.2.2. Model results

Publishing model results

The model instances/objects must tell the solver framework what kind of outputs they generate. Objects generating results must derive from class `AbstractModel` (or derived helper classes) and implement the abstract interface functions.

`Schedules` and `FMIInputOutput` models are handled differently, when it comes to retrieving generated results.

The framework first requests a reference type (prefix) via `AbstractModel::referenceType()` of the model object. This reference type is used to group model objects into meaningful object groups.

Example 3. Reference types

Typical examples are `MRT_ZONE` for zonal quantities, or `MRT_INTERFACE` for quantities (fluxes) across wall-surfaces.

Each invididual result variable is published by the model in an object of type `QuantityDescription`. The framework requests these via call to `AbstractModel::resultDescriptions()`. Within the `QuantityDescription` structure, the model stores for each computed quantity the following information:

- id-name (e.g. "Temperature")
- physical display unit (e.g. "C") - interpreted as "display unit", calculations are always done in base SI units
- a descriptive text (e.g. "Room air temperature") (optional, for error/information messages)
- physical limits (min/max values) (optional, may be used in iterative algorithms)

- flag indicating whether this value will be constant over the lifetime of the solver/integration interval

> *A note on units*
>
> The results are stored *always* in base SI units according to the IBK-UnitList. A well-behaving model will always store the result value in the basic SI unit, that is "K" for temperatures, "W" for thermal loads and so on (see `IBK::UnitList`).

The unit is really only provided for error message outputs and for checking of the base SI unit matches the base SI unit of the requested unit (as an additional sanity check).

Vector-valued results

Sometimes, a model may compute a vector of values.

Example 4. Vector results

A ventilation model may compute ventilation heat loads for a number of zones (zones that are identified via object lists).

When a so-called vector-valued quantity is generated, the following additional information is provided:

- whether index-based or id-based access is anticipated
- a list of ids/indexes matching the individual positions in the vector (the size of the vector is also the size of the vector-valued quantities)

Example 5. ID-access example

The aforementioned ventilation model may be assigned to zones with IDs 1,4,5,10 and 11. Then the resulting ventilation heat losses will be defined for those zones only. Hence, the published quantity will look like:

```
- name = "NaturalVentilationHeadLoad"
- unit = "W"
- description = "Natural ventilation heat load"
- index-key-type = "ID"
- ID-list = {1,4,5,10,11}
```

> *Strong uniqueness requirement*
>
> The id-names of quantities are unique within a model object, and vector-valued quantities may not have the same ID name as scalar quantities. More precisely, variables need to be globally unique (see lookup rules below). This means that when there are two models with the same model-reference-type, the variable names must be unique among all models with the same reference-type.
>
> For example: from a zone parametrization several models are instantiated with the same reference-type `MRT_ZONE`, for example `RoomBalanceModel` and `RoomStatesModel`. The resulting variables in both models *must not* have the same ID name, in order to be uniquely identified on global scope.

Initializing memory for result values

Each model object is requested by the framework in a call to `AbstractModel::initResults()` to reserve memory for its result variables. For scalar variables this is usually just resizing of a vector of doubles to the required number of result values. For vector-valued quantities the actual size may only be known later, so frequently here just the vectors are created and their actual size is determined later.

> Since the information collected in `initResults()` is also needed when publishing the results to the framework, the function `AbstractModel::initResults()` is called before `AbstractModel::resultDescriptions()`.

Convenience implementation

Scalar variables are stored in double variables of the model. When using the convenience implementation in `DefaultModel` these are stored in vector `m_results`.

Vector-valued variables are stored in consecutive memory arrays with size matching the size of the vector. When using the `DefaultModel` implementation, these are stored in `m_vectorValuedResults`, which is a vector of `VectorValuedResults`.

The `DefaultModel` implementation makes use of enumerations `Results` and `VectorValuedResults`.

3.2.3. Model inputs

Similarly as output variables, model objects need input variables. Models requiring input must implement the interface of class `AbstractStateDependency`.

Input variable requirements are published similarly to results when the framework requests them in a call to `AbstractStateDependency::inputReferences()`. The information on requested results is stored in objects of type `InputReference`. The data in class `InputReference` is somewhat similar to that of `QuantityDescription` but contains additional data regarding the expectations of the model in the input variable.

> A model may request scalar variable inputs only, even if the providing model generates these as a vector-valued quantity. That means, a model has the choice to request access to the entire vector-valued variable (and will usually get the address to the start of the vector-memory space), or a single component of the vector. In the latter case, the index/model-ID must be defined in the `InputReference` data structure.

Example 6. Input reference to a zone air temperature inside a ventilation model

An input required by the ventilation model can be formulated with the follwing data:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "AirTemperature"
```

Given that information, the framework can effectively look-up the required variables.

Once the variable has been found, the framework will tell the object the memory location by calling

`AbstractStateDependency::setInputValueRef()`.

FMI Export (output) variables

When FMI export is defined, i.e. output variables are declarted in the FMI interface, a list of global variable IDs to be exported is defined. For each of these variables an input reference is generated (inside the `FMIInputOutput` model), just as for any other model as well.

Example 7. FMI Output Variable example

> Suppose an FMI exports the air temperature from zone id=15 and for that purpose needs to retrieve the currently computed temperature from the zone state model. The FMI output variable would be named *Zone[15].AirTemperature* and the input reference would be created as in the example above. This way, the framework can simply provide a pointer to this memory slot to the `FMIInputOutput` model just as for any other model.

Outputs

When initializing outputs, any published variable can be collected. Outputs declare their input variables just as any other model object.

3.2.4. Variable lookup

The frameworks job is to collect all

Resolving persistant pointers to result locations

Later, when the framework connects model inputs with results, the framework requests models to provide persistant memory locations for previously published results. This is done by calling `AbstractModel::resultValueRef()`, which get's a copy of the previously exported `InputReference`.

In order to uniquely identify a result variable **within** a model, normally only two things are needed:

- the ID name of the variable,
- and, *only in the case of vector-valued quantities*, the index/id.

However, in some cases, a model may request a variable with the *same quantity* name, yet from two different objects (for example, the air temperature of neighbouring zones). In this case, the quantity name alone is not sufficient. Hence, the full input reference including object ID is passed as identifier (**A change from NANDRAD 1!**).

> To identify an element within a vector-valued result it is not necessary to specify whether it should be index or id based - the model publishing the result defines whether it will be index or id based access.

Naturally, for scalar result variables the index/id property is ignored.

The `QuantityName` struct contains this information (a string and an integer value).

Now the model searches through its own results and tries to find a matching variable. In case of vector-valued quantities it also checks if the requested id is actually present in the model, and in case of index-based access, a check is done if the index is in the allowed range (0…size-1).

If a quantity could be found, the corresponding memory address is returned, otherwise a nullptr. The framework now can take the address and pass it to any object that requires this input.

Global lookup rules/global variable referencing

To uniquely reference a resulting variable (and its persistent memory location), first the actual model object/instance need to be selected with the following properties:

- the type of object to search for (= reference-type property), for example `MRT_ZONE` or `MRT_CONSTRUCTIONINSTANCE`
- ID of the referenced object, i.e. zone id oder construction id.

Some model objects exist only once, for example schedules or climatic loads. Here, the reference-type is already enough to uniquely select the object.

Usually, the information above does select several objects that have been created from the parametrization related to that ID. For example, the zone parameter block for some zone ID generates several zone-related model instances, all of which have the same ID. Since their result variables are all different, the framework simply searches through all those objects until the correct variable is found. These model implementations can be thought of as one model whose equations are split up into several implementation units.

The actual variable *within* the selected object is found by ID name and optionally vector-element id/index, as described above.

The data is collected in the class `InputReference`:

- `ObjectReference` (holds reference-type and referenced-object ID)
- `QuantityName` (holds variable name and in case of vector-valued quantities also ID/index)

Also, it is possible to specify a `constant` flag to indicate, that during iterations over cycles the variable is to be treated constant.

> If several model objects are addressed by the same reference-type and ID (see example with models from zone parameter block), the variable names must be unique among all of these models.

FMI Input variable overrides

Any input variable requested by any other model can be overridden by FMU import variables. When the framework looks up global model references, and an FMU import model is present/parametrized, then first the FMI generated quantity descriptions are checked. The FMU import variables are exported as global variable references (with ObjectReference). Since these are then the same global variable identifiers as published by the models, they are found first in the search and dependent models will simply store points to the FMU variable memory.

Examples for referenced input quantities

Setpoint from schedules

Schedules are defined for object lists. Suppose you have an object list name "Living rooms" and corresponding heating/cooling setpoints.

Now a heating model may be defined that computes heating loads for a given zone. The heating model is implemented with a simple P-controller, that requires zone air temperature and zone heating setpoint.

Definition of the input reference for the zone air temperature is done as in the example above. The setpoint will be similarly referenced:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "HeatingSetPoint"
```

4. SIM-VICUS User-Interface

4.1. 3D Engine

4.1.1. Shaders

Vertex-Layout notation:

- V - single coordinate (e.g. x-coordinate)
- N - component of normal vector
- C - color component (e.g. CCC for RGB)
- T - texture coordinate

Grid

- Shader-Index: 0
- Vertex-Structure: (VV) = (xz)
- Files: `grid.vert` and `grid.frag`
- Uniforms (variables):
  - `worldToView`
  - `gridColor`
  - `backColor` (fade-to color)
  - (optional) `farDistance` (fade-to normalization distance)

Background Objects

Uniform color, no lighting effect, no transparency

- Shader-Index: 1
- Vertex-Structure: (VVVCCC) = (xyzrgb)
- Files: `vertexColor.vert` and `flat.frag`
- Uniforms (variables):
  - `worldToView`

Regular Opaque Objects

- Shader-Index: 2
- Vertex-Structure: (VVVNNNCCC) = (xyzNxNyNzrgb)
- Files: `vertexNormalColor.vert` and `specularShading.vert`
- Uniforms (variables):
  - `worldToView`
  - `lightPos`