Contents

[Motivation 2](#_Toc94710620)

[C++ for writing data intensive applications 2](#_Toc94710621)

[A DSL has the following properties: 2](#_Toc94710622)

[General architecture of the framework 3](#_Toc94710623)

[Persistence classes 4](#_Toc94710624)

[Example application 5](#_Toc94710625)

[Explanation of problem domain 5](#_Toc94710626)

[Template Metaprogramming in C++ Motivation and Definition 9](#_Toc94710627)

[Introduction 9](#_Toc94710628)

[Computation Examples 10](#_Toc94710629)

[Numeric calculations 10](#_Toc94710630)

[Type computations and control structures 11](#_Toc94710631)

[Framework elements 12](#_Toc94710632)

[Support for compound record types 17](#_Toc94710633)

[Safe approach to DB Foreign Key constraints 17](#_Toc94710634)

[Persistent Comboboxes and Listboxes: BoxContents component 18](#_Toc94710635)

[The JoinedGridDisplayer component 19](#_Toc94710636)

[The RefIntegrityManager component 22](#_Toc94710637)

[The Data.h component 23](#_Toc94710638)

[The data.cpp component 24](#_Toc94710639)

[The RecordLinks component 24](#_Toc94710640)

[The Connections component 26](#_Toc94710641)

[Dependencies 26](#_Toc94710642)

[Conclusions 30](#_Toc94710643)

[Future development 31](#_Toc94710644)

[Bibliography 31](#_Toc94710645)

# Motivation

C++ is the language of choice for infrastructure development, but many don’t realize that it can be used for ordinary applications with surprising results. The main problem faced by these application types is how to deal with persistence, i.e., database access and modification. The current language of choice for this type of development is SQL, because of its declarative nature and clear relational conceptual model. Its main purpose is accessing the normalized tables and creating runtime arbitrary record types from potentially complex select queries. When the application language is object oriented the question is how we deal with the concept of objects versus the concept of records. A very complete library we have been using for some time allows us to write SQL directly in C++ (a DSL library, see below) without the use of strings at the application level of the code. It maps C++ structs with normalized tables in the database in a declarative manner. More complex objects formed by the joining of normalized tables can be dealt with as needed by different use cases that contain the application’s functionality.

With the problem domain expressed conceptually in SQL syntax, the applications are written in terms belonging to the area of such domain. Therefore, it allows a very object-oriented development in declarative style.

Finally, it becomes necessary to provide extensions of GUI components, so they interact with the DB in an intuitive and syntactically simple manner. The combination of such GUI controls on top of the SQL DSL allows the creation of a framework that enables fast application declarative development in a language that few would consider able to do so.

After tutorial like introduction to the various concepts central to this document, we are going to describe the main elements of the framework applied to a very simple data management application.

# C++ for writing data intensive applications

C++ is a large language with a very expressive and rich syntax. Perhaps its most salient characteristic is its ability to control the level of abstraction enabling application programming to be done in terms of the problem domain’s concepts. An extension to this capability is the ability to define Domain Specific Languages (DSL) like SQL in terms of “ordinary” C++ code. This makes C++ a very compelling language for writing data intensive applications.

## A DSL has the following properties:

* It is a language, that defines: [CPPTM,216]
  + An alphabet (set of symbols)
  + Well defined rules stating how the alphabet may be used to build well-formed compositions
  + A well-defined subset of all well-formed compositions that are assigned specific meanings
* It is domain specific not general-purpose
  + Examples include regular expressions, UML, Morse code
  + By this restriction, we gain significantly higher level of abstraction and expressiveness because
    - Specialized alphabet and notations allow pattern-matching that matches our mental model
    - Enabling writing code in terms close to the abstractions of the problem domain is the characteristic property and motivation behind all DSLs
    - We use the language’s notation to write down a statement of the problem itself and the language’s semantics take care of generating a solution
    - The most successful DSLs are often ***declarative*** languages providing us with notations to describe what rather than how
      * The how can be seen as a consequence of the what
* In a sense, DSL is an enhancement to object-oriented programming in which the development is done in terms of the problem domain conceptual model
  + We are just taking an extra step towards enriched notational support

In particular a SQL DSL like that found in the ***sqlite\_orm*** library, provides direct support for writing SQL in C++. This is indeed a worthy capability and one that allows for the clear concise creation of data intensive applications. This particular library is perhaps more a SQL DSL than an object relational tool. However this is not a disadvantage, rather it truly enables the application to be written (read “thought of”) in terms of fourth generation SQL concepts.

One can truly raise C++ abstraction level by thinking in a combination of imperative C++ enhanced with compile time metaprogramming and SQL. The synergy is indeed attractive and powerful.

In this article I will present a framework that provides persistence to windows’ controls and enables fast primarily declarative programming in C++ and SQL DSL.

# General architecture of the framework

Using a UML ‘extended’ component diagram to express the general architecture of the proposed framework we can illustrate it like this:

Diagram

Description automatically generated

# Persistence classes

We start with 1-1 class-table for all classes that require persistence, thus the normal form used to partition the database (DB) is visible at the application code level. However arbitrarily complex record types can be created as required by use cases via arbitrarily complex select statements with joins. These complex record types can be displayed in grids through the template JoinedGridDisplayer (see below).

The persistent classes have methods for performing class specific queries, but more general queries are associated with application dialogs as they are tied to specific use cases for which these dialog boxes exist. This keeps persistent classes simple, which is very important for relational integrity. Recall that types composed from these simple classes can have data belonging to different persistent classes. Type deduction mechanisms in modern C++ allows us to create composite persistent types on the fly or encapsulate them within particular classes.

# Example application

A very small and simple application has been created to illustrate the framework. We have created much larger applications with many more tables and dialog boxes (i.e., use cases). It comes from the world of Investment portfolios, and it is composed from only 3 normalized classes:

1. Investment Funds (called simply Fondos in the code) with data slots:
   1. Abbreviation (abreviacion)
   2. Name (nombre)
   3. Return frequency (tipo\_cupon: mensual o trimestral)
2. Investment Returns (called Rendimientos in the code) with data slots:
   1. Return per stock (rendimiento\_unitario)
   2. Date of return (fecha)
   3. Investment fund returning (fkey\_fondo)
3. Investments (called Inversiones in the code) with data slots:
   1. Number of stocks purchased or sold (num\_participaciones)
   2. Date of transaction (beginning\_date)
   3. Investment fund affected (fkey\_fondo)

## Explanation of problem domain

The investment funds are financial instruments that produce returns on investment according to their return frequency, be it monthly or by trimester. They have very simple data requirements, only a name and an abbreviation. The investor is allowed to purchase or sell stocks and the investment returns from each specific investment fund depends on how many stocks he owns and the current investment return per stock. Thus, each fund produces different amounts of returns depending on the date which establishes the corresponding return per stock times the number of currently owned stocks. These calculations are specific for each fund and thus appear as instance operations in the Investment Fund class. The reader may wonder why we use *struct* instead of *class* for the normalized classes and it is really a matter of taste and convenience. The struct’s default visibility is public and the omission of constructors enables the use of aggregate initialization which is very convenient in C++. Being persistent slots of carefully normalized classes, they do not change often at all. Furthermost, the corresponding object relational mapping is not frequently affected either. The implementation details are mostly in the mapping so get\_ set\_ methods don’t hide any of these details, yet they further complicate the user code with very little, if at all, convenience. Even in the face of change, it is after all easier to replace data slot names in the code than changing the get\_ and set\_ methods everywhere they are used. This is not a limitation of the DSL, which supports them, but rather a conscious design decision we have made. The get\_ and set\_ methods we dislike for persistent classes give a false sense that object encapsulation is being followed and that implementation details are being encapsulated. In all truth, the implementation details are spread in the object relational mapping much more so than in the data slots themselves. Changes in the data slots mapping is not to be taken lightly. The normalization of the persistent classes must be made after a very careful domain analysis. Changes in them can cause the dropping of tables and the loss of information. The DB must be carefully versioned should there be any need to change the mapping. The impact on the whole application code is very serious and is not going to be improved by trivial get\_ set\_ methods (because for persistent classes, these methods are indeed trivial).

An entity-relationship diagram displays the simple nature of the example program:

Diagram

Description automatically generated

The normalized classes in C++ with the object-relational mapping follows:

struct Fondo

{

int id;

std::string abreviacion;

std::string nombre;

int tipo\_cupon;

enum TipoCupon

{

mensual = 1, trimestral = 3

};

std::string simple\_dump() const noexcept;

int num\_participaciones\_al(std::chrono::sys\_days fecha) const noexcept;

double get\_rendimiento\_unitario\_al(std::chrono::year\_month\_day fecha) const noexcept;

};

* Object relational mapping like this:

make\_table("Fondos",

make\_column("id\_fondo", &Fondo::id, autoincrement(), primary\_key()),

make\_column("nombre", &Fondo::nombre),

make\_column("abrev", &Fondo::abreviacion),

make\_column("tipo\_cupon", &Fondo::tipo\_cupon))

struct Inversion

{

int id;

int num\_participaciones;

std::chrono::sys\_days beginning\_date;

int fkey\_fondo;

std::string simple\_dump() const noexcept;

};

* Object relational mapping like this:

make\_table("Inversiones",

make\_column("id\_inversion", &Inversion::id, autoincrement(), primary\_key()),

make\_column("fkey\_fondo", &Inversion::fkey\_fondo),

make\_column("begin\_date", &Inversion::beginning\_date),

make\_column("num\_participaciones", &Inversion::num\_participaciones),

foreign\_key(&Inversion::fkey\_fondo).references(&Fondo::id))

struct Rendimiento

{

int id;

int fkey\_fondo;

double rendimiento\_unitario;

std::chrono::sys\_days fecha;

std::string simple\_dump() const noexcept;

};

* Object relational mapping like this:

make\_table("Rendimientos",

make\_column("id\_rendimiento", &Rendimiento::id, autoincrement(), primary\_key()),

make\_column("fecha", &Rendimiento::fecha),

make\_column("rend\_unitario", &Rendimiento::rendimiento\_unitario),

make\_column("fkey\_fondo", &Rendimiento::fkey\_fondo),

foreign\_key(&Rendimiento::fkey\_fondo).references(&Fondo::id))

The database is defined by passing the tables to the function make\_storage. When we call storage.sync\_schema(true) the DB with all its tables, columns, constraints and triggers is created automatically! If changes are made to the object-relational mapping the DSL tries to update the DB structure preserving information, which succeeds for simple changes but requires DB versioning for more profound changes. Again, the need to perform a relational design of the problem domain cannot be left for later… it should be one of the first steps in data intensive development given its impact in the face of change.

inline auto& Storage\_Impl::get\_storage()

{

static auto storage =

make\_storage(db\_name,

make\_table("Fondos",

make\_column("id\_fondo", &Fondo::id, autoincrement(), primary\_key()),

make\_column("nombre", &Fondo::nombre),

make\_column("abrev", &Fondo::abreviacion),

make\_column("tipo\_cupon", &Fondo::tipo\_cupon)),

make\_table("Inversiones",

make\_column("id\_inversion", &Inversion::id, autoincrement(), primary\_key()),

make\_column("fkey\_fondo", &Inversion::fkey\_fondo),

make\_column("begin\_date", &Inversion::beginning\_date),

make\_column("num\_participaciones", &Inversion::num\_participaciones),

foreign\_key(&Inversion::fkey\_fondo).references(&Fondo::id)),

make\_table("Rendimientos",

make\_column("id\_rendimiento", &Rendimiento::id, autoincrement(), primary\_key()),

make\_column("fecha", &Rendimiento::fecha),

make\_column("rend\_unitario", &Rendimiento::rendimiento\_unitario),

make\_column("fkey\_fondo", &Rendimiento::fkey\_fondo),

foreign\_key(&Rendimiento::fkey\_fondo).references(&Fondo::id)));

storage.sync\_schema(true);

return storage;

}

The class Storage\_Impl has the following members which are only accessible to class Storage:

class Storage\_Impl

{

private:

Storage\_Impl() = delete; // prohibit instantiation

static auto& get\_storage();

static auto& get\_old\_storage();

static auto& get\_new\_storage();

static void copy\_old\_to\_new();

friend class Storage;

static constexpr const char\* db\_name{ "C:\\Users\\juan\_\\OneDrive\\Finances\\Investments.sqlite" };

};

class Storage

{

public:

using Storage\_t = decltype(Storage\_Impl::get\_storage());

Storage();

static void initialize();

static Storage\_t& getStorage() { return Storage\_Impl::get\_storage(); }

static void fill\_db\_with\_test\_data();

static void empty\_database();

static void upgrade\_database();

static void backup\_db();

};

The type of storage is a complex template which is therefore deduced by declaring Storage\_t. It’s complexity is completely hidden to the application and framework elements.

# Template Metaprogramming in C++ Motivation and Definition

## Introduction

[S,2013] “Programing that manipulate program entities, such as classes and functions, is commonly called metaprogramming.” Templates in C++ are used to generate classes and functions and template programming compute at compile time and generates programs.

Template meta programming is performed at compilation time and therefore does not affect runtime performance. Programs written with this style, which is becoming the de facto standard in modern C++, may run much faster than traditional C++ code (C++98). We should only know to be judicial in its use because it will delay compilation times.

[CPPTM, 3] Metaprogramming in C++ language was a major discovery of [Unruh,1994]. Here it was found that template compile time programming was supported by the language and indeed templates and their template instantiations mechanism are Turin complete! The compile time engine provides a pure functional programming language with no variables, increment operators and so on, and the only limitations to the problems that can be solved are compiler limitations such as the number of template instantiations and other compile time resources that are required by a template use. An example of template metaprogramming is a Lisp interpreter that was written in just a few pages using templates [S,2013, 780][Czarnecki,2000].

Generic programming should be distinguished from template metaprogramming in that although both use templates their intention is quite different. The first creates data structures and algorithms that are type safe: e.g. a List<Types> can be instantiated as a list of whatever type you desire (as long as it provides the minimum interface: the necessary operations that are required by List’s compilation). As long as the type conforms to that basic *interface*, we have the corresponding static polymorphism without explicit interfaces. This weakness is removed by the addition of C++20 *concepts*, which lets the programmer declaratively specify what the template expects of the template arguments. For instance, one can declare that a certain template requires an arithmetic supporting type or a type that can be copied, etc.

Template metaprogramming is a compile time *computation* that yields types or functions to be used at runtime. In other words, it contains compile time algorithms on compile time data structures (see Chapter 28 of S,2013). Many standard library components, such as function, thread and tuple are examples of relatively simple applications of metaprogramming techniques.

## Computation Examples

There are 2 basic kinds of computations that can be performed at compile time using templates: numeric calculations and type computations.

### Numeric calculations

Examples of the first can be seen in the implementation of a factorial function like this:

template<int N> requires (N > 0)

struct Factorial

{

static constexpr int value = N \* Factorial<N - 1>::value;

};

template<>

struct Factorial<1>

{

static constexpr int value = 1;

};

Notice how the algorithm is basically equivalent to this runtime code and how the second struct stops the recursion:

int factorial(int num)

{

return (num < 2) ? 1 : num \* factorial(num - 1);

}

The struct is called a meta-function and it only works with compile time constants. The template parameters work as “meta-function” parameters, and the nested constant value is the “meta-function” return type. Its use is also a little different in that we need to create a type to access the value:

using fact3 = Factorial<3>; // modern C++ “typedef” is using – to be preferred

typedef Factorial<3> Fact3; // old C++

std::cout << fact3::value << Fact3::value << std::endl;

This functionality can also be achieved using function templates, but they are less powerful. The reader recommended reading is chapter 28 of [S,2013]. Function templates examples are:

template<int N> requires (N > 0)

constexpr int FactorialFunc()

{

return N \* FactorialFunc<N - 1>();

}

template<>

constexpr int FactorialFunc<1>()

{

return 1;

}

Again, the template parameters are the input but this time the output is just the return value. This template can only be used by compile time constants for the parameter N. Its use is very simple:

constexpr int ret = FactorialFunc<3>();

If we allow for other solutions that do not use metaprogramming templates, we can define a function that is executable either at compile time or at runtime depending on whether the argument is a compile time constant or not. Just for completeness’ sake and because it can be used in compile time programming, we show an example:

constexpr int factorialAlt(int num)

{

return (num < 2) ? 1 : num \* factorialAlt(num - 1);

}

There are 2 ways of using it:

int variable = 3;

constexpr int res = factorialAlt(3); // compile time computation

int runtime\_res = factorialAlt(variable); // runtime time computation

### Type computations and control structures

Types being the compile time data, we need to provide a means for selection and recursion. We saw recursion in the past section where we analyzed numeric computations. How can we provide control structures for compilation time programming? It is simpler than one might think, for instance selecting a type from an arbitrarily large sequence of types we obtain the following solution by using variadic templates:

template<unsigned N, typename ...Cases>

struct select;

template<unsigned N, typename Head, typename...Tail>

struct select<N, Head, Tail...> : select<N-1, Tail...>

{};

template<typename Head, typename...Tail>

struct select<0, Head, Tail...>

{

using type = Head;

};

The implementation uses inheritance to produce the template instantiations[[1]](#footnote-1) for decreasing N until N becomes 0; this happens N times. The Head type at 0 is the type to select. The instantiation of the select<N,Head,Tail…> provokes the instantiation of select<N-1, Tail…> which removes the Head from the type list. Please do not confuse this with object-oriented subclassing.

It is used like this:

void useSelect()

{

using type = select<3, char, short, int, long, long long>::type;

static\_assert(std::is\_same<type, long>::value);

}

# Framework elements

First, use cases that require user interaction are represented by base dialog boxes which can use further windows as needed. Once the relational normalization is defined, the first step is to create one dialog box for maintaining each identified table. Doing so allows us to clarify each persistent classes restrictions like invariants, what we consider valid values for its data members[[2]](#footnote-2). These dialogs have a very simple API approximating this one:

Text

Description automatically generated with medium confidence

They are composed of MFC[[3]](#footnote-3) GUI controls and framework elements. One such typical dialog can be defined like this:

#include "..\ORM\_Extensions/BoxContents.h"

#include "Data.h"

#include "..\ORM\_Extensions/RadioButtonGroup.h"

class CFondoDlg : public CDialog

{

DECLARE\_DYNAMIC(CFondoDlg)

BoxContents<Fondo, &Fondo::id> m\_list\_all\_fondosLB;

std::optional<Fondo> m\_fondo{}; // persistent struct

RadioButtonGroup m\_tipo\_cupon;

public:

CFondoDlg(CWnd\* pParent = nullptr); // standard constructor

virtual ~CFondoDlg();

// Dialog Data

#ifdef AFX\_DESIGN\_TIME

enum { IDD = IDD\_CFondoDlg };

#endif

protected:

virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV support

DECLARE\_MESSAGE\_MAP()

private:

CListBox m\_list\_fondos;

CButton m\_mensual;

CButton m\_trimestral;

CEdit m\_fondo\_id;

CEdit m\_fondo\_abrev;

CEdit m\_fondo\_name;

public:

virtual BOOL OnInitDialog();

void Refresh();

afx\_msg void OnBnClickedApply();

afx\_msg void OnSelchangeListFondos();

afx\_msg void OnBnClickedNew();

afx\_msg void OnBnClickedErase();

virtual LRESULT WindowProc(UINT message, WPARAM wParam, LPARAM lParam);

};

The executing dialog box is:

Graphical user interface, application

Description automatically generated

Database access operations are always made inside try/catch clauses and have a standard exception handling (one simple function to handle applying changes):

void CFondoDlg::OnBnClickedApply()

{

m\_fondo = getCurrent<Fondo>(m\_fondo\_id);

int tipo;

m\_tipo\_cupon >> tipo;

std::string abrev, name;

m\_fondo\_abrev >> abrev;

m\_fondo\_name >> name;

try

{

if (!m\_fondo) // insert

{

m\_fondo = m\_list\_all\_fondosLB.insert(abrev, name, tipo);

}

else

{

m\_fondo->abreviacion = abrev;

m\_fondo->nombre = name;

m\_fondo->tipo\_cupon = tipo;

m\_list\_all\_fondosLB.update(\*m\_fondo);

}

setIdFromRecord<Fondo>(m\_fondo\_id, m\_fondo->id);

Refresh();

}

catch (std::exception& exc)

{

handleApply(exc);

}

}

A non-maintenance general use case dialog box will usually have one or more grids like this:

#include "..\ORM\_Extensions/IDisplayer.h"

#include "..\ORM\_Extensions/GridDisplayer.h"

#include "Data.h"

#include "..\ORM\_Extensions/JoinedGridDisplayer.h"

#include "..\ORM\_Extensions/BoxContents.h"

// RendimientosTableDlg dialog

class RendimientosTableDlg : public CDialog

{

DECLARE\_DYNAMIC(RendimientosTableDlg)

BoxContents<Fondo, &Fondo::id, CComboBox> m\_fondosCB;

public:

RendimientosTableDlg(CWnd\* pParent = nullptr); // standard constructor

virtual ~RendimientosTableDlg();

// Dialog Data

#ifdef AFX\_DESIGN\_TIME

enum { IDD = IDD\_RendimientosTableDlg };

#endif

protected:

virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV support

DECLARE\_MESSAGE\_MAP()

public:

void Refresh();

virtual BOOL OnInitDialog();

private:

CJDGridCtrl m\_rendimientos\_grid;

std::unique\_ptr<IDisplayer> m\_rendimientos\_displayer;

template<typename T>

void InitializeGridRendimientos(const T& whereClause);

CComboBox m\_fondos\_box;

CDateTimeCtrl m\_fecha\_inicio;

CDateTimeCtrl m\_fecha\_final;

CButton m\_filter\_by\_dates;

public:

afx\_msg void OnBnClickedBFilter();

afx\_msg void OnBnClickedBFilterClear();

afx\_msg void OnBnClickedBRendimiento();

};

The grid is defined using type erasure in the header file (note that only the abstract interface IDisplayer is declared). For more on this component look at JoinedGridDisplayer below.

The dialog box for this use case displays like this:

Graphical user interface

Description automatically generated

The filtering is handled by this code and can be composed from simple where clauses:

void RendimientosTableDlg::OnBnClickedBFilter()

{

auto fondo = m\_fondosCB.current();

auto fondo\_id = fondo ? fondo->id : -1;

auto fondoWhere = (not fondo or (c(alias\_column<als\_f>(&Fondo::id)) == fondo\_id));

bool filter\_by\_dates;

m\_filter\_by\_dates >> filter\_by\_dates;

std::chrono::sys\_days start\_date, finish\_date;

m\_fecha\_inicio >> start\_date;

m\_fecha\_final >> finish\_date;

auto filter\_by\_dates\_where = (not filter\_by\_dates or

(c(alias\_column<als\_r>(&Rendimiento::fecha)) >= start\_date

&& (c(alias\_column<als\_r>(&Rendimiento::fecha)) <= finish\_date)));

auto whereClause = fondoWhere && filter\_by\_dates\_where;

InitializeGridRendimientos(whereClause);

}

And it is cleared like this:

void RendimientosTableDlg::OnBnClickedBFilterClear()

{

// **TODO: Add your control notification handler code here**

m\_fondosCB.select(-1);

m\_filter\_by\_dates << false;

OnBnClickedBFilter();

}

Note that the operators >> and << make it very easy to insert or extract the values from GUI controls into or out of data members. These hide a lot of non-domain code. For instance all CEdit controls (text edit controls) use UNICODE character encoding while **sqlite\_orm** and the general application expects encoding in std::string. The implementation of these operators shows just how much is hidden in favor of coding simplicity and expressiveness:

inline void operator<<(CEdit& ctrl, std::string s)

{

SetText(ctrl, s);

}

template<typename T> requires std::is\_arithmetic<T>::value

inline void operator<<(CEdit& ctrl, T d)

{

SetAmount(ctrl, d);

}

template<typename Data>

void SetText(CEdit& edit\_box, Data data)

{

edit\_box.SetWindowTextW(Util::to\_cstring(data));

}

export inline CString to\_cstring(std::string msg)

{

std::wstring message = to\_wstring(msg);

CString msg\_as\_cstring{ message.c\_str() };

return msg\_as\_cstring;

}

// convert string to wstring

export inline std::wstring to\_wstring(const std::string& str, const std::locale& loc = std::locale{})

{

std::vector<wchar\_t> buf(str.size());

std::use\_facet<std::ctype<wchar\_t>>(loc).widen(str.data(), str.data() + str.size(), buf.data());

return std::wstring(buf.data(), buf.size());

}

Note that the code for string conversions is using C++20 modules[[4]](#footnote-4) (as suggested by the *export* keyword).

## Support for compound record types

Although an emphasis is placed on the class-table one to one relation, it is perfectly possible to create arbitrarily complex compound record types consisting of columns from more than one table. Such capability can be obtained by creating a struct modeled after the return of an arbitrarily complex SQL query. Another implementation might be simpler: model a composite struct by aggregating basic persistent objects. Which of these approaches is the best is currently unknown for such a scenario did not come up in the development of the example application or in other more complex applications we have created with the framework. As of now, we remain at simplicity: one class per table.

## Safe approach to DB Foreign Key constraints

A safe approach avoids using SQLITE native support for FK constraints. We know it is impossible to break FK-PK links due to constraint support at the level of the SQLITE engine, but we don’t want to depend on it because information is lost. For every record that is to be inserted or updated we verify that all FK correspond to existing PKs and report violations by name. Native SQLITE only reports violation of FK constraint without being explicit as to which FK was violated.

Likewise for record deletion, we like to verify that there are no dependent rows with FKs pointing to this record PK before proceeding with the attempt to remove the record. This makes it unnecessary to catch exceptions sent from deep withing the DSL library which could leave some functions in invalid states, something we don’t want to risk. No matter how good a library is, it is better to avoid exceptions from deep within it, if we can handle them at a higher level. This is the responsibility of the RecordLinks and Connections components.

RecordLinks is very much a forwarding component and one that *declares* the dependencies between tables and FK-PK links (see RecordLinks component below).

Connections component is metaprogramming at its best (see Connections component below).

## Persistent Comboboxes and Listboxes: BoxContents component

We would like to use comboboxes and listboxes that mirror record contents. Inserting or deleting a record into/from a box should also affect the DB and viceversa. BoxContents is the component with that responsibility. One important functionality is to “stringize” the record contents into a string displayable in the box. This is done by a lambda object in the BoxContents constructor in collaboration with the corresponding persistent record (by means of the standard method simple\_dump).

m\_list\_all\_fondosLB(m\_list\_fondos, [](const Fondo& fondo)

{

return Util::to\_cstring(fondo.simple\_dump());

})

Insertion is done like this:

template<typename ...Cols>

std::optional<Table> insert(Cols&&... cols)

{

Table record{ -1, cols... };

refIntManager.throwIfcannotInsertUpdate(record);

record.\*keyCol = storage.insert(record);

this->insert\_into\_listbox(record);

return record;

}

Deletion like this:

bool delete\_current\_sel()

{

using namespace sqlite\_orm;

auto current = this->current();

if (!current)

{

throw std::exception("No hay registro seleccionado");

}

refIntManager.throwIfcannotDelete(\*current);

storage.remove<Table>(get\_pk(\*current));

int cur\_sel = m\_box.GetCurSel();

m\_box.DeleteString(cur\_sel);

return true;

}

Access to the current record like this:

std::optional<Table> current()

{

std::optional<Table> record;

int cur\_sel = m\_box.GetCurSel();

if (cur\_sel != npos)

{

auto id = static\_cast<long long>(m\_box.GetItemData(cur\_sel));

record = refIntManager.get(id);

}

return record;

}

Selecting a particular record like this:

std::optional<Table> select(int pk)

{

std::optional<Table> record;

if (pk < 0)

{

m\_box.SetCurSel(npos);

return record;

}

int index = find\_index\_in\_listbox(pk);

if (index != npos)

{

m\_box.SetCurSel(index);

record = refIntManager.get(pk);

Posting::get().PostSelChangeNotification(m\_box);

}

return record;

}

## The JoinedGridDisplayer component

This component displays arbitrarily complex select queries in a grid. We declare the component in the header file of the dialog box, as:

std::unique\_ptr<IDisplayer> m\_rendimientos\_displayer;

and we build the object in the source code in these steps:

1. Define the headers text
2. Define the SQL select statement with all necessary joins and where/sort/group by clauses
3. Obtain a vector of records from the select
4. Create the JoinedGridDisplayer object passing headers, records, formatting information for currency, etc.
5. Order the object to display itself.
6. Define where and order\_by clauses to suit user needs for data presentation as defined in the corresponding use case

The code is as follows:

struct IDisplayer

{

virtual void display() = 0;

};

template<typename T, typename ColonesCols, typename DollarsCols>

class JoinedGridDisplayer : public IDisplayer

{

using Container = std::vector<std::remove\_reference\_t<T>>;

using RowType = typename Container::value\_type;

inline static constexpr size\_t NumCols = std::tuple\_size<RowType>::value;

CJDGridCtrl& grid;

Container lines;

std::vector<std::string> headers;

public:

// constructor

JoinedGridDisplayer(CJDGridCtrl& grid,

Container&& lines\_,

std::vector<std::string> headers\_)

: lines{ std::move(lines\_) }, grid{ grid }, headers{ std::move(headers\_) }

{

// reset grid

grid.SetRowCount(0);

grid.SetColumnCount(0);

// new grid

grid.SetColumnCount(NumCols + 1);

grid.SetRowCount(lines.size() + 1);

grid.SetFixedRowCount();

grid.SetFixedColumnCount();

grid.SetHeaderSort(true);

grid.SetSingleRowSelection(true);

grid.m\_sortingFunctions.resize(NumCols + 1);

grid.m\_sortingFunctions[0] = Util::Comparison::Text;

int col = 1;

for (auto& str : headers)

{

auto head = Util::to\_cstring(str);

grid.SetItemText(0, col, head);

++col;

}

}

void display() override

{

RECT rect;

grid.GetWindowRect(&rect);

for (int i = 0; i < lines.size(); ++i)

{

PrintDataInGrid<0, Container, NumCols>::Apply(i, lines, grid);

}

grid.SetColumnWidth(0, 100);

int width{};

for (int i = 0; i <= headers.size(); ++i)

{

grid.AutoSizeColumn(i); // skip vertical headers

width += grid.GetColumnWidth(i);

}

auto frame = grid.GetParentFrame();

width += GetSystemMetrics(SM\_CXVSCROLL);

grid.SetWindowPos(frame, 10, 10, width, rect.bottom - rect.top, SWP\_NOZORDER | SWP\_NOMOVE);

}

private:

template<int Col, typename FieldType, typename T>

static CString FormatCol(T& value, CJDGridCtrl& grid)

{

CString cs;

if constexpr ((std::is\_integral\_v<FieldType> || std::is\_floating\_point\_v<FieldType>) && !std::is\_same\_v<std::remove\_const\_t<FieldType>, bool>)

{

grid.m\_sortingFunctions[Col + 1] = Util::Comparison::Money;

if (ColonesCols::template found<Col + 1>())

{

Util::Colones c(value);

cs = format(c);

}

else if (DollarsCols::template found<Col + 1>())

{

Util::Dolares d(value);

cs = format(d);

}

else

{

cs = format(value);

}

}

else

{

grid.m\_sortingFunctions[Col + 1] = Util::Comparison::Text;

cs = format(value);

}

return cs;

}

template<int Col, typename Container, unsigned NumCols>

struct PrintDataInGrid

{

static void Apply(int row, const Container& z, CJDGridCtrl& grid)

{

CString cs;

using FieldType = std::remove\_reference\_t<decltype(std::get<Col>(z[row]))>;

FieldType\* pT = nullptr;

if constexpr (std::is\_same\_v<FieldType, const std::unique\_ptr<double>>)

{

auto&& value = extractValue(std::move(std::get<Col>(z[row])));

cs = FormatCol<Col, double>(value, grid);

}

else

{

auto&& value = std::get<Col>(z[row]);

cs = FormatCol<Col, FieldType>(value, grid);

}

grid.SetItemText(row + 1, Col + 1, cs);

PrintDataInGrid<Col + 1, Container, NumCols>::Apply(row, z, grid);

}

};

template<typename Container>

struct PrintDataInGrid<NumCols, Container, NumCols>

{

static void Apply(int row, const Container& z, CJDGridCtrl& grid)

{

}

};

};

Preparing a grid for display is done like this:

template <typename T>

void RendimientosTableDlg::InitializeGridRendimientos(const T& whereClause)

{

auto rows = Storage::getStorage().select(columns(

alias\_column<als\_r>(&Rendimiento::id),

alias\_column<als\_f>(&Fondo::nombre),

alias\_column<als\_r>(&Rendimiento::fecha),

alias\_column<als\_r>(&Rendimiento::rendimiento\_unitario)),

inner\_join<als\_f>(on(c(alias\_column<als\_r>(&Rendimiento::fkey\_fondo)) == alias\_column<als\_f>(&Fondo::id))),

where(whereClause),

multi\_order\_by(order\_by(alias\_column<als\_r>(&Rendimiento::fkey\_fondo)), order\_by(alias\_column<als\_r>(&Rendimiento::fecha)).desc()));

std::vector<std::string> headers{ "ID RENDIMIENTO", "FONDO ", "FECHA", "REND UNITARIO"};

m\_rendimientos\_displayer.reset(new JoinedGridDisplayer<decltype(rows[0]), IntegerList<>, IntegerList<4>>(m\_rendimientos\_grid, std::move(rows), std::move(headers)));

m\_rendimientos\_displayer->display();

}

Note moving of data to increase performance and avoid unnecessary copying (e.g. the rows and headers vectors are moved by using *std::move*). The grid displayer is finally told to display itself which is a polymorphic call from the *IDisplayer* interface.

## The RefIntegrityManager component

Implements the DB safety checks before deletion or insert/update. Works in close connection with RecordLinks and RecordLinks with Connections component.

void throwIfcannotDelete(Table const& record) // may throw std::exception

{

!RecordLinks::has\_links(record);

}

void throwIfcannotInsertUpdate(Table const& record) // same

{

RecordLinks::foreignKeysExist(record);

}

The safe update methods are described in the BoxContents component.

## The Data.h component

Implements the Storage\_Impl class which implements the functionality required by the Storage class. Both declarations are:

class Storage\_Impl

{

private:

Storage\_Impl() = delete; // prohibit instantiation

static auto& get\_storage();

static auto& get\_old\_storage();

static auto& get\_new\_storage();

static void copy\_old\_to\_new();

friend class Storage;

static constexpr const char\* db\_name{ "C:\\Users\\juan\_\\OneDrive\\Finances\\Investments.sqlite" };

};

class Storage

{

public:

using Storage\_t = decltype(Storage\_Impl::get\_storage());

Storage();

static void initialize();

static Storage\_t& getStorage() { return Storage\_Impl::get\_storage(); }

static void fill\_db\_with\_test\_data();

static void empty\_database();

static void upgrade\_database();

static void backup\_db();

};

It contains the object-relational mapping in class operation Storage::getStorage().

inline auto& Storage\_Impl::get\_storage()

{

static auto storage =

make\_storage(db\_name,

make\_table("Fondos",

make\_column("id\_fondo", &Fondo::id, autoincrement(), primary\_key()),

make\_column("nombre", &Fondo::nombre),

make\_column("abrev", &Fondo::abreviacion),

make\_column("tipo\_cupon", &Fondo::tipo\_cupon)),

make\_table("Inversiones",

make\_column("id\_inversion", &Inversion::id, autoincrement(), primary\_key()),

make\_column("fkey\_fondo", &Inversion::fkey\_fondo),

make\_column("begin\_date", &Inversion::beginning\_date),

make\_column("num\_participaciones", &Inversion::num\_participaciones),

foreign\_key(&Inversion::fkey\_fondo).references(&Fondo::id)),

make\_table("Rendimientos",

make\_column("id\_rendimiento", &Rendimiento::id, autoincrement(), primary\_key()),

make\_column("fecha", &Rendimiento::fecha),

make\_column("rend\_unitario", &Rendimiento::rendimiento\_unitario),

make\_column("fkey\_fondo", &Rendimiento::fkey\_fondo),

foreign\_key(&Rendimiento::fkey\_fondo).references(&Fondo::id)));

storage.sync\_schema(true);

return storage;

}

## The data.cpp component

This component implements the Storage and Storage\_Impl classes and provides definitions for persistent classes’ methods such as Fondo::num\_participaciones\_al which is a nice combination of object orientation and SQL programming via the DSL:

int Fondo::num\_participaciones\_al(std::chrono::sys\_days fecha) const noexcept

{

sys\_days when = fecha;

auto suma\_participaciones = storage.select(sum(&Inversion::num\_participaciones), where(c(&Inversion::beginning\_date) <= when and (c(&Inversion::fkey\_fondo) == this->id)), group\_by(&Inversion::fkey\_fondo));

if (suma\_participaciones.empty()) { return 0; }

auto suma = \*std::move(suma\_participaciones[0]);

auto sum\_participaciones = static\_cast<int>(suma);

return sum\_participaciones;

}

## The RecordLinks component

Summarizes all FKs-PKs links in the DB in a declarative format providing support before attempted deletion of a record (has\_links) and attempted insertion or updating of a record (foreignKeysExist). In both cases it asks the DB to check if all links will be kept upon the database operation and advises via std::exception if otherwise. Note how we declare persistent classes with their PKs and FKs. This is indeed descriptive and high level; it is actually a good way to have a complete picture of the dependencies structure of the DB all declared in one place. It of course depends on the mapping done in component data.h. This information exists in the **sqlite\_orm** library but its complexity motivated defining a simpler approach (the RecordLinks component).

Originally, the RecordLinks component was repetitive, low level and error prone, as you can verify here:

bool RecordLinks::has\_links(const Fondo& fondo)

{

int count[2]; // verify if there are Rendimientos pointing to fondo.id

count [0] = storage.count<Rendimiento>(where(is\_equal(&Rendimiento::fkey\_fondo, fondo.id))); // verify if there are Inversiones pointing to fondo.id

count [1] = storage.count<Inversion>(where(is\_equal(&Inversion::fkey\_fondo, fondo.id)));

return count[0] || count[1];

}

bool RecordLinks::foreignKeysExist(const Inversion& inv)

{

// verify inv.fkey\_fondo has a corresponding PK in Fondo

int count = storage.count<Fondo>(where(is\_equal(&Fondo::id, inv.fkey\_fondo)));

return count > 0;

}

Although the code was written in SQL, its intention was not clear. The new RecordLinks has a much higher level of abstraction and expressiveness:

#include "PersistentClasses.h"

#include "..\ORM\_Extensions/Connections.h"

class RecordLinks

{

// TableKeys

using FondoPK = TableKey<Fondo, &Fondo::id>;

using InversionFondoFK = TableKey<Inversion, &Inversion::fkey\_fondo>;

using RendimientoFondoFK = TableKey<Rendimiento, &Rendimiento::fkey\_fondo>;

// Fondo structure

struct Fondos

{

using PKDependents = PKDependencies<FondoPK, InversionFondoFK, RendimientoFondoFK>;

};

// Rendimiento structure

struct Rendimientos

{

using FKDependents = typename FKDependencies<RendimientoFondoFK, Fondos::PKDependents>::construct::result;

};

// Inversion structure

struct Inversiones

{

using FKDependents = typename FKDependencies<InversionFondoFK, Fondos::PKDependents>::construct::result;

};

public:

static bool has\_links(const Fondo& fondo)

{

return Fondos::PKDependents::has\_links(fondo);

}

static bool has\_links(const Inversion& inversion) { return false; }

static bool has\_links(const Rendimiento& rendimiento) { return false; }

static bool foreignKeysExist(const Fondo& fondo) { return true; }

static bool foreignKeysExist(const Inversion& inversion) { return Inversiones::FKDependents::foreignKeysExist(inversion); }

static bool foreignKeysExist(const Rendimiento& rendimiento) { return Rendimientos::FKDependents::foreignKeysExist(rendimiento); }

};

## The Connections component

This component’s API is composed of 4 classes: TableKey, PKDependencies, FKDependencies, TableConnectionList. Additionally, there are several implementation helpers.

### Dependencies

Types with the format *PKTable*PK like FondoPK define a TableKey, which contains information about the table associated with the persistent class Fondo and its PK. Types with the format *FKTablePKTable*FK like InversionFondoFK and RendimientoFondoFK are also TableKeys but contain information about the corresponding persistent structs and each of their FKs. Since they have only one FK each, then there is only one FK TableKey per class. Otherwise, we would have to define one for each additional FK. TableKey is defined like this:

// T is a persistent class

// K is its primary key

template <typename T, int T::\* K> requires (is\_persistent<T>::value)

struct TableKey

{

using Table = T;

using KeyType = decltype(K);

static constexpr KeyType Key = K;

static void setRecord(const Table\* const record)

{

tableData = record;

}

static int getKeyValue()

{

if (tableData == nullptr) throw std::exception{ "tableData is null" };

return tableData->\*Key;

}

private:

inline static const Table\* tableData = nullptr;

};

PKDependencies defines the PK dependencies: the only table that has dependencies is the one associated with Fondo and its PK is accessed from the tables associated with Inversion and Rendimiento, thus we define a PKDependencies inside the Fondos struct which we use as a “namespace”[[5]](#footnote-5)

struct Fondos

{

using PKDependents = PKDependencies<FondoPK, InversionFondoFK, RendimientoFondoFK>;

};

PKDependencies is defined like:

// T and RefBy are TableKeys

// Target is special, has primary key

// The TableKeys in RefBy have foreign keys

template <typename T, typename ...RefBy> requires ( is\_tableKey<T>::value && (is\_tableKey<RefBy>::value && ...))

struct PKDependencies

{

public:

using Target = T;

static std::tuple<RefBy...> reference\_list;

static bool has\_links(const typename Target::Table& targetRec)

{

Target::setRecord(&targetRec);

auto pk = Target::getKeyValue();

if( pk == -1)

{

std::ostringstream ss;

ss << "Registro de " << Storage::getStorage().tablename<typename Target::Table>() << " no está almacenado";

throw std::exception(ss.str().c\_str());

}

constexpr size\_t size = std::tuple\_size\_v<decltype(reference\_list)>;

bool has = has\_links<size>();

if(has)

{

std::ostringstream ss;

ss << "Registro de " << Storage::getStorage().tablename<typename Target::Table>() << " tiene dependientes";

throw std::exception(ss.str().c\_str());

}

return has;

}

private:

template<size\_t index>

static bool has\_links()

{

using namespace sqlite\_orm;

auto& storage = Storage::getStorage();

// is a TableKey

using DependentClass = std::tuple\_element\_t<index - 1, decltype(reference\_list)>;

bool has = storage.count<DependentClass::Table>(where(is\_equal(DependentClass::Key, Target::getKeyValue()))) > 0;

return has || has\_links<index - 1>();

}

template<>

static bool has\_links<0>() { return false; }

};

FKDependencies is defined for all tables that have FKs. This class extracts the type information about FK-PK defined in PKDependencies instantiations (in our example accessed by Fondos::PKDependents); this avoids having to write the relations once again which is always a bad programming practice. The class FKDependencies does a lot of type manipulation in order to provide each TableConnection implicit in its declaration. An example of a class with FKs is Rendimiento and so we define:

struct Rendimientos

{

using FKDependents = typename FKDependencies<RendimientoFondoFK, Fondos::PKDependents>::construct::result;

};

Notice how the class hides the building and the appending functionality and only exposes a construct struct which both starts the building and reports the TableConnectionList it creates. The class is defined like so:

// T is a TableKey, also known as T::Target or tableKey

// tblDefs are PKDependencies types

// tblDefs[x]::reference\_list is list of TableKeys

// we are going to search TableKey T in each tblDefs[x]::reference\_list

// for each found we will create a TableConnection<T, tblDefs[x]::Target>

// we add such connection to TableConnections list by appending

// (see struct append inside struct build and see how the list is "exported"

// in struct construct: see how construct triggers the building of the list

// and reports its final stage)

template<typename T, typename...tblDefs >

struct FKDependencies

{

using tableKey = T;

static std::tuple<tblDefs...> tblDefs\_list;

private:

template<typename Col>

struct build

{

template <typename Col, int N>

struct append

{

private:

using tableDef = std::tuple\_element\_t<N - 1, decltype(tblDefs\_list)>;

using tableKeys = decltype(tableDef::reference\_list);

static constexpr int index = TableKeyInTuple < tableKey, tableKeys>::index();

using typeToAdd = TableConnection<tableKey, typename tableDef::Target>;

using newCol = static\_if<Col, typeToAdd, index>::type;

public:

using result = typename append<newCol, N - 1>::result;

};

template<typename Col>

struct append<Col, 0>

{

using result = Col;

};

static constexpr int size = std::tuple\_size\_v<decltype(tblDefs\_list)>;

using result = typename append<Col, size>::result;

};

public:

struct construct

{

private:

using res = typename build<Loki::NullType>::result;

public:

using result = TableConnectionList<res>;

};

};

We can see encapsulation of implementation details in the type alias res in nested struct construct above.

The TableConnection class is also very clear:

// connection between 2 TableKey's, example:

// Dependent pair

// DepPair::Table = Rendimiento

// DepPair::Key = getKeyValue()

// Target pair

// TargetPair::Table = Fondo

// TargetPair::Key = &Fondo.id

template <typename DepPair, typename TargetPair> requires (is\_tableKey<DepPair>::value && is\_tableKey<TargetPair>::value)

struct TableConnection

{

using Dependent = DepPair;

using Target = TargetPair;

static bool foreignKeyExists(const typename Dependent::Table& dep)

{

Dependent::setRecord(&dep);

using namespace sqlite\_orm;

auto& storage = Storage::getStorage();

int count = storage.count<typename Target::Table>(where(is\_equal(Target::Key, Dependent::getKeyValue())));

return count > 0;

}

};

The TableConnectionList class is a list of TableCollections implemented via Loki::Typelist[[6]](#footnote-6)

// List of TableConnection instances

template <typename List>

struct TableConnectionList

{

using typeAt0 = typename Loki::TL::TypeAt<List, 0>::Result;

using DependentTable = typename typeAt0::Dependent::Table;

static bool foreignKeysExist(const DependentTable& dep)

{

constexpr size\_t size = Loki::TL::Length<List>::value;

bool exists = foreignKeyExists<size>(dep);

if (!exists)

{

std::ostringstream ss;

ss << "Registro de " << Storage::getStorage().tablename<DependentTable>() << " contiene dangling FKs";

throw std::exception(ss.str().c\_str());

}

return exists;

}

private:

template<size\_t index>

static bool foreignKeyExists(const DependentTable& dep)

{

// is a TableConnection

using Connection = typename Loki::TL::TypeAt<List, index - 1>::Result;

bool exists = Connection::foreignKeyExists(dep);

return exists && foreignKeyExists<index - 1>(dep);

}

template<>

static bool foreignKeyExists<0>(const DependentTable& dep)

{

return true;

}

};

template<>

struct TableConnectionList<Loki::NullType>

{

static bool foreignKeysExist(...) { return true; }

};

# Conclusions

As demonstrated by Yevgeniy Zakharov’s wonderful library sqlite\_orm, it is possible to write a Domain Specific Language for SQL in C++. Mixing C++ abstraction level control with this SQL DSL and a set of components for rapid application development it has been shown that modern C++ is indeed a very expressive and powerful language suitable for data intensive applications. A general pattern for applying the described framework has shown this to be the case.

Combining SQL with template metaprogramming techniques, yields a very high level of abstraction where the problem domain is represented syntactically as the development vocabulary. Object-orientation concepts like polymorphism in the template implementation and the explicit use of C++20 concepts to restrain template instantiation prove to be a rich arena for application development not restricted to C++’s very well-known success in infrastructure programming.

Operator overloading also enables this high level of abstraction. The presented framework includes not only database functionality but GUI components which support persistency in a natural and intuitive way. We have not found necessary to restrict the use of SQL to a certain part of an application. Rather because of its support for domain vocabulary, we have no problem using it mixed with GUI programming. There is therefore no layers proposed as is usual in persistent frameworks. Rather, we want the whole program to speak SQL using the SQL DSL library anywhere needed. Factoring of code into classes is of course at the essence of any true object-oriented development and more so when using a persistence one to one definition of classes and tables, because such structure is usually very stable in the long run. The need for careful domain relational analysis cannot be over emphasized as it defines the central skeleton of any data intensive application.

# Future development

There remains a large area of exploration to see how the proposed framework behaves in much larger applications. The use of complex record types composed of several normalized tables is a very interesting design challenge and we plan to investigate how better to approach such classes. Also, we have not touched many parts of the SQL DSL, of which the use of triggers is probably the most interesting. Programming triggers in C++ remains a new area of research for me… Another area to consider is the creation of code generating utilities that would enforce the framework’s best practices when dealing with basic maintenance dialog boxes for normalized tables that naturally follow a well-defined pattern.

Further collaboration with Mr. Zakharov is an exciting activity that has proven to be extremely rewarding and educational. We propose to continue to learn from him and from his excellent work in template metaprogramming of DSL in the SQL domain. We feel we have only scratched the surface of the DSL API and there is much to be discovered, now that we feel we have established a framework for rapid application development for data intensive projects.

# Bibliography

[S,2013] Bjarne Stroustrup. *The C++ Programming Language, 4th edition*. Addison Wesley 2013.

[CPPTMP,2005] David Abrahams, Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison Wesley, 2005

[MCPPD,2001] Andrei Alexandrescu. *Modern C++ Design - Generic Programming and Design Patterns Applied*. Addison Wesley 2001

[DPMCPP,2018] Dmitri Nesteruk. *Design Patterns in Modern C++ - Reusable Approaches for Object-Oriented Software Design*. APress, 2018

[CPP20, 2021] Rainer Grimm. *C++ 20 Get the Details*. Leanpub book, 2020-2021

[Zakharov,2022] Yevgeniy Zakharov. The sqlite\_orm library tutorial. [fnc12/sqlite\_orm: ❤️ SQLite ORM light header only library for modern C++ (github.com)](https://github.com/fnc12/sqlite_orm)

[DMCPP,2016] Peter Gottsching. *Discovering Modern C++ - An Intensive Course for Scientists, Engineers, and Programmers*. Addison Wesley, 2016.

[Unruh,1994] Erwin Unruh. “Prime number computation”. ANSI X3J16-94-0075/ISO WG21-462. 1994.

[Czarnecki,2008] Krzysztof Czarnecki. “Template Metaprogramming in C++”. <https://courses.cs.vt.edu/~cs5984/2008-spring-oosl/documents/Lecture6_Template_Metaprogramming.pdf>

1. This actually generates code [↑](#footnote-ref-1)
2. Restrictions on valid values can be done at the DB level with the use of triggers which are also supported by the **sqlite\_orm** library – this frees the dialog code from having to verify each restriction explicitly but also implies that these exceptions will have to be caught by code with DB updating operations. We believe this is a worthwhile tradeoff. [↑](#footnote-ref-2)
3. Microsoft Foundation Classes: a C++ framework for object-oriented development for the Windows platform. [↑](#footnote-ref-3)
4. Only some very well-defined utility classes are held in modules; we are not ready to apply them to the whole development. It is not yet clear how to best mix ordinary includes with the new modules of C++ 20. [↑](#footnote-ref-4)
5. because we cannot define syntactic namespaces inside a struct/class, using an embedded struct is the preferred workaround: it gives us the feeling and syntax of a namespace [↑](#footnote-ref-5)
6. See Chapter 3 of [MCPPD,2001] by template metaprogramming pioneer Andrei Alexandrescu [↑](#footnote-ref-6)