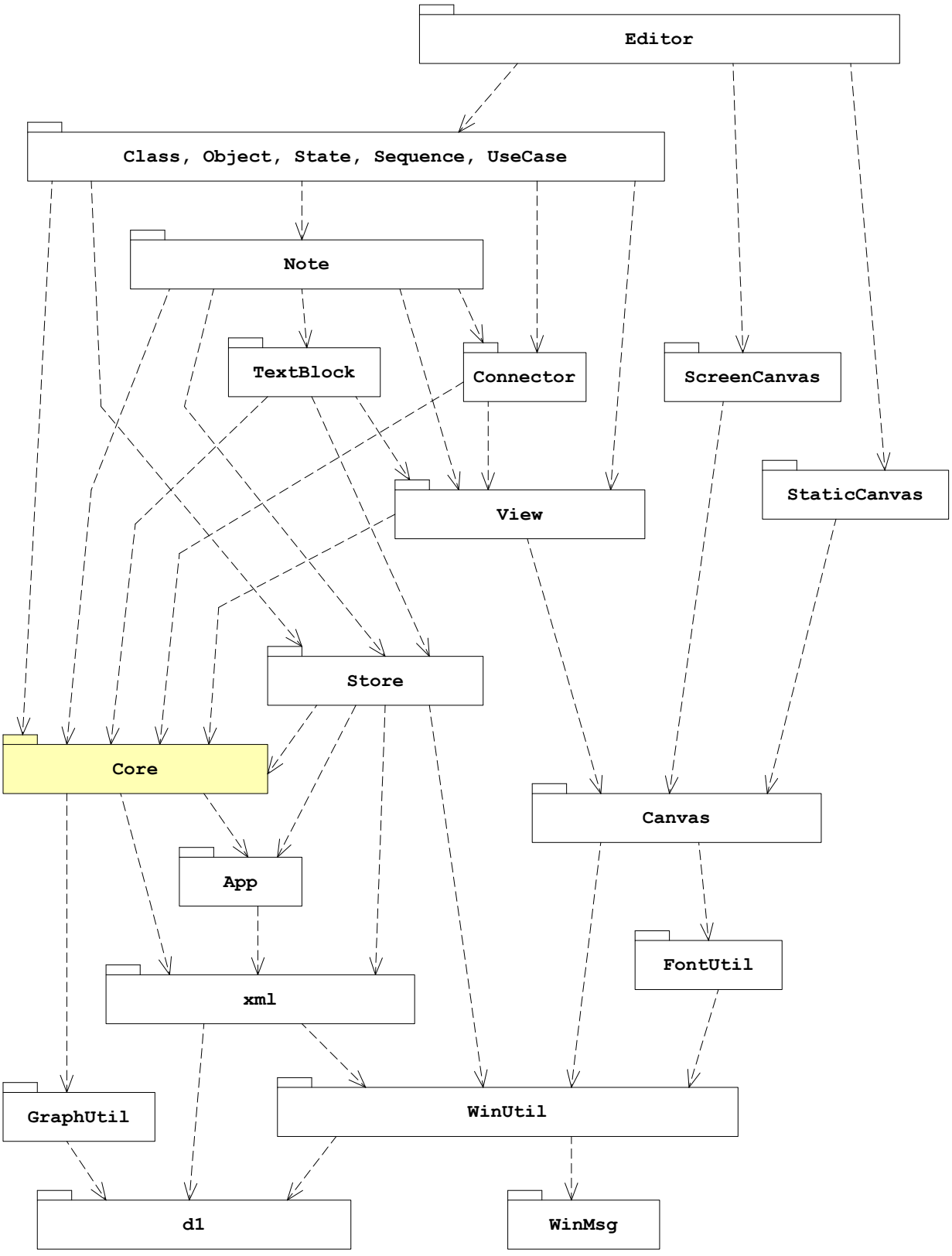
*Adrian Bühlmann, adrian@cadifra.com*

# Converting a C++ application to modules

We have converted the C++ sources of our Cadifra UML Editor[1] from using header files to C++ 20 modules.




1 https://www.cadifra.com

The sources are organized into ~40 packages. Each package uses a C++ namespace with the same name as the package. In the drawing above, you can see the most important packages with their dependencies. The picture was drawn using our UML Editor. Some less important packages have been omitted.

We had roughly one major class per `*.h/*.cpp` pair. We used *forward declarations* for classes to minimize dependencies between packages, following the guideline by Herb Sutter[2]:

> *Guideline: Never #include a header when a forward declaration will suffice.*

## First attempt

In a first naive attempt, I converted nearly every header file to an interface module (`.ixx`), with the implementation module in the `.cpp` file.

Then I had a problem with the forward declarations. For example, for the interface `View.IShiftControl`:

```
export module View.IShiftControl;

import Base.Forward;

import Core.Forward;

import d1.Point;
import d1.Shared;


namespace View
{

export class IShiftControl: public d1::Shared
{
    Core::IElement& itsElement;

public:
    IShiftControl(Core::IElement& m):
        itsElement{ m }
    {
    }

    auto Element() const -> Core::IElement& { return itsElement; }

    virtual void Shift(Core::Env&, const Base::ShiftVector&,
        const d1::fPoint& mouse_pos) = 0;

    virtual void Finalize(Core::Env&) = 0;

    IShiftControl(const IShiftControl&) = delete;
    IShiftControl& operator=(const IShiftControl&) = delete;
};

}
```

2 https://herbsutter.com/2013/08/19/gotw-7a-solution-minimizing-compile-time-dependencies-part-1/

I imported `Core.Forward`, which contains forward declarations for all classes in the package `Core`:

```
export module Core.Forward;

export namespace Core
{
class CopyRegistry;
class ElementSet;
class Env;
class ExtendSelectionParam;
class IClub;
class IDiagram;
class IDirtyMarker;
class IDirtyStateObserver;
...
}
```

The problem with this is, that according to the C++ 20 language specification, a name, which is declared in a module, is *attached to that module* and must thus be defined in that same module.

So, this didn't work. But there is a – partial – solution for this.

## Module partitions

I changed the line

```
export module Core.Forward;
```

to

```
export module Core:Forward;
```

thus replacing the dot (`.`) in the middle with a colon (`:`).

This now defines partition `Forward` of module `Core`.

So now, we have a bigger module named `Core`, which is separated (partitioned) into a number of partitions.

Partitions are just a means for splitting the source files of an interface module (`Core`).

The same applies to, for example, `View.IShiftControl`, which must be changed to `View:IShiftControl` accordingly.

The fun part now is, that all declarations in every partition of `Core` are *attached* to module `Core`, not to a partition module.

Which in turn means, we can forward declare classes *inside* `Core`.

To glue the partitions together, I created a file `Core/Module.ixx`, which contains:

```
export module Core;

export import :Contains;
export import :CopyRegistry;
export import :Elements;
export import :ElementSet;
export import :Env;
export import :Exceptions;
export import :ExtendSelectionParam;
export import :Finalizer;
export import :FollowUpJob;
export import :IClub;
export import :IDiagram;
export import :IDirtyMarker;
export import :IDirtyStateObserver;
export import :IDocumentChangeObserver;
export import :IElement;
export import :IElementPtr;
export import :IFilter;
export import :IGrid;

(etc.)
```

Then, wherever something from `Core` is needed somewhere, we have to

```
import Core;
```

Note that if a class *outside* of `Core` is used by reference (or a pointer), we now have to import `Core` as well, since we cannot forward declare a class in a module, which is defined in a different module. We also cannot use `Core:Forward` outside of `Core`.

Partitions can only be used *inside* a module anyway and may only be exported by the primary interface of the module (`Core/Module.ixx` in our case).

Inside module `Core`, we can import the `Forward` partition with

```
import :Forward;
```

which imports the forward declarations of the classes of module `Core` into the current partition.

## Final remark

For the conversion to modules, no refactorings of our design were needed. The classes were ready for the conversion.