Design of VDisp Software

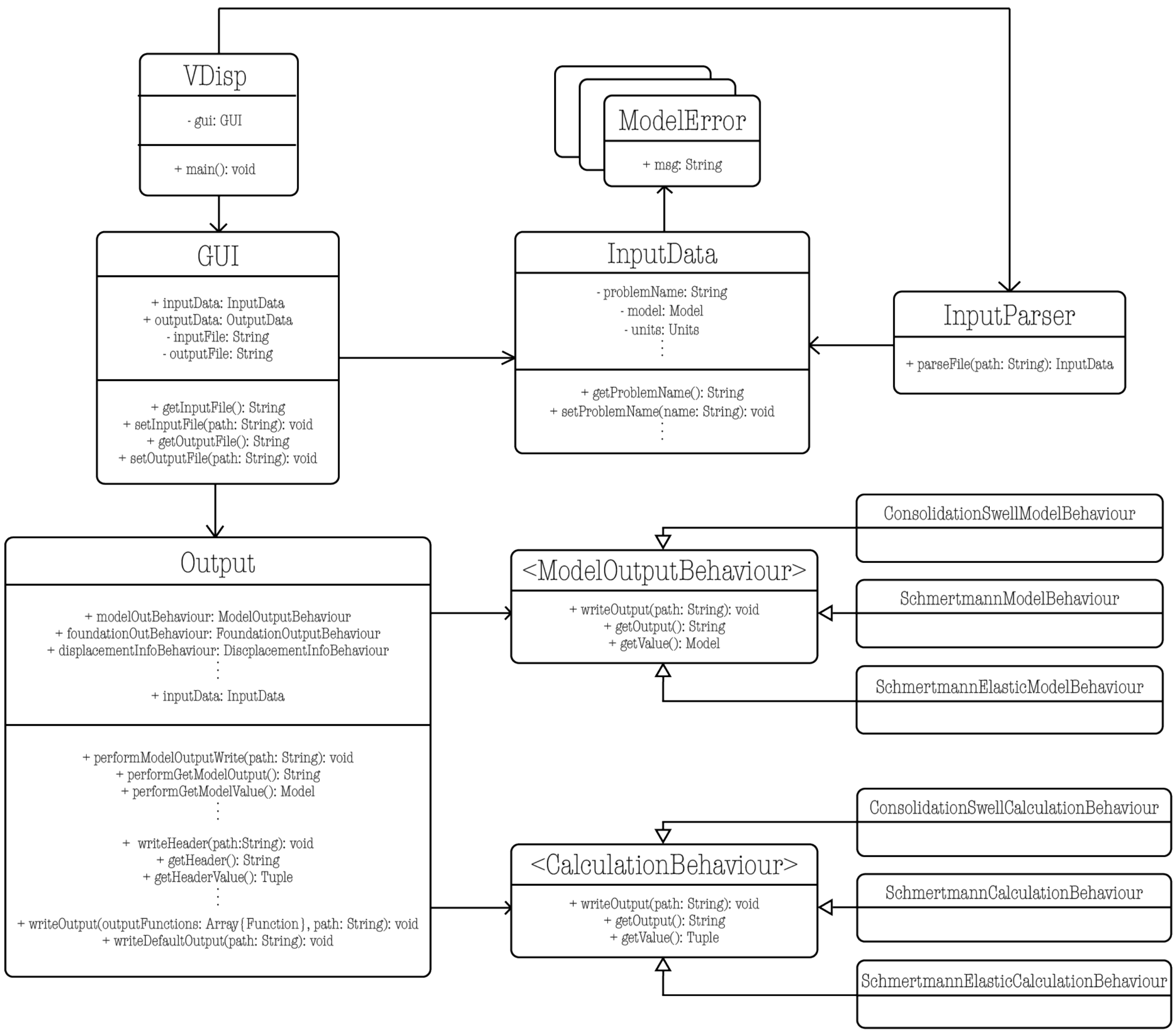
Emil Soleymani, Dr. Spencer Smith
McMaster University

September 11, 2022

The **VDisp** software was carefully designed to adhere to the *SOLID* software design principles. This document aims to outline the initial proposed design (which will be called the ideal design), the actual implemented design, and the limitations of the Julia programming language that lead to this drastic change.

Ideal Design

This section aims to describe the *ideal* design which was drafted for the **VDisp** software.



The diagram above has been condensed. *InputData* uses many Custom Exception classes which help identify specific problems in the input file format allowing for helpful and descriptive messages to be given to the user. Furthermore, the **FoundationOutputBehaviour**, **DisplacementInfoBehaviour**, **ForcePointBehaviour** and **EquilibriumInfoBehaviour** interfaces (and the classes that implement them) have been left out of the diagram.

Examining the **ModelOutputBehaviour** interface is sufficient to understand the implementation of the excluded interfaces, while the **CalculationOutputBehaviour** is more complex than the other interfaces. The classes that implement **CalculationOutputBehaviour** are responsible for making method specific calculations, and return a set of values that is dependant on the method itself. This is why the **CalculationOutputBehaviour** *getValue()* function is said to return a **Tuple**. This **Tuple** contains different info based on the specified model, and classes that access this **Tuple** must be prepared to parse it.

The design pattern of the *Output* class and the interfaces it uses (all the interfaces that end in "*Behaviour*") is inspired by the *Duck example* in the opening chapter of this textbook on design patterns.

Actual Design

This section aims to describe the actual design which was implemented in the **VDisp** software. It strays from the ideal design due to some limitations outlined in the Julia Limitations section.

Julia Limitations

This section aims to describe the limitations imposed upon us by Julia during development of the *VDisp* software. These limitations mainly arose from the limited object-oriented capabilities of Julia — mainly importing structs from other modules.

Importing Structs and Enums Between Modules

Suppose you wanted to define a struct that will hold all the data the user inputs into the program. This is exactly what the *InputData* struct aims to do. It is found in the *InputParser.jl* module. Suppose there is a function within *InputParser.jl* that takes an argument which must be an instance of *InputData*:

```
function doSomething(inputData::InputData)
```

```
    # Do stuff
end
```

Now suppose you would like to use this function in another module, *MyModule.jl*. You would first have to import the *InputParser.jl* module, then create an instance of *InputData*. Then you can try calling the function, and would feel confident that the following code produces no errors:

```
using InputParser

inputData = InputData(...args)
doSomething(inputData)
```

However, you get an error that says something along the lines of:

```
MethodError:
No function found for doSomething(
    Main.MyModule.InputParser.InputData)

Closest Methods:
doSomething(Main.InputParser.InputData)
```

I'm guessing this happens because Julia "copies" over the *InputParser.jl* module into the *MyModule.jl* file at compilation time due to the *using InputParser* statement.

This behavior also caused many problems in the *VDisp* custom exceptions which are defined to catch various different errors in the input files. These custom exceptions are defined as structs, and a sample declaration is given as follows:

```
struct MyException <: Exception
    field1::Type
    field2::Type
    MyException(a::Type, b::Type) = new(a, b)
end
```

Suppose this struct is defined in the *InputParser.jl* module. Now suppose there is a function, *parse(file)*, in the same module that parses input files, and potentially throws *MyException* or other exceptions. If we call this function from another module, maybe *MyModule.jl*, we would again have to first import the *InputParser.jl* module, then call the *parse(file)* function. Since this function can throw errors, we will catch them to be safe. Say we wanted to print *foo* if a *MyException* was thrown, else print *bar*. One would expect the following code to get the job done:

```
using InputParser

try
    parse("file.dat")
catch e
    if isa(e, MyException)
        println("foo")
    else
        println("bar")
    end
end
```

However, even if *parse("file.dat")* does indeed throw a *MyException*, the code snippet above will still output *bar*. To further investigate what is going on here, try the following code:

```
using InputParser

try
    parse("file.dat")
catch e
    println(typeof(e))
end
```

This code produces the output *Main.MyModule.InputParser.MyException*. Thus, the original if statement condition, *isa(e, MyException)*, was not true since *Main.MyModule.InputParser.MyException* $\neq$ *Main.InputParser.MyException*.

VDisp does implement custom functions, so clearly there was a workaround. However, this was more of a "hack" — not a permanent, sustainable, scalable fix. There is an Enum called *ErrorId* defined in the *InputParser.jl* class (implemented using the *@enum* macro). Each custom defined exception has a corresponding *ErrorId*, which is stored in each struct's *id* field. Thus, the following code will achieve what we want:

```
using InputParser

try
    parse("file.dat")
catch e
    if Int(e.id) == Int(MyExceptionId)
        println("foo")
    else
        println("bar")
    end
end
```

This code is assuming *MyExceptionId* is properly defined in the *InputParser.jl* module (*@enum ErrorId MyExceptionId ...*), and that the *id* of every *MyException* is set to *MyExceptionId*. More careful readers may be wondering why the conditional was stated as *Int(e.id) == Int(MyExceptionId)*. Directly checking *isa(e.id, MyExceptionId)* results in the same issue that we began with. However, we can convert any instance of an enum to its corresponding integer value by wrapping it in the *Int()* function. Now we can catch specific runtime errors, and perform the necessary operations. Although omitted from the above code, it would be best practice to first check if the error *e* has a field called *id* in the first place. This would arise if you accidentally forgot to give a new error an *id* field, or an exception was raised that is not one of your custom defined exceptions.

QML.jl Package

Although the Julia community has grown significantly over the years, the options for 3rd party packages are still limited compared to that of more mature and ubiquitous languages like Python. This was especially problematic when it was time to choose a library to draw the graphics for the user interface. The best option was the *QML.jl* package, which used another package, *CxxWrap.jl*, to communicate desired Qt/QML functionality to its native language, C++. This package is developed and maintained by a single person, so naturally it cannot be as robust and complete as a package like *Tkinter* or *pygame*.

The biggest problem related to the *QML.jl* package came up when *VDisp* needed to execute a function contained in its Julia code on a certain UI event triggered in the QML markup. For example, if we wanted to run Julia code in a function called *saveFile()* when clicking on a button defined in a QML file, we would have to first “pass the function“ into QML by calling the *@qmlfunction saveFile* macro, then have a QML file as follows:

```
import import QtQuick 2.12
import org.julialang 1.0

Rectangle{
    /* Design */

    MouseArea {
        anchors.fill: parent
        onClicked: Julia.saveFile  // Call saveFile()
    }
}
```

Although this is the method found in the *QML.jl* documentation, and it was used in a demo UI created to test the package before choosing it for this project (see this GitHub repo), when trying to call functions from the UI the following error occurred:

```
ERROR: LoadError: Could not find module QML when looking up function
get_julia_call

Stacktrace:
[1] exec()
@ QML ~/.julia/packages/CxxWrap/ptbgM/src/CxxWrap.jl:619
```