Clinguin Project Document

2022-08-29 Clinguin

1 Project Summary

The *Clinguin* project can be seen as a language extension to a Logic-Program, which adds the functionality to create a User-Interface (UI). Capability wise the UI shall be capable of dynamic updates, i.e. not just a static display of a stable-model(s), but a program where one can interact with the logic program.
An example for this is the Sudoku problem. Using the Clinguin one is able to generate a Sudoku playing-field, where the field is interactive in a sense, that if one selects one of the possible values a single field may have, the program automatically updates the not selected values and fills them in, if only one possibility exists. This is similar to forward checking for Constraint-Satisfactory-Problems (CSPs).

2 Prototype

Susana Hahn has written a prototype, which implements the Sudoku example from above. The used programming language is Python, further Clingo libraries (like the Clingo-Object-Relational-Mapper (CLORM)) are and the UI-framework tkinter (https://docs.python.org/3/library/tkinter.html) are used. Syntactically the prototype provides an extension by the following keywords (predicates which are used as keywords/tokens):

- *window(<attribute-name>,<argument-key>,<argument-value>)* - Defines the general structure/-size/etc. of the displayed field
- *widget(<type-name>,<ui-id>,<master-id>)* - Adds a widget (ui-element/item) to the view. The accepted types are *frame* (container) and *menu* (dropdown menu). The *ui-id* should be unique and the *master-id* represents the enclosing item (for top-level frames this is the *window*).
- *geo(<ui-id>,<ui-layout>,<argument-key>,<argument-value>)* - Used for specifying the exact location of a widget. The *ui-id* is of a widget, the *ui-layout* can be chosen from a tuple of *(grid, place, pack)*, the *argument-key* is used for specifying what one wants to set (e.g. row) and the *argument-value* is the corresponding value.
- *config(<ui-id>,<argument-key>,<argument-value>)* - The *ui-id* is of the corresponding widget, the *argument-key* is the property to set (e.g. font width or some color) and the *argument-value* is the corresponding value to the key.
- *opt(<ui-id>,<value>,<original-state>)* - This is used for the dynamic updates of the gui. The *ui-id* is for the corresponding widget. In the prototype the stable matches are computed via the *brave* option, and for each possible value the *original-state* can have, also one *opt* is generated. This is used for computing what values a single sudoku field may have.

In general the program works like this: The program scans the logic-program for the syntactic elements above, and then creates the ui with the elements (function *create_layout()*, uses cautious reasoning). Then the dropdown-menus (i.e. the Sudoku cells) are filled, with the function *update_options()* (uses brave reasoning). Here for each such dropdown-menu a callback function is added, which when clicked adds an assumption, that a certain value must be used for the sudoku field. Then the process repeats itself.

3 Architecture Proposals

3.1 Initial Thoughts

The architecture of Clinguin should be flexible in a sense, that it should be compatible with several UI-Libraries/-frameworks. Further it should be efficient (which is not handled at this point, but e.g. incremental updates can be made instead of full updates).
The need for flexibility leads to the conclusion, that the program may not directly construct the *tkinter* (or some other frameworks) window, but must create an internal representation first (See Figure 1):

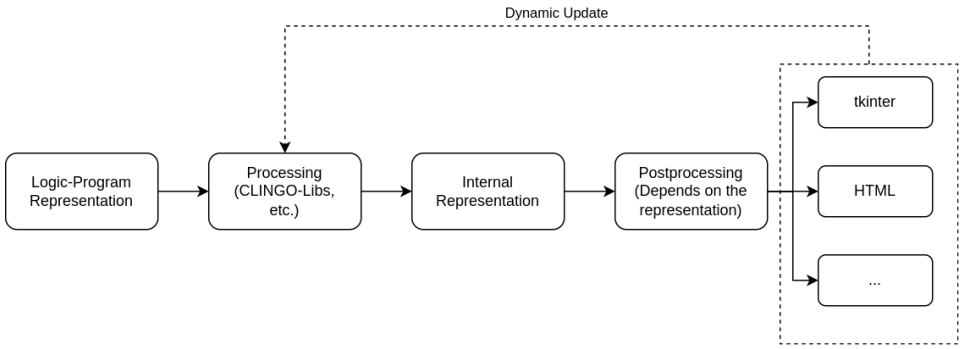

Figure 1: Informal idea of the workings of the program

Therefore for the construction of the program a few things have to be defined/thought about:

1. Logic-Program Representation
   - Syntax of language extension
   - Initial starting point → syntax of prototype
   - In general the syntax should be defined in a way that helps understanding of the program
2. Processing
   - Produces internal representation from the logic program
   - Main thing: Efficiency
   - Other things: Libraries to use (depends on selected internal representation), etc.
3. Internal Representation
   - **Very important to think about now**: Many different options exist, like *XML*, *Json*, *Class-Hierarchy*, *HTML*, ...

- Different representations have different pros/cons:
  - XML - Pro: Clearly defined structure, portable format. Cons: Format needs to be parsed again before it can be used (efficieny loss), postprocessing will be pretty complex
  - Json - Pro: For hierarchical GUI tools postprocessing will not be that much, portable format. Cons: Format needs to be parsed again, for non hierarchical GUI tools postprocessing will be quite complex
  - Class-Hierarchy - Pro: Efficiency, for hierarchical GUI tools postprocessing will be relatively easy. Cons: Non portable format (one is limited to Python represetation), for non hierarchical GUI tools postprocessing will be quite complex.
  - HTML - Pro: Direct representation of GUI elements, i.e. one can directly view the html, otherwise similar to XML. Cons: For other views than HTML postprocessing will be pretty complex

4. Postprocessing
   - Depends entirely on the chosen internal representation and chosen GUI-framework
   - Could be implemented as a *1:1* match between internal representation and GUI-framework, which would lead to a modularization of the Clinguin
   - Efficiency will be important here...
5. Initial GUI-Framework to use (tkinter, web/HTML, other)

3.2 Monolithic Custom

Hard-Coded-Monolith: The idea is to think of an architecture, which is a monolith. The architecture should be designed in a way that fulfills most requirements. One requirement which would be hard to fulfill is the Json/Independability requirement, i.e. that the gui may be switched in the future.

Json-Monolith: It is also possible to implement it with Json as intermediary representation, to achive a seperation between gui and the rest. This approach is essentially the same as the Client-Server approach (be sure to look at it first in Section 3.4), just that for the message passing no server is needed, but is instead done via function calls or events. With this approach one needs to implement a lot from the Client-Server part, just without setting up a server. The downside of this approach is, that one can only use gui-frameworks from the selected programming language (i.e. Python).

I didn't explore on these ideas much further, but it should definitely be possible, if one drops the independability/modularization.

3.3 Monolithic Model-View-Controller (MVC)

As the MVC pattern is a widely spread Gui-Architectural pattern it makes sense to consider it. In Figure 2 the general idea of MVC is depicted, where normally there exists one such MVC-triad per GUI element, e.g. one for the top level gui element, which contains "children", like menu-bar,

grid-controller, etc. (as depicted in Figure 3).

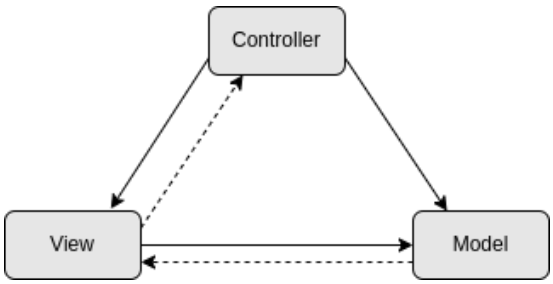

Figure 2: General MVC idea

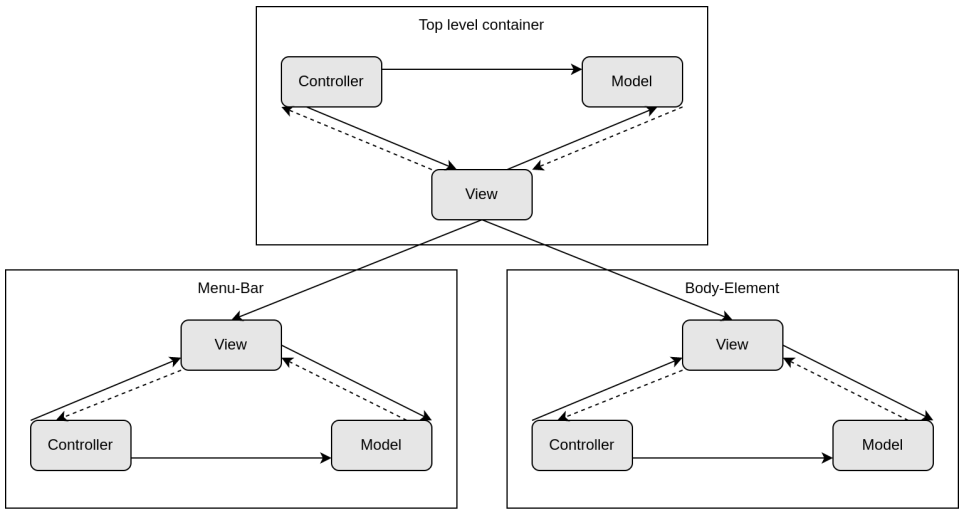

Figure 3: MVC Hierarchy

For Clinguin this architecture open two possible tracks of implementation, where it seems to be the case that for both tracks it would be pretty hard to get the callbacks right while using an independent representation like Json.

My main reason of concern in both cases is how would the notification process (callbacks) work together with Json?
As far as I know, when using MVC one ”hard-codes” the View, the Controller and the Model, each in a seperate file, where then a framework (like Windows-Presentation-Foundation (WPF) for C# - which is actually the MVVM principle, but pretty similar...) connects the files.
On the other hand, when one uses Json as a means for the intermediate representation, one normally uses some kind of Message-Oriented-Architecture (MOA), like a simple Client-Server system.

- Implementation of Clinguin as a single MVC triad
  - Implementing this in a way, that multiple elements are handled in one MVC should violate the MVC design principle
  - Normally in MVC the View should be static in a sense, that the elements of a view doesn't change, just the content - but here this would be violated

  - When using Json for data transfer, I can't imagine how the callbacks should work

- Implementation of Clinguin as a MVC hierarchy
  - Either as a code generator (i.e. a program which generates the python code, should fulfill the MVC design principle, but seems to be hard to implement)
  - Or implement it just via OOP - but which would still be hard and it would even be harder to get the callbacks right.
  - When using Json for data transfer, I can't imagine how the callbacks should work

3.4 Version 0: Simple Message based approach (Client-Server)

The next option to look at is the Client-Server approach. The main point for this architecture is the flexibility, that one can use the Json data as an intermediate representation and use whatever gui one finds suitable. Further the architecture is straightforward (also the callbacks).
The server architecture could be implemented as a normal layered-architecture, where the data layer is responsible for the communication with the underlying logical-data-structrure (i.e. CLORM), the logic layer contains some business logic (might actually not be necessary) and the presentation layer handles incoming requests, processes them (i.e. forwarding to the logic/data-layers) and then formats the response as a Json.

The Client just gets the Json and constructs with this information a gui. If a user clicks on a field a request is sent to the server, which processes it, computes the stable models, etc. . Implementation wise the client can be constructed in a variety of different ways, a straightforward one would be that the Json contains enough information that one may construct a gui with possible actions and if an action occurs, sends it back. The client may be written in any programming language.

Another pro: When one uses a *REST*-API, then One drawback with this approach is, that the development time might be higher than with a monolithic approach, but definitely still acceptable.

One could use the following libraries for this:

- Server
  - FastAPI - A REST framework which is similar to JavaSpringBoot or NestJS, released in 2018, high performance and seems to be maintained regularly
  - Flask - The standard tool for developing REST Api's in Python (syntactically similar to NestJS and Java Spring Boot). Seems to be not that great in terms of performance
- Client
  - api-client - seems to be exactly what we need

Further one can package both together, that from the outside it seems like, it is just one program, although they are two in fact. In terms of endpoints/APIs only three are proposed:

- GET: "/health" - For startup process, check if server is running
- GET: "/" - Get Json data to display
- POST: "/" - Send server, what has been clicked, etc.

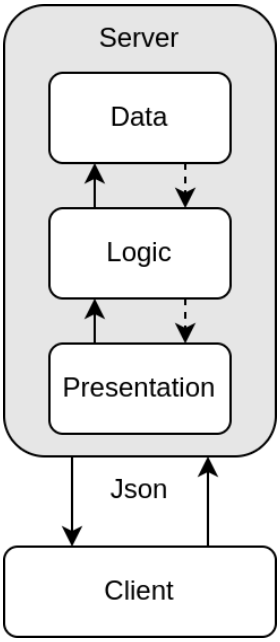

Figure 4: Cient-Server approach

3.5 Version 1: Simple Message based approach (Client-Server)

Based on the *Version 0* of the simple message based approach, the new approach (besides the flexibility granted by the intermediate Json format) addresses two other important tasks:

- Making the server-solver as flexible as possible, that it can be switched out/in.
- A concept structure of a *tkinter* frontend.

The figure 5 displays the advanced architecture on a high level, so one gets an idea how this works:

The idea for the backend is now the following: The presentation layer consists of the API-endpoints, which just consists of two:

- *call* - POST: "/" - Solver function call.
- *health* - GET: "/health" - For startup process, check if server is running.

When the *call* endpoint receives a request, it looks at it and parses it (a request should look like a function call, so e.g. *assume(sudoku(1,1,1))*). The function name must be a name of a function in the solver (e.g. *assume*). The other payload may be used as one wishes.
Therefore the next step is to call the specified function in the solver, which processes the request and returns something (e.g. for the function *solve()* this could be the Json representation of the whole GUI). For the conversion there should exist a Json-Converter which should be static in the sense, that every solver can use it easily.
As the previous line suggests, there will be many solvers. A user may specify a solver as one wants. The location of the solver (i.e. the module name in Python) can be specified as a command-line-argument (e.g. *$ clinguin –solver special-solver-1,special-solver-2 1.lp 2.lp 3.lp*)

The first UML diagram depicts the use with the standard solver, the second one depicts a use with a special solver:

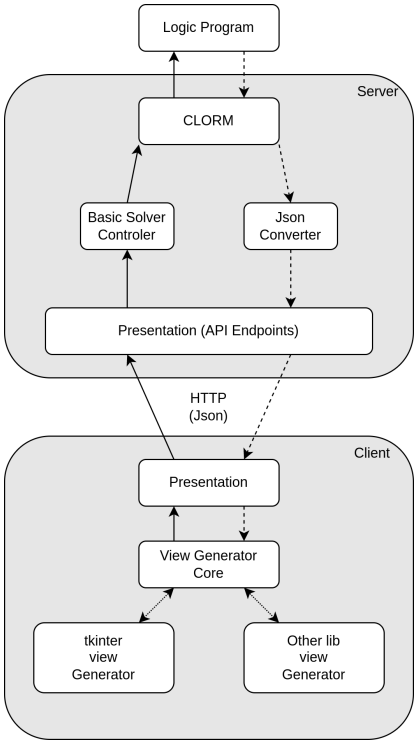

Figure 5: Cient-Server-v1 approach

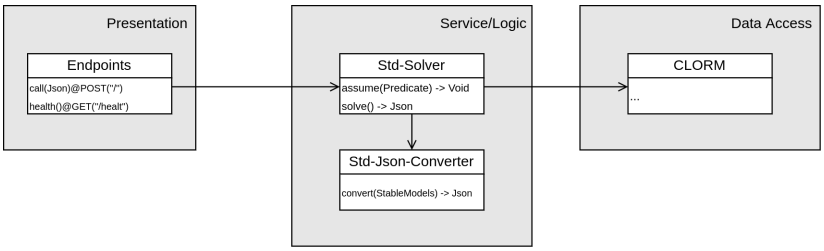

Figure 6: UML-Std.-Server

The client is also constructed in a way that should achieve, that one may switch from one python-gui framework to another. This is achieved, by splitting up the engine into a base engine (traverses the Json tree) and calling then the right method. This selection what to call is done by dependency injection. What engine to use can be specified by a command-line-argument. The UML diagram (Figure 8) depicts how this should play out. The *interface-mock* is a class that should not be instantiated, as it only contains empty/not callable functions, which shall be implemented by the actual gui-implementations. It is termed *interface-mock* as interfaces do not exists, implementation wise it could be implemented as an abstract class (which exists in python through the *ABC* library).

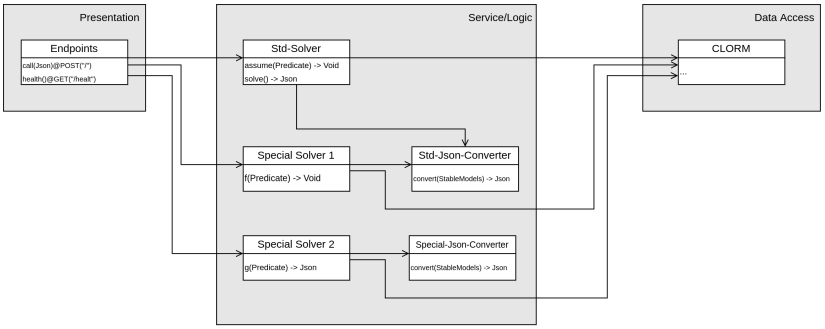

Figure 7: UML-Extended-Server

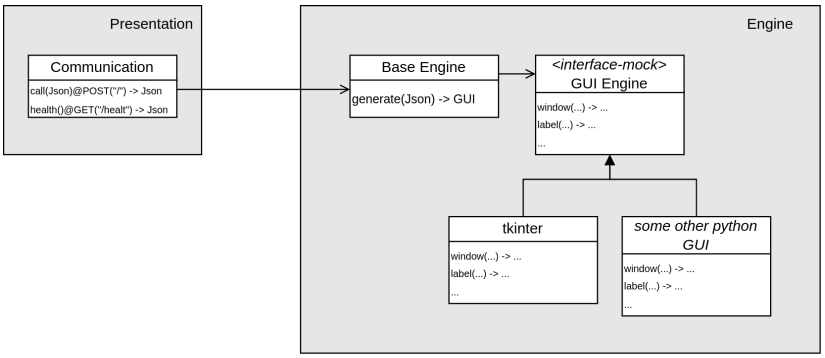

Figure 8: UML-Client

3.6 Version 2: An update

Is based on *Version 1* of the Client-Server-Architecture, a result from the meeting on the 11.07.2022 and is the first version, that shall be implemented as a prototype. The basic idea, that one implements the functionality with a Client-Server based architecture stays the same. The main difference between version *1* and *2* lies in the following:

- Only one solver should be active at the time.
- For a first prototype only a very limited set of features shall be implemented (see syntax-definition of *20220714_alex_syntax.lp*). In general the syntax shall be constructed in a more HTML way, therefore for example the possible values of a dropdown menu shall now be defined as seperate elements.
- For the prototype the Json-format the *20220711_alex_json_format.txt* will be used.
- For instrunctions how to execute the prototype see the */engine_pt/README.md* Readme-file.

The following shows a more detailed view of the workings of the prototype. At first the general idea of this prototype is outlined, then sequence diagrams are shown and then a UML-Class diagram is shown:

General Working:

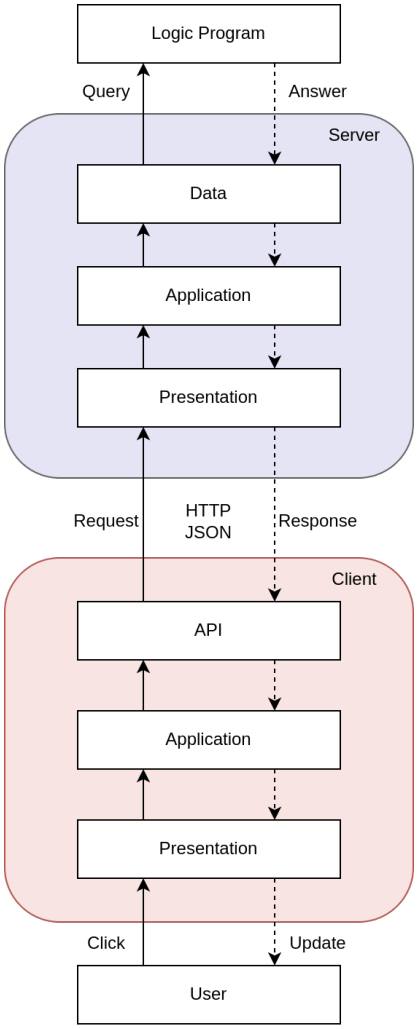

Figure 9: Architectural Overview

Sequence Diagrams

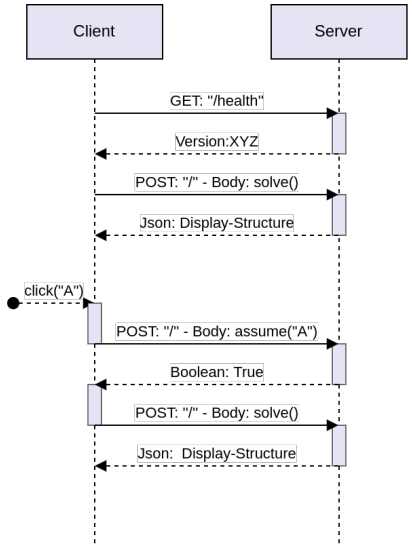

Figure 10: Sequence Diagram Normal

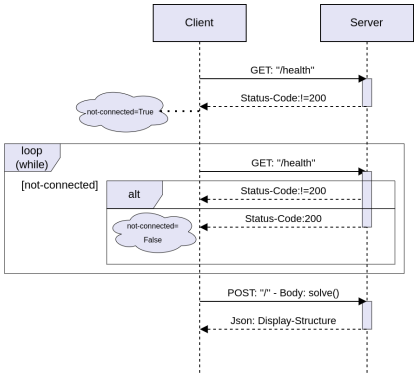

Figure 11: Sequence Diagram Failure

Class Diagram

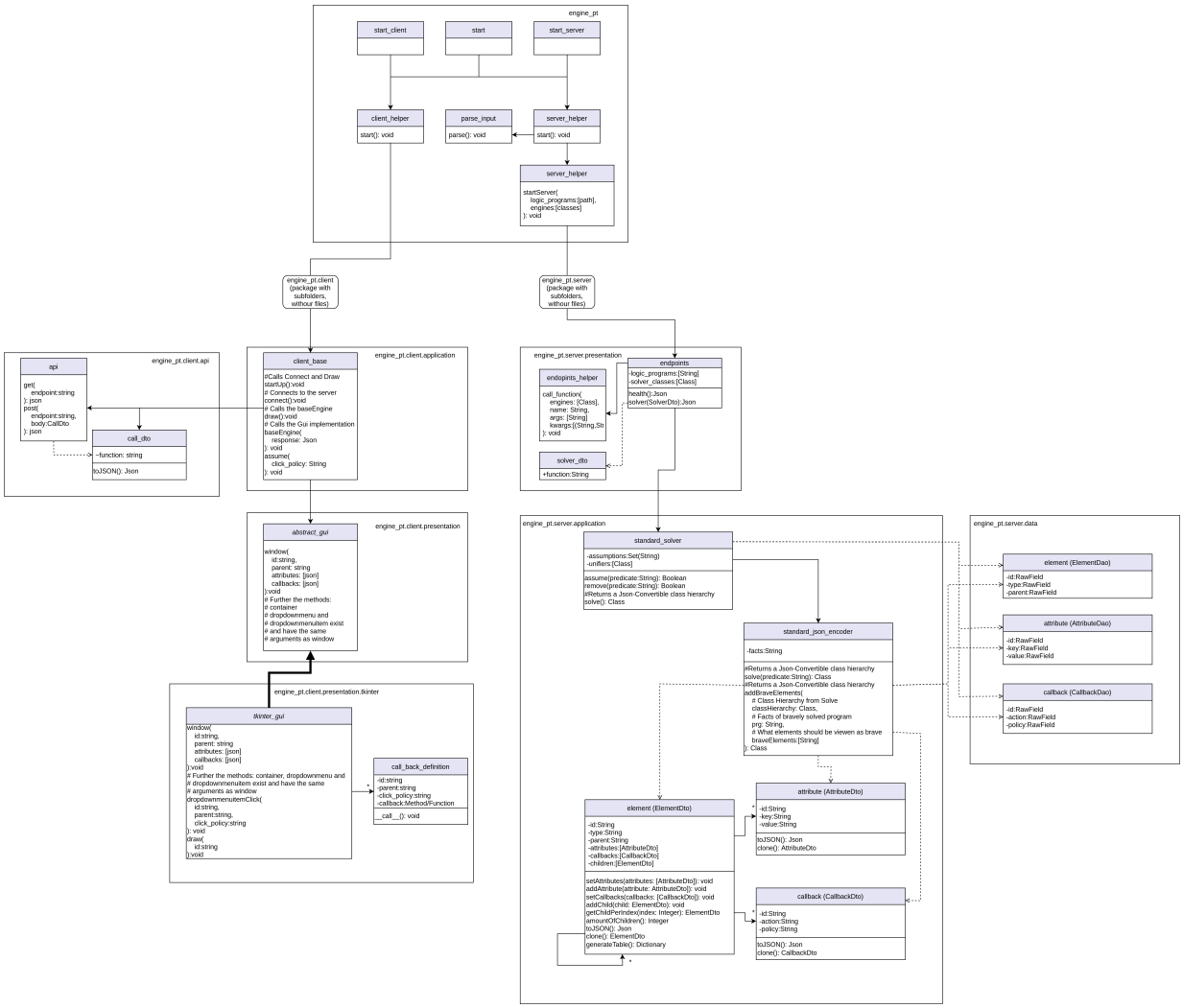

Figure 12: Class Diagram

3.7 Version 3 - Adaption and Improvement

Version 3 features various improvements on version 2, especially in terms of the user perspective. The major new/improved features are:

- Startup:
  - The application can be installed as a python package by *python setup.py install* and can then be accessed via *clinguin ....*
  - One starts the server or the client via command-line-arguments (*clinguin client*, *clinguin server ...*, *clinguin client-server*).
  - The client help section now features the actual syntax definition (*clinguin client -h*)
  - The server help section now features the actual possible arguments of the accessible solver (*clinguin server -h*).
  - One can specify which client to use (*–custom-client-classes* and *–client*).

  - The way how one specifies the server class has been refactored as well (*–custom-solver-classes* and *–solver*).

- Features:
  - Button-Element
  - Label-Element
  - Menubar-Element
  - Layout-Options for container and window
  - Various other attributes, like *onHover*, *text-font*,...
- Architecture:
  - For the user it is now easier to specify a new solver.
  - For the user it is now easier to specify a new client.
- Code-Style/Refactorings:
  - Added a style-guide (in prototype-V4 the code shall be refactored accordingly)
  - OS-Independent default paths
  - Many smaller refactorments...

3.7.1 Architecture

Architecture wise is the program in principle the same as in pt-V2 (so layered architectures both in client and server). But the details changed, so abstractly speaking one can think of the program in three parts:

1. ProgramBase
   - Contains the startup code
   - Contains the parse code
   - Contains the communication code for both client and server
2. CoreModule, aka. Solver, aka. Backend
   - Part of the server, which responsibilities include handling the callbacks, providing access to clingo, etc.
   - Can be user defined, several options possible (when the program is in a later stage of development, in this prototype just one, the *ClingoBackend*)
   - Can be specified via (*–custom-solver-classes* and *–solver*)
3. ClientGui, aka. Client
   - Part of the client, which sole responsibility is to draw the gui from a provided Json.
   - Can be user defined, several options possible (at the moment just tkinter).

Class Diagram - ProgramBase

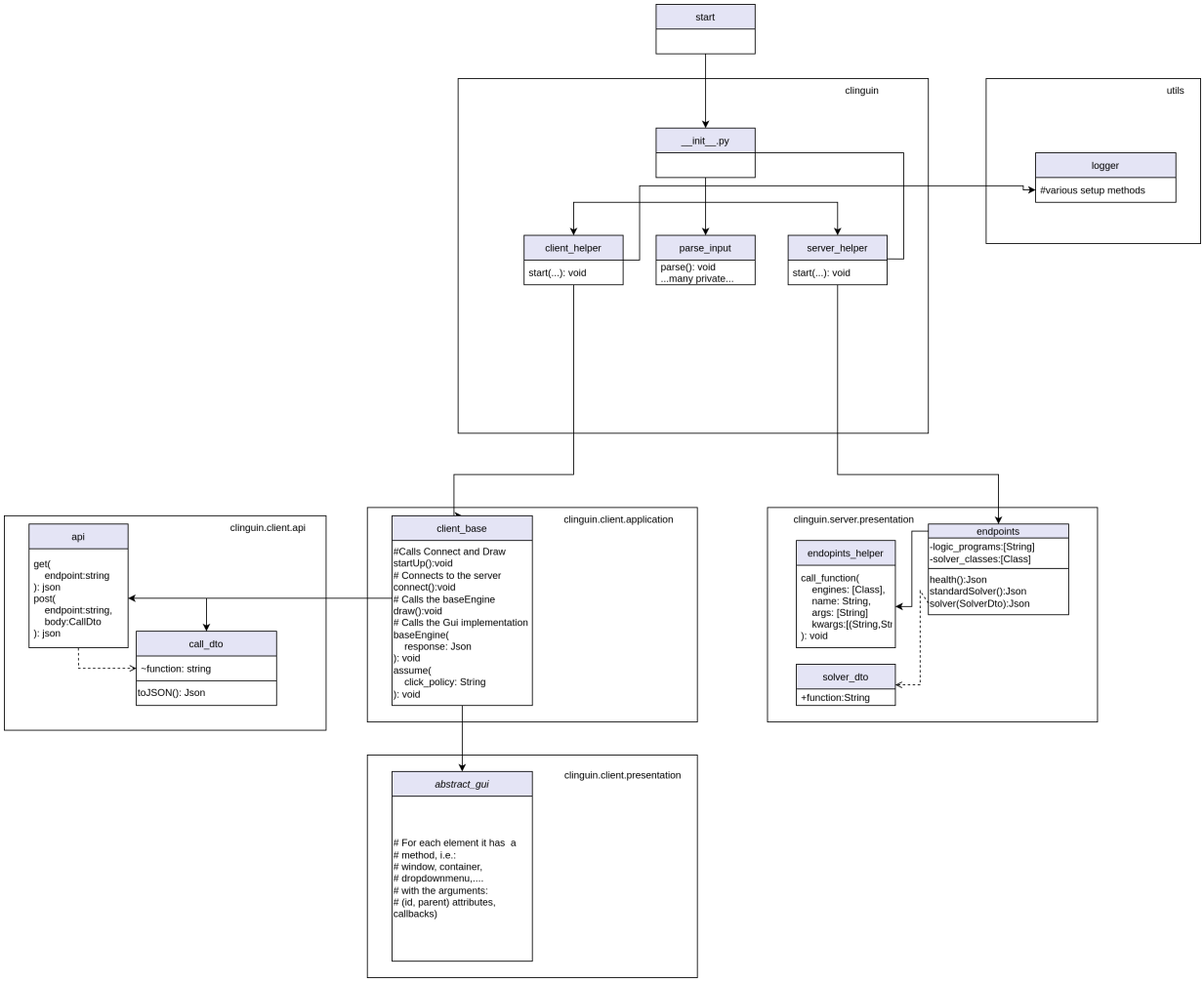

Figure 13: Class Diagram of the ProgramBase

Class Diagram - ClientGui (Tkinter)

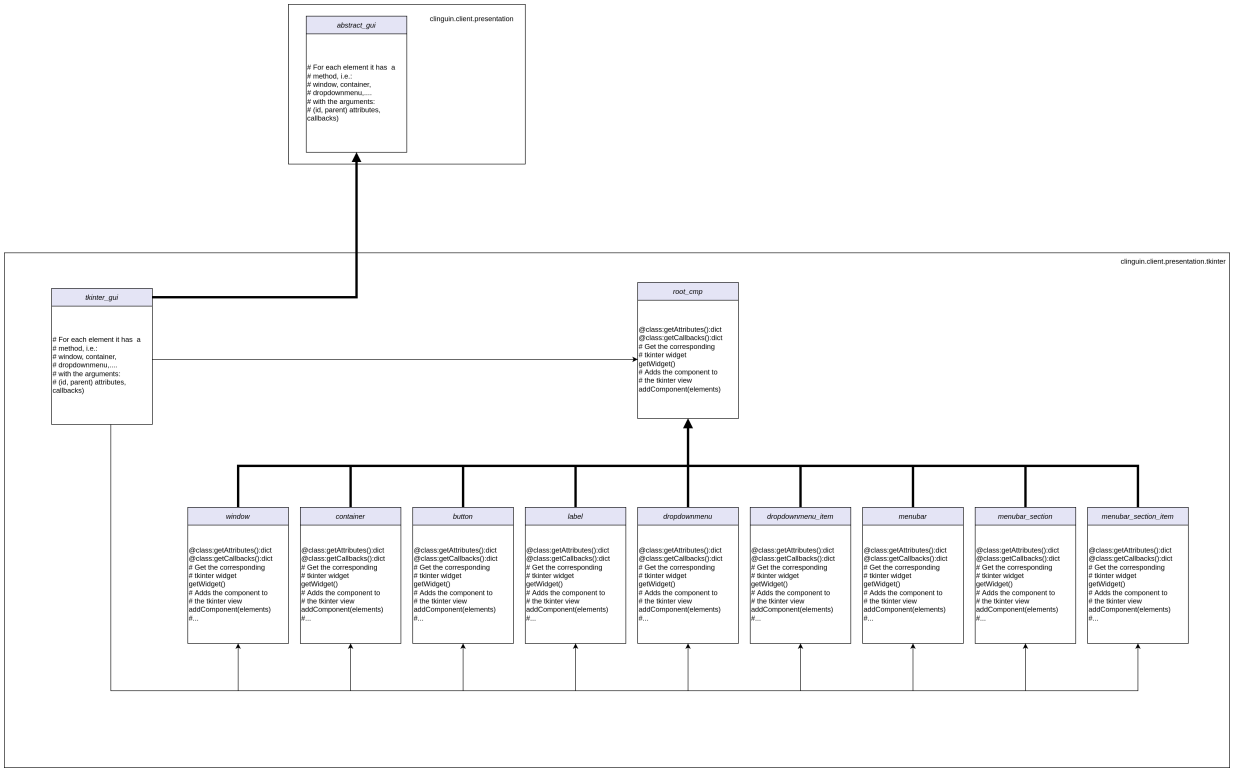

Figure 14: Class Diagram of the ClientGui (Tkinter)

Class Diagram - CoreModule (ClingoBackend)

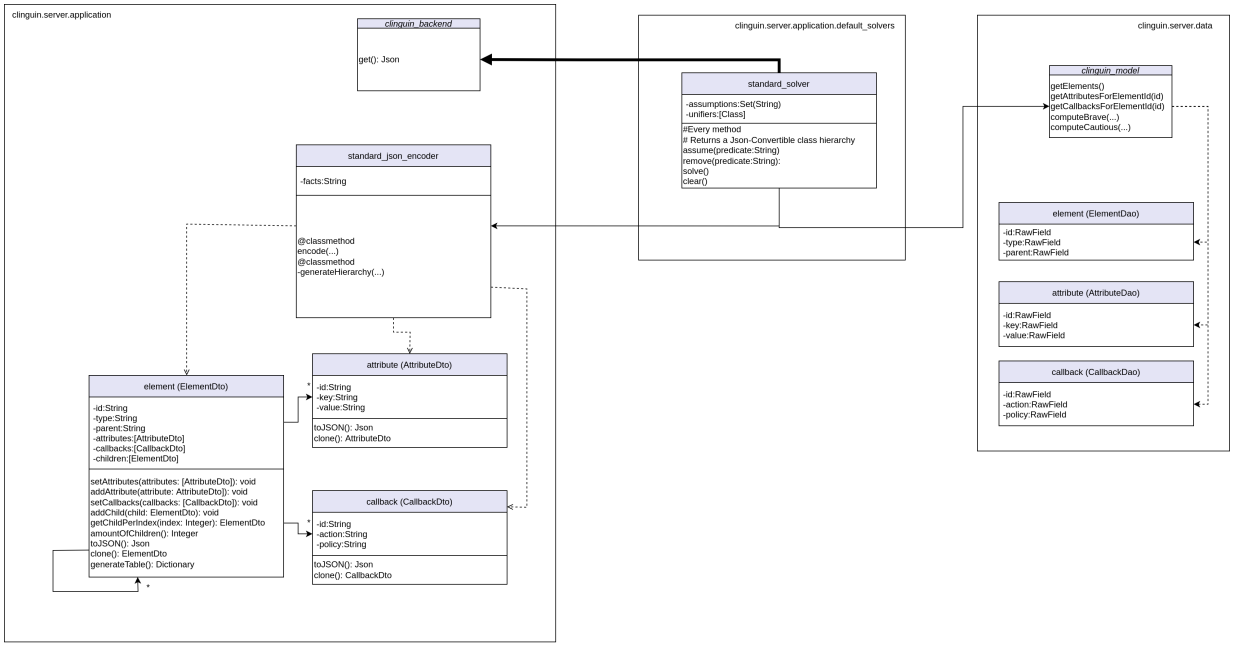

Figure 15: Class Diagram of the Core-Module

3.8 Version 4 - Features and polishing

This stage of development saw various refactorings, some improvements and some code cleanup. In terms of refactoring the part of the backend (we agreed on the term backend instead of solver) was de facto rewritten two times, to get a more general, widely applicable and extensible backend.
The following points summarize the changes:

New features:

- Backend features/examples
  - Housing example
  - Temporal clingo backend and elevator example
  - Clingraph backend and coloring example
  - TSP backend and tsp example
- GUI features
  - Canvas widget for images
  - Message widget for popup messages
  - Layouting options for label, button and dropdown menu
  - Auto fit for containers, label, button and dropdown menu
  - Manual positioning of the window
- Syntax can now be shown dynamically, with the acceptd types
- Logs are now shown in colors
- Various other improvements

Refactorings, fixes, etc.

- Backend, the behavior how one specifies which elements shall be rendered bravely or acutiously for the default backend are now customizable
- Frontend, the layouting for containers was fixed
- The argument parser was refactored, so that one can now add multiple backends with the same arguments, small further improvements

The general idea and the general architecture didn't change in this revision, therefore most of the things previously defined still hold. Regarding the available syntax, one can take a look at the recent examples. The intermediate Json format didn't change since the first implementation. Regarding the endpoints, there actually was a revision. Now there are three endpoints:

- *"/"* with GET - This is the "default" backend, where one gets the GUI one gets on startup. This endpoint calls the *get* method in the corresponding backend, which each backend **must** provide (a *"get(self):"* method).
- *"/backend"* with POST - This is the endpoint, where one can send a Json with a single key *function*, where the value of function determines which backend function will be called with what

arguments. E.g. the json *"function":"addAtom(p(1))"* tries to call the method "addAtom" with the argument "p(1)".

- *"/health"* with GET - Get version and status of program.

3.8.1 Architecture

As described in the previous section (Prototype Version 3), one could divide the program into three parts: ProgramBase, Backends and Guis.

ProgramBase: The ProgramBase contains basically the code that is necessary to start the whole application. In addition to the code from PT-V3 it also contains more util-features, like parsing types, and other functionalities. In this definition also the part of the Client is included, that handles API requests (but to stress out one thing: The client mustn't be started, therefore one can develop the whole client by oneself, but it still makes more sense to see the part of the client that handles the API as a part of the ProgramBase for this prototype).

Backend: The backend code is code that can be dynamically loaded on application startup that interacts with Clingo. Therefore several backends exist, like the ClingoBackend, ClingraphBackend or TemporalBackend. Which backend is loaded can be specified via command line arguments during startup.

Gui: The gui is, like the backend, dynamically loaded. The core of the Gui is part of the ProgramBase, therefore just the "displaying" functions are part of this Gui. These include the files for the tkinter-widgets or in the future the files for the IPython widgets.

Even though the principal working is de facto the same as before, in the following the updated-overview-UML-class diagrams are shown:

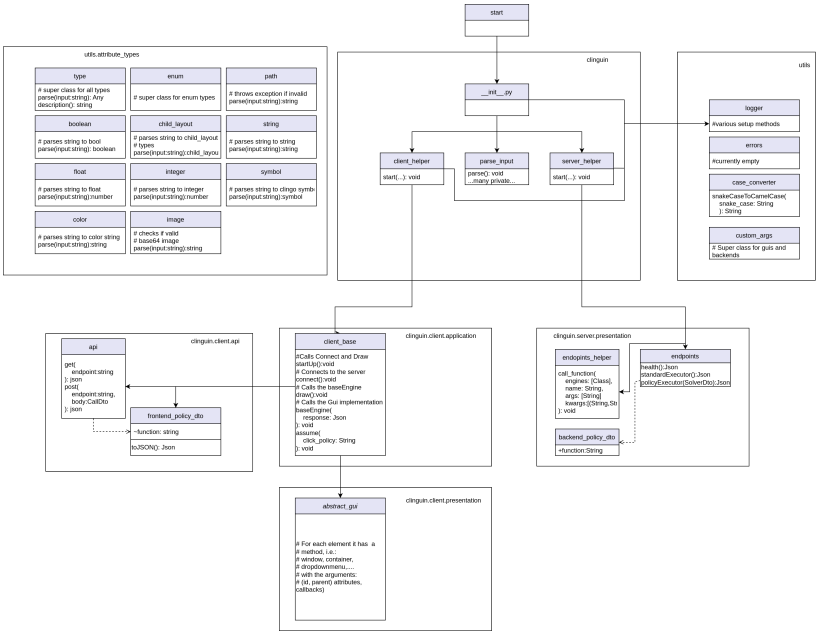

Figure 16: UML-Program-base

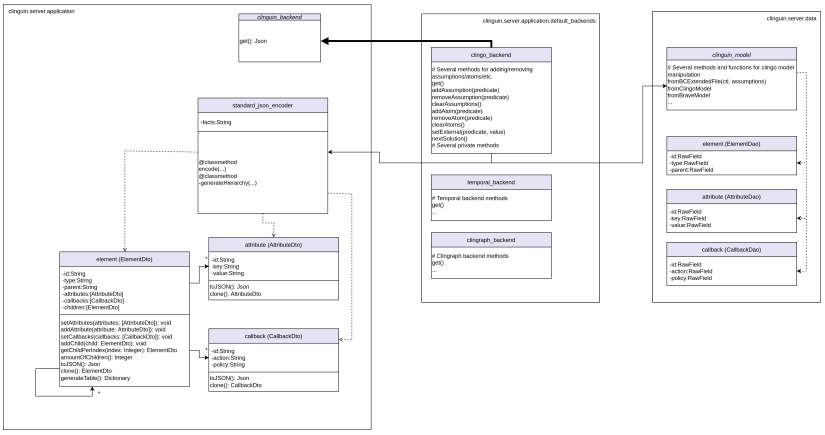

Figure 17: UML-clingo-backend

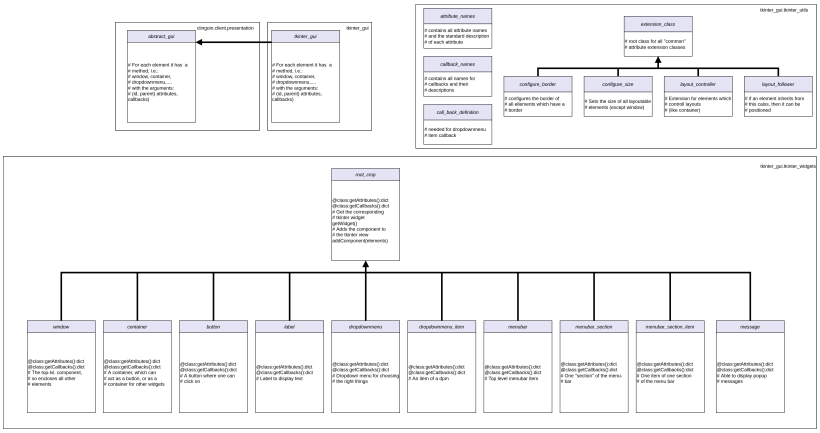

Figure 18: UML-tkinter-gui