Come, see, create

1 Vivids and Organelles

Public Specification
July 2025 by Hansjoerg Petschko

Contents

1 Release History and Notes

November 2014: First draft, still combined with the explanation of symbiosis in the VVV

December 2014: Separation from the symbiosis explanations

January 2015: First official version

January 2018: Fully updated version

May 2021: Class diagram added

October 2022: Improved ViViVerse diagrams

August 2023: More explanation about faces, configuration editing and Linux added

December 2023: Dictionary added

January 2024: Vivid synthesis diagram added

July 2025: Changes to the source code

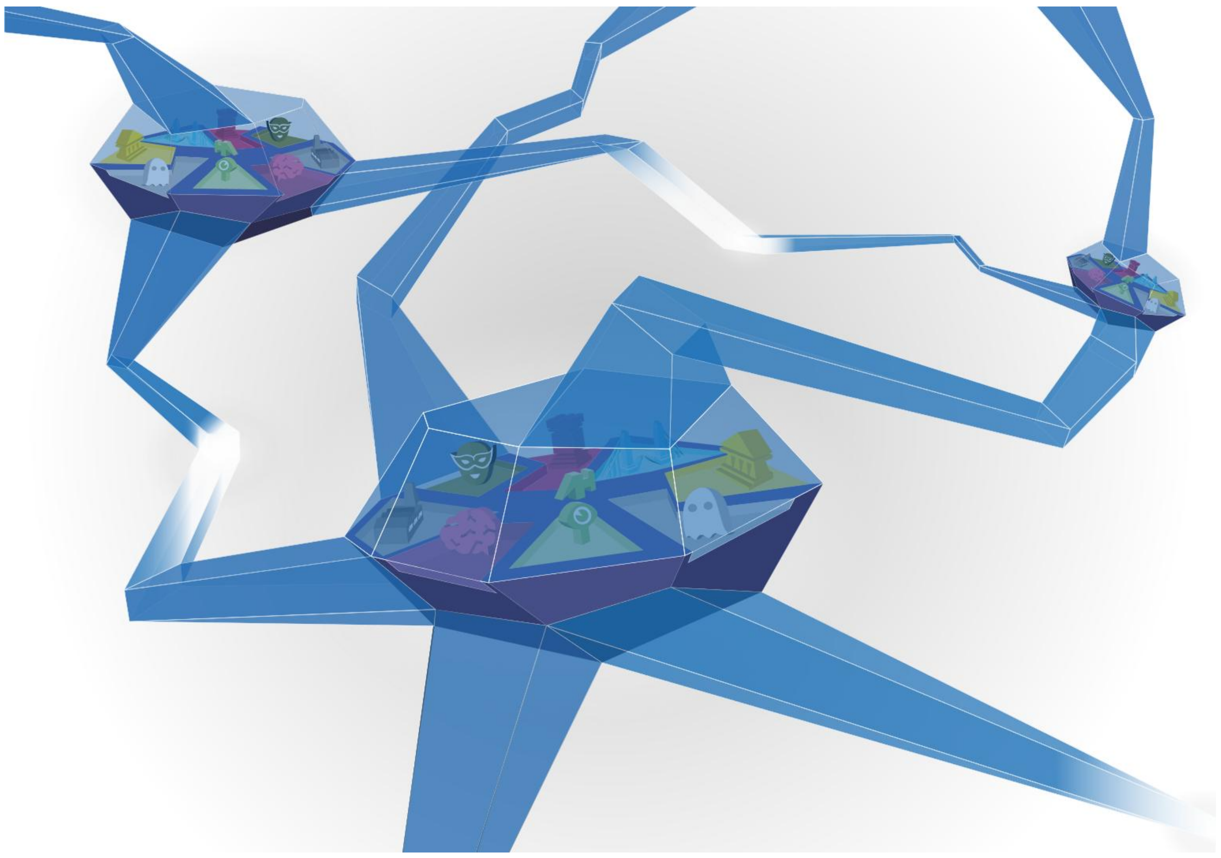
2 Introduction

Biological life forms, with their inherent mission to survive within the real, physical world, are forced to interact with the reality surrounding them. With the externally imposed purpose of the applications based on the automation software platform which constitutes the ViViVerse (VVV) being to sense, analyse and finally alter the real, physical world, the VVV finds itself in practical neighbourhood of biological life. The flexibility and adaptability required of both, life and technique, has led to similar solutions: small units which either operate standalone to a great extent, or which cooperate within a larger context. In the VVV this similarity is expressed by borrowing existing terminology from biology, but also by building new terminology based on existing one.

This document introduces the basic terms used in the VVV and tries to motivate their usage.

It is important to note, that, even though there are apparent similarities, the VVV does not pretend to be a ***bionic*** project. Analogies are not wilfully enforced - neither on a technical level nor in naming. Whenever another analogy seems more striking, it is used.

Say hello to the ViViVerse!



3 Motivation

3.1 A simple control loop: Thermostat

In this chapter, we want to introduce you to some basics of automation software and show you why the ViViVerse is as it is. For this purpose, we use a rather simple example of an automation application, the thermostat: the temperature of a system shall be regulated towards a set-temperature. The system can either be a room, a refrigerator, or a furnace.

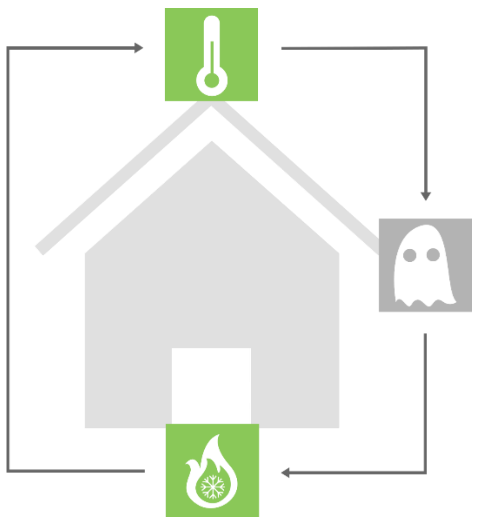


Figure 1 The principle of a thermostat as an example for a simple control loop: sense, analyse, act.

In Figure 1 we see the principle of a thermostat laid out as a ***control loop***. It consists of the following steps:

- Sense: measure the temperature
- Analyse: determine a control value from the is- and the set-temperature
- Act: control a heating/cooling element using the control value

It is called a control loop, because after the last step, the procedure starts all over again.

How could this control loop be implemented in software? An initial solution could look like this:

```
initialise temperature sensor
initialise heating/cooling device

while (true)
    read temperature
    calculate difference to set temperature
    determine control value
    write control value
end while
```

Code example 1 A minimum control loop implementation

Getting the current temperature, determining the difference from the set-temperature and the best control value, and controlling the heating/cooling element is implemented directly in one while true-loop.

While this solution will surely work, is easy to program and understand, it also has some disadvantages – mainly its lack of flexibility. If e.g., the temperature sensor changes, the software needs to be reprogrammed. What we want, is a way to change the components of the control loop at configuration time or even at runtime.

To overcome this problem, one would typically employ two familiar patterns: ***interfaces*** and ***factories***.

An interface is an ***abstraction*** of the functionality which shall be provided by an object. In the physical world, it may be compared to a connector.

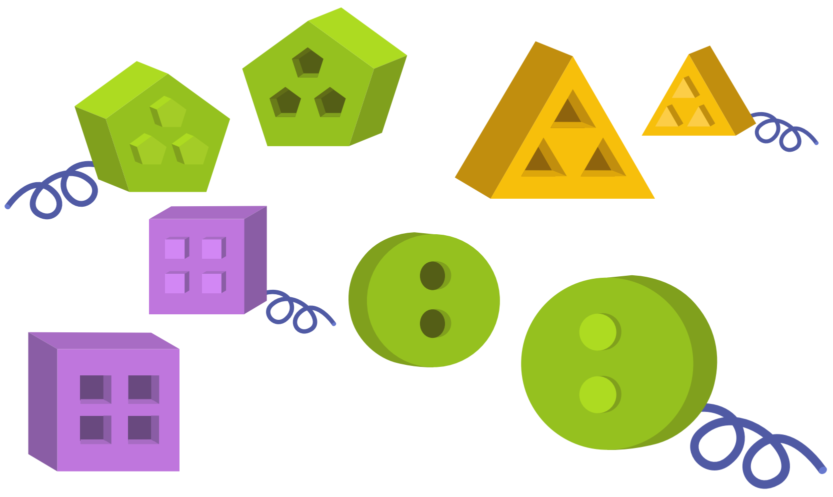


Figure 2 Software interfaces are comparable to connectors in the real world.

Like the layout of the connector is bound to a certain functionality but does not compellingly infer what is behind the jack/plug, an interface serves a clearly defined purpose without determining the way the functionality is implemented. And it is exactly this property that gives us the flexibility we were looking for: instead of implementing and using functionality directly within the control loop, we separate the two by putting an interface between the usage and the implementation. This enables us to create and 'plug in' the object which provides the functionality defined by the interface (also called 'implements the interface') before the control loop is started.

The component responsible for the creation of the component is called factory. Which component it creates for a certain purpose, can be determined by making it read from a configuration or by detecting attached hardware.

Care must be taken to define the interfaces in a sufficiently generic way, without aiming too much at the components (e.g. sensors) in your initial setup.

Now let us see, which interfaces would be needed to make the Thermostat work?

We need

- an interface to the component which connects to the temperature sensor
- an interface to the algorithm which determines the temperature difference and control value
- an interface to the component which connects to the heating/cooling element

If the application shall get a user interface as well, we may need to add a fourth interface via which the HMI can turn on and off the temperature control.

- an interface to the control loop itself for enabling/disabling it

Now the control software might look like this:

```
thermometer = factory.create("TemperatureSensor")
thermo_algo = factory.create("ThermoDesign")
thermo_ctrl = factory.create("ThermoController")

while (true)
    thermometer.read_temperature
    thermo_algo.determine_temperature_control_value
    thermo_ctrl.write_control_value
end while
```

Code example 2 The control loop using a factory and interfaces (using pseudo-object-oriented programming style).

There are three things to notice here:

First: While an interface to the control value computation algorithm seems to be an overkill in this example, think of a requirement for a clever algorithm which finds the optimal way to control the temperature with a minimum of energy. You may replace your initial, simple try once a better algorithm has been developed.

Second: Because the control loop itself also provides an interface (see the last point above) it should also be seen as a separate component: the 'control application' component. This has the very interesting consequence that the actual control application is 'demoted' to a component rather than being a 'program' in its own right.

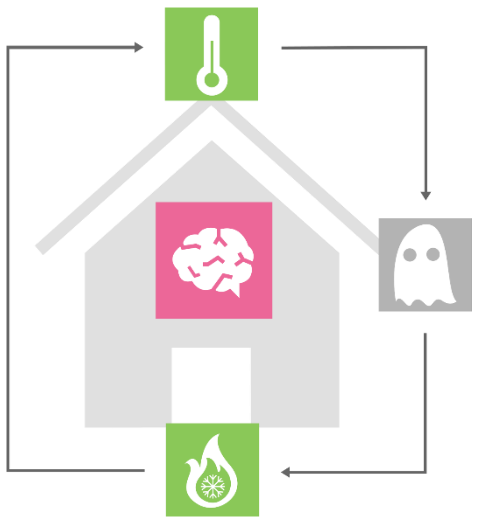


Figure 3 The principle of the Thermostat with the control application component added as a separate component (see Figure 1).

And last: This change in rank leads us to an interesting question: who makes the factory create the control application component?

We shall answer this question below.

3.2 Component Sharing and Auto Detection

The factory approach introduced in the previous chapter has its limitations though: imagine we want to add a user interface without changing the existing software. The user shall be able to see the current temperature, see and change the set-temperature and turn the thermostat on and off altogether. To do this, the HMI component needs to use the same components already used by the control loop. It can therefore not create them but must search for already existing ones.

With hardware attached (it's all about real world automation here!), there comes another requirement: software components which connect to hardware (aka things) shall be created automatically when the presence of the hardware is detected, and they shall be removed automatically when the connection to the hardware is broken.

In addition to the configuration time flexibility requirement, the components in the VVV shall also be able to share the functionality they are using with others without knowing about them and to use components which have been detected at runtime rather than been created from configuration. Also, we want to be able to add functionality without changing the software. This added functionality may want to use components already in use by others.

Example for sharing: logger or HMI using the measured temperature from the sensor component already used in the control loop. How does the logger get the same object as used in the control loop? We cannot let the factory create it twice.

Example for runtime detection: hot swapping the temperature sensor

The solution: components do not request the factory to create other components but query a **marketplace** for existing components, which registered at the marketplace upon their creation. These components have been created by the factory from configuration, on request by other components or upon detection.

Thus, the control software is split into two sections: the factory part and the actual control part.

```
while (component_name = configuration.read)
    factory.create(component_name)
end while
```

```
thermometer = market_place.find("TemperatureSensor")
thermo_algo = market_place.find("ThermoDesign")
thermo_ctrl = market_place.find("ThermoController")

while (true)
    thermometer.read_temperature
    thermo_algo.determine_temperature_control_value
    thermo_ctrl.write_control_value
end while
```

Code example 3 The control loop using a factory, a marketplace, and interfaces.

This final approach is the one used by the ViViVerse. In addition, the ViViVerse allows queries to be performed in **remote** marketplaces and components to be used transparently across process and machine boundaries, by this enabling **IoT** and **cloud** scenarios.

After having shown – compellingly - the reasons for its architectural design, we will now introduce you to some more transcending aspects of the ViViVerse, but also its real implementation.

4 Vivids and Organelles

4.1 Biological cells and organelles

The cells of all living organisms on earth have a very similar base structure. In fact, they are so similar, one could state that they all stem from a single cell.

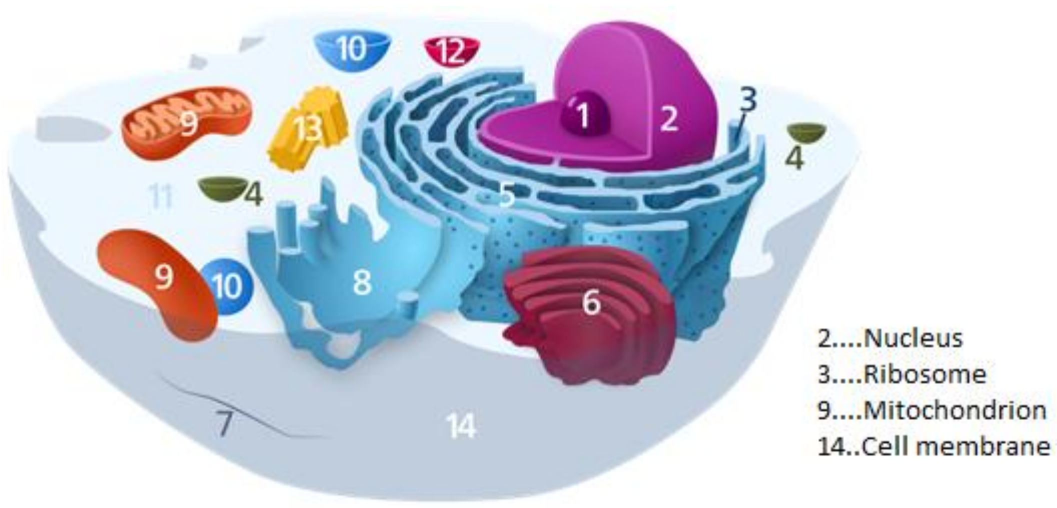


Figure 4 A biological cell and its main components (source: Wikipedia).

Some of the main delimited components of the cell are called ***organelles***.

In addition to the organelles present in almost all cells, like ribosomes or mitochondria, cells can also have additional organelles which differentiate them and provide them with the capabilities necessary to survive in various environments or 'function' in a larger context.

The biological model organism in the VVV world is the Euglena Gracilis. As additional organelles, it possesses a flagellum which it uses as propeller and an eyespot for the detection of light, which it needs, because its metabolism is based on photosynthesis. It lends its shape to the graphical representation of a ***Vivid***.

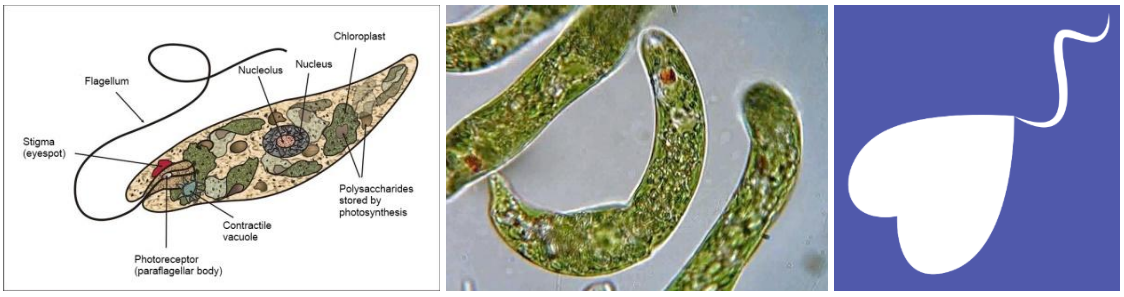

Chloroplast
Nucleolus
Nucleus
Flagellum
Stigma (eyespot)
Polysaccharides stored by photosynthesis
Contractile vacuole
Photoreceptor (paraflagellar body)

Figure 5 Euglena Gracilis: structure, microscopic picture, and graphical representation of a Vivid (source: Wikipedia).

4.2 The Vivid

Vivids are the smallest operational units within the VVV. Initially, a Vivid can be compared to a basic cell: it cannot do a lot. But together with its basic organelles and their capability to synthesise and organise additional organelles, it becomes the core building block of the VVV.

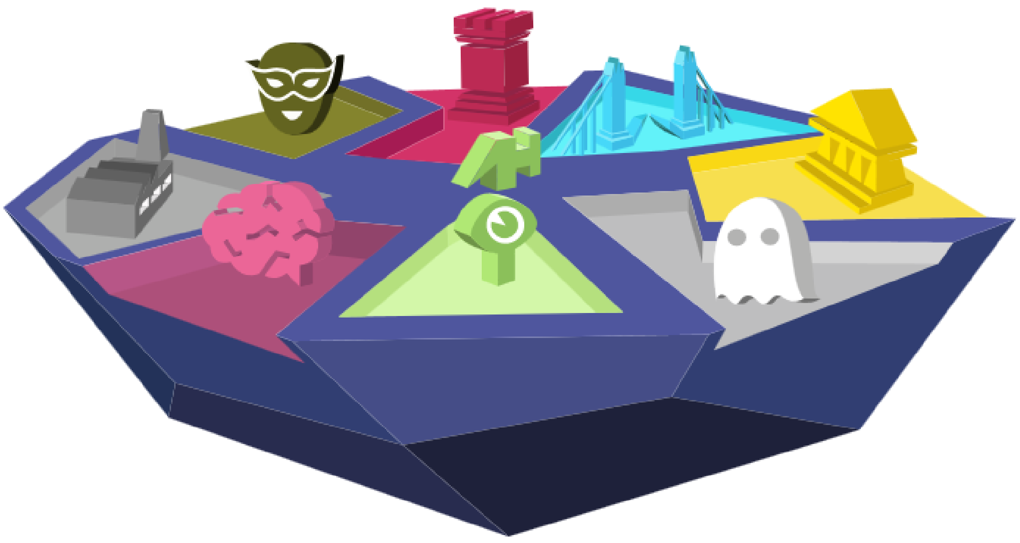


Figure 6 A Vivid and its main components

Figure 6 shows a Vivid and its organelles. They will be explained in more detail in the following sections.

Like one biological cell can either be a life form in its own right, or be part of a larger organism, one Vivid can either constitute an entire automation application or only contain partial functionality. It is the same mechanism that enables this separation that also makes it possible to combine full application Vivids to meta-applications.

Since the VVV is implemented in C++, which is an object-oriented programming language, a Vivid and the organelles it contains, are represented by software objects. The creation of these software objects is called ***synthesis*** in the VVV. The deletion of such an object is called ***dissolution***. When a software object (the ***acceptor***) uses the services provided by another software object (the ***donor***), they are in a ***symbiosis*** with each other.

4.3 Core organelles

Each Vivid has three static (meaning that they are always there) organelles. They are called ***core organelles***.

Agora
This is the organelle registry or marketplace of the Vivid. It is the place where the organelles register, can find each other, and then start symbiosis, hence its name.

Chaperone
This is the defender of the Vivid. It is responsible for the authentication of other Vivids and their organelles and for the authorisation of symbiosis actions.

Demiurg
This is the factory of the Vivid. It creates (synthesises) and organises the dynamic organelles which define the actual nature of the Vivid.

In addition to the three static organelles, a vivid can also have any number of connections to other Vivids. These connections are effected by ***dendrons***.

Dendron
Dendrons use bridges and the ***Whisper*** protocol to connect vivids, by this enabling symbioses between organelles in separate Vivids. The dendrons are managed by the Chaperone. The dendrons between two Vivids meet at a *Synapse*.

4.4 Pheno organelles

The dynamic or additional organelles of the Vivid are called ***pheno organelles*** because they define the appearance, behaviour, and capabilities of the Vivid towards the outside world. The VVV knows 5 pheno organelle classes:

Bridge
The communication resources like socket, serial port, CAN bus.

Rebus
Sensors, actuators, and other things (like the computer object).

Fantasma
This class comprises algorithms. Anything that should be replaceable via configuration or be accessible remotely but does not belong to any of the other classes.

Ego
The primary application algorithms or application know how.

Visage
This class comprises the HMI components or ***bioware*** drivers.

The Demiurg synthesises the pheno organelles when the Vivid itself is synthesised, based on a configuration file which could be regarded as the DNA of the VVV. Pheno organelles can also be synthesised or dissolved during runtime, based on runtime needs and circumstances.

5 Taxonomy

Taxonomy is the science of classifying items. The art behind it is to create a set of reasonable criteria which allows an object to be assigned to a category and by this be found more easily in the Agora.

Each taxonomic scheme knows several levels of specialisation, called ***ranks***. The name of the category in a specific rank is called ***taxon***.

5.1 In Biology

Because of the huge diversity of life forms, biology knows a lot of ranks.

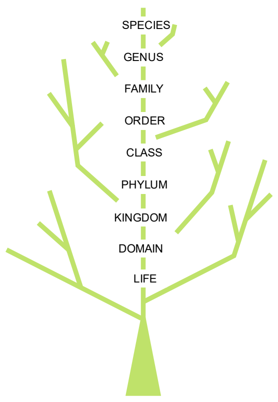


Figure 7 The 9 taxonomic ranks in biology in ascending top-down specialisation order.

For example, the full classification of the domestic cat is:

```
Animalia/Chordata/Mammalia/Carnivora/Felidae/Felis/Felis catus
```

Here, one taxon is given for each rank starting with the kingdom.

5.2 In the ViViVerse

The VVV only knows 4 ranks, which however should not be seen as a limitation of the diversity within her. These ranks are (in ascending specialisation order): class, family, species, and identity.

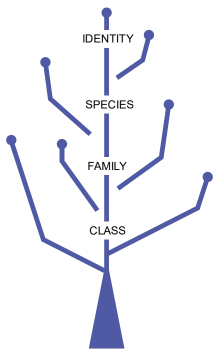


Figure 8 The 4 taxonomic ranks in the VVV, in ascending top-down specialisation order.

5.2.1 The Organelle Class

The organelle class rank (or short: ***class*** rank) is the collection of the core organelle and the pheno organelle classes plus the Vivid as ***meta*** organelle.

Organelle classes are denoted in C++ by the enumeration value `organelle_class`.

```
enum class organelle_class : i32
{
    organelle_class::none = -1,                         //  undefined
    organelle_class::vivid,                             //  vivid (meta organelle)
    organelle_class::agora,                             //  agora (core organelle)
    organelle_class::chaperone,                         //  chaperone (core organelle)
    organelle_class::demiurg,                           //  demiurg (core organelle)
    organelle_class::dendron,                           //  dendron (core organelle)
    organelle_class::bridge,                            //  bridge class (pheno organelle)
    organelle_class::rebus,                             //  rebus class (pheno organelle)
    organelle_class::fanta,                             //  fanta class (pheno organelle)
    organelle_class::ego,                               //  ego class (pheno organelle)
    organelle_class::visage                             //  visage class (pheno organelle)
};  //  enum organelle_class
```

All non pheno classes – like Chaperone or Demiurg - do actually have only one class member having the class name as member name.

5.2.2 The Family

Families are contextually associated sets of services provided by one organelle and used by another organelle or generally any other software object. This rank definitely is the most important in the VVV taxonomy. Families are denoted by their name string and a ***guid***. A guid is a very large number which is guaranteed to be unique.

This rank shows a noticeable difference from its biological counterpart, in that family membership is not exclusive: because the family defines the purpose of an organelle, multipurpose organelles automatically belong to multiple families (and thus, by the way, may also belong to multiple classes, which is rather rare).

Because it has such an important role, it has got its own icon:

Figure 9 The Family icon.

5.2.3 The Species

Species denote the different implementations of the services provided by a family. Sensors of different manu-facturers, which all provide the same type of measurements, belong to different species. Species are denoted by their name string.

5.2.4 The Identity

The identity identifies an individual. An identity comprises a name string, an organelle id, which is a number unique within the vivid, the guid of the vivid and the guids of the Synapses (the connections between the dendrons) through which they were registered in the Agora. This last collection of values is only relevant for ***ectoviv symbiosis***, the collaboration of organelles across vivid boundaries, which is described in the document `2 Symbiosis in the ViViVerse`. The Synapses describe the topology of the ViViVerse, but their use has not been completely elaborated until now.

Identities are called ***persistent*** if they keep their names between incarnations of the same vivid and can thus be fully referenced in the configurations.

5.2.5 The Holotaxon

Since there exists no term for the full taxonomic definition of a specimen in biology (as given for the cat), the VVV has introduced the word ***holotaxon***. It consists of all four ranks.

The holotaxon given by class/family/species/identity of a temperature sensor of type 'Top' made by manufacturer 'ManuX' is:

```
organelle_class::rebus/TemperatureSensor/ManuXTop/ManuXOnCom2
```

For clarity, the string representation of the class enumeration value is used, and only the name part of the identity is shown.

Holotaxa can also be partially defined by omitting rank taxa completely or using wildcards in strings. This feature is used when acceptors are looking for donors in the Agora using ***donor queries***, as is explained in the document about the ViViVersal symbiosis.

6 The life of a simple Vivid

6.1 Basic source code

The 'life' of a Vivid starts when the member function `synthesise` of the associated software class `vivid` is called. A configuration file, describing amongst others the pheno organelles to be synthesised, must be passed to this function. The default filename extension of VVV configuration files is `vvvdna`. The life of the Vivid ends when the function `dissolve` is called. During the time between the calls to `synthesise` and `dissolve`, the Vivid leads a completely independent life, performing the tasks it was meant to. Hiding the function calls in the constructor and destructor of the class `vivid`, the simplest program hosting a Vivid could therefore look like this:

```
i32 wmain(
      i32    argc,
      wchar* argv[])
{
    vivid v(argc > 1 ? argv[1] : nullptr);          //  the vivid which performs the automation task
    getwchar();                                     //  let the task run until the user presses a key
    return 0;
}
```

Code example 4 The simplest host program being able to run any automation application

Depending on the pheno organelles, which are synthesised during the synthesis of the Vivid (or thereafter), this tiny piece of software can do virtually anything. As long as there are no resource collisions (such as the attempted usage of a serial port already in use), any number of Vivids could be synthesised in this program by defining more objects of the class `vivid`.

6.2 Basic configuration

As an example for a simple automation application, we will use the Thermostat application introduced earlier in this document. How would the ViViVerse implementation look? Here is the organelle diagram, which shows the necessary components and their relationships:

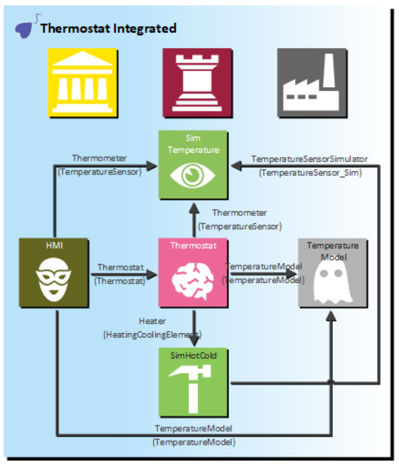

Thermostat Integrated
Sim Temperature
Thermometer (TemperatureSensor)
TemperatureSensorSimulator (TemperatureSensor_Sim)
Thermometer (TemperatureSensor)
HMI
Thermostat
Temperature Model
Thermostat (Thermostat)
TemperatureModel (TemperatureModel)
Heater (HeatingCoolingElement)
SimHotCold
TemperatureModel (TemperatureModel)

Figure 10 The ViViVerse organelle diagram for the Thermostat application.

The labels on the connecting lines denote the names of the symbioses and the required families in parenthesis. The small arrow on one end indicates the **donor** (the organelle which provides the functionality); the **acceptor** sits on the other end. In many cases (like the symbiosis between the Thermostat Ego and the TemperatureModel Fantasma), the name of the symbiosis is equal to the family name. This is not possible if symbioses to two organelles of the same family are needed - for example, if two temperatures must be measured.

The DNA files, which contain the configuration, are written using a reduced version of XML. The DNA file for the Thermostat Vivid could look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<VVV_Vivid>
  <Name>ThermostatComplete</Name>
  <Demiurg>
    <OrganelleModules>
      <Module>vvv_thermostat.vvvorg</Module>
    </OrganelleModules>
    <Synthesis>
      <Base.SimTemperature>
        <Temperature>293.150000000000</Temperature>
        <DeliveryInterval>1000</DeliveryInterval>
        <DeliveryMode>2</DeliveryMode>
      </Base.SimTemperature>
      <Base.TemperatureModel>
        <SetTemperature>295.150000000000</SetTemperature>
        <DeadBandWidth>0.100000000000</DeadBandWidth>
      </Base.TemperatureModel>
      <Base.SimHotCold>
        <Environment Type="C">
        </Environment>
        <StartTemperature>293.150000000000</StartTemperature>
        <TemperatureChange>0.100000000000</TemperatureChange>
        <SetInterval>1000</SetInterval>
        <SetTemperature>true</SetTemperature>
      </Base.SimHotCold>
      <Base.Thermostat>
        <Environment Type="C">
        </Environment>
        <ControlInterval>1000</ControlInterval>
      </Base.Thermostat>
      <Base.ThermostatW32 Id="ThermostatHMI">
        <AutoSynthesise>false</AutoSynthesise>
        <View>
          <BackgroundColour>0 0 0</BackgroundColour>
          <ForegroundColour>200 200 200</ForegroundColour>
        </View>
      </Base.ThermostatW32>
    </Synthesis>
  </Demiurg>
</VVV_Vivid>
```

Code example 5 The configuration of the Thermostat Vivid with integrated HMI

As we already mentioned, the Demiurg is responsible for synthesising the pheno organelles. Before they can be synthesised, the module `vvv_thermostat.vvvorg` which contains the necessary executable code must be loaded. In the `Synthesis` section, five independent organelles (`SimTemperature`, `TemperatureModel`, `SimHotCold`, `Thermostat`, `ThermostatW32`) are defined, using their species names.

Once synthesised, organelles which need functionality from other organelles, must find them. This is a two-step procedure, which starts with a query against the Agora. The search criteria of this query only define the basic requirements and are hardcoded, e.g., to get the is-temperature, the Ego needs a member of the `TemperatureSensor` family. If more detailed filtering is necessary, the `Environment` entries can be used to define the ***symbioses*** between the organelles. It is only necessary to write them into the configuration if special requirements need to be met. If, in the above example, the configuration should only accept 'real' temperature sensors and no simulators, the following entry into the environment section would exclude simulators for the `Thermometer` symbiosis, because they belong to the `Fantasma` class.

```
      <Base.Thermostat>
        <Environment>
          <Thermometer>
            <Holotaxon>
            <Class>Rebus</Class>
            </Holotaxon>
          </Thermometer>
        </Environment>
        <ControlInterval>1000</ControlInterval>
      </Base.Thermostat>
```

Code example 6 Using restrictions for symbioses.

If these conditions are not fulfilled, the Ego will not find the donors it needs and will not work.

In the Thermostat example, this restriction does not make much sense, because the temperature sensor simulator is synthesised explicitly and having a contradicting configuration would apparently break the application. However, the VVV supports automatic detection of Things and when more than one sensor of the same family or when a sensor of an unacceptable accuracy class is connected, these entries enable the selection of the correct symbionts.

For more information about the configuration and symbiosis between the organelles, please see the documents `6 The ViViVerse System Services.docx` and `2 Symbiosis in the ViViVerse.docx` from the ViViVerse Documentation Suite.

The latter document also explains the feature which transforms a collection of independent Vivids into the ViViVerse: ***ectoviv*** symbioses – symbioses between organelles within separate Vivids. This feature enables you for example to quickly move the HMI into its own piece of hardware, without changing a single line of source code!

7 More about pheno organelles

7.1 Application development

There are several approaches to develop a real-world[1] application using the ViViVerse. The pure approach uses a general-purpose host program which has one or more vivids embedded. The application specific functionality resides exclusively in the plug-ins loaded on vivid synthesis, while the host program knows nothing about the specific purpose. But it is also possible to use a mixed approach, with having parts of the application specific functionality, like the user interface, in the host program and letting the Vivid and its organelles only do the rest. Whatever way is chosen, the functionality provided by the ViViVerse comes in form of the pheno organelles. While the Vivid and the core organelles are merely infrastructure, it is the pheno organelles that do the real application job. They are even used internally by the core organelles; for example, for authentication purposes the Chaperone uses the family `SecurityDatabase`[2].

The pheno organelles are packed into **organelle modules**, which are dynamically loadable libraries. These libraries are assembled by referencing the organelles implemented in static libraries, the **organelle collections**. Organelle collections for shared usage are organised in **packages** which serve a certain purpose, like fleet management, visual quality control or others. From these packages and application specific organelle collections, the organelle module for an application is built. It is also possible to split an application into any number of organelle modules for greater flexibility, with the extreme case of having each organelle in its own module.

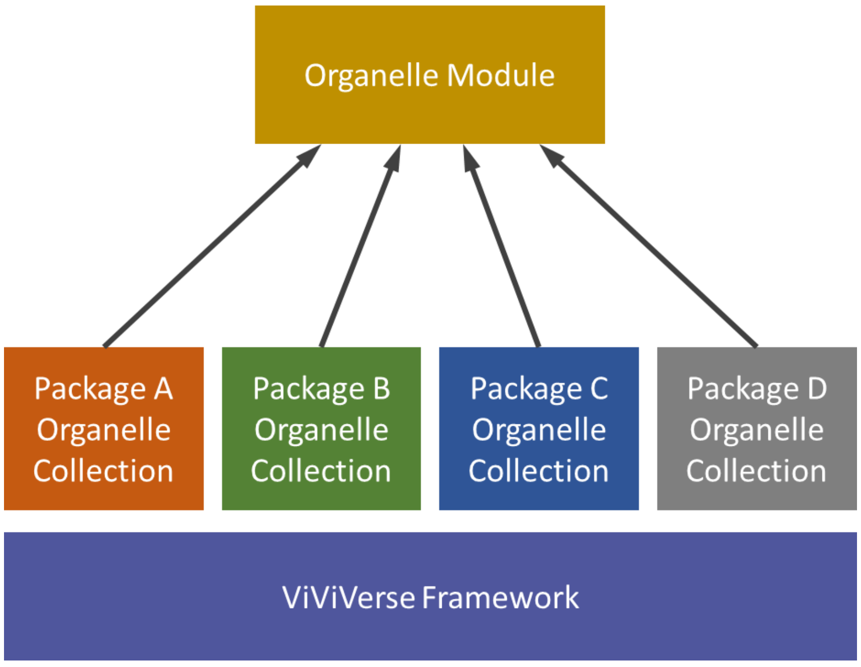

Organelle Module
Package A Organelle Collection
Package B Organelle Collection
Package C Organelle Collection
Package D Organelle Collection
ViViVerse Framework

Figure 11 Assembling an organelle module, e.g., representing an application, from various organelle packages.

For more information about how organelle collections, packages and Organelle modules are organised, see the document `5 The ViViVerse Folder Structure`.

[1] we use this word to distinguish this application from executable software
[2] more precisely: the dendrons do so

7.2 Implementing pheno organelles

So, if you want to provide functionality, how can you do that in the VVV?

First you must decide whether it is necessary at all to implement an organelle. The flexibility they provide clearly comes at the price of a higher implementation effort, so the benefit must outweigh the additional costs. There are two use cases which justify the existence of organelles:

- Plugin flexibility. Your functionality shall be added to an existing application by just adding a software module and an entry into the configuration, or you want to be able to replace one implementation by another by changing the configuration.
- Remote accessibility. Your functionality shall be used by others from within separate vivids, which may live in separate processes or on separate machines.

If none of these use cases applies, creating a class in a library will do the job. Take as an example an implementation of the Fast Fourier Transform (FFT) algorithm. Unless you want to do some cloud- computing or want to be able to replace the implementation 'on the fly', there is no need to put that into a Fantasma.

If you decide to implement an organelle, the amount of additional work depends on whether the organelle shall provide its functionality as member of a family and if so, whether this family has already been founded, i.e., whether the feature set of the family has already been defined or not.

Not all organelles need to provide functionality to other organelles to be useful. Imagine a logger, which collects values from other organelles and writes them to a file. It uses the family feature sets of others but does itself only work effective in the outside world, not within the VVV. It does not have to be controllable by others and therefore does not have to be a member of any family.

If the organelle shall provide functionality to other software components within the VVV, this should be done via family symbiosis. If you are lucky and the family has already been founded, just declare your organelle a member of the family and provide the promised features. If the family does not yet exist, you can either take the easier way and just make the family feature set equal to what your intended implementation must offer, or you can sit down and think beyond your specific implementation. If you have a simple temperature sensor, the family could just provide the temperature values. But you could also think about more complex cases and add reset and calibration commands to the feature set. If that deters and demotivates you from founding families, there is mitigation: ***derived families***. Just start with the easiest, most basic family and add features later to families, which provide the base family feature set plus the new stuff. The temperature sensor simulator from our example only indirectly belongs to the `TemperatureSensor` family. Actually, it belongs to the family `TemperatureSensor_Sim` family, which provides all the features of its base family but adds the ability to set the 'measured' temperature from outside. Derivation is indicated by an underscore in the family name.

Choosing the appropriate organelle class comes next. Bridges are quite easy to identify, as are Things. Any organelle that represents a piece of hardware; sensors, actuators, but also the computer the software is running on, belong to the latter class. Visages are a bit trickier: while it seems clear, that software, which writes to the screen and reacts on mouse clicks or touch events, belongs to this class, where should we put a piece of software that reacts on a hardware button being pushed or turns a siren on and off? Actually, the uncertainty here is an indication that we do not differentiate clearly between the interaction concept and the interaction media, the reaction on a stimulus and the kind of stimulus. For this reason, Visages are separated into two subcategories: ***models*** and ***views***. Read more about them in the document about HMIs. The rest are Fantasmas. Well, all the remaining organelles, except for those that you reckon the centrepieces of your applications. In the Thermostat example, the `TemperatureModel` organelle, although being indispensable, is just one component, while the `Thermostat` organelle basically represents the application. It is the one that knows. Alas, here you have your Ego.

Organelles are implemented in static libraries. But how do you get them working?

7.3 Deploying pheno organelles

With one goal of the VVV providing functional flexibility, using a plug-in mechanism is the logical choice. And with the organelles being the vehicles of the functionality, they will consequently be contained by the organelle modules. On the Windows operating system, these modules are Dlls, on Linux dynamic library files with the `.so` file extension.

Creating an organelle module is straightforward: reference the organelle and link the static library. Here is (almost) all that you must do:

```
//  this array contains all organelles of the module
const organelle_info*  organelle_module_implementation<>::organelle_infos_[] =
{
    &fanta_library::sim_temperature_org_info,
    &fanta_library::temperature_model_org_info,
    &fanta_library::sim_hot_cold_org_info,
    &ego_library::thermostat_org_info,
    &visage_library::thermostat_w32_org_info
};  //  const organelle_info
```

Code example 7 The organelle map in the organelle module vvv_thermostat.vvvorg

As in the Thermostat example, all organelles of your application can go into one module, or you can (for larger applications) move them into class modules (one for Bridges, one for Things etc.). In any case, you can add as many modules as you want. Before it synthesises the organelles, the Demiurg loads the modules given in the respective configuration section.

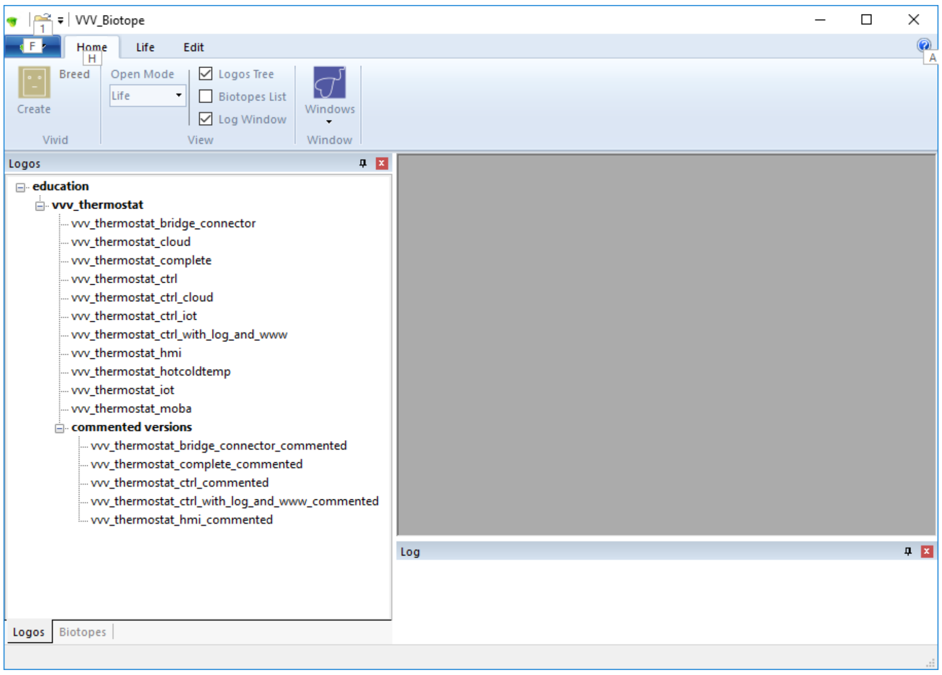
8 Appendix A: The Thermostat example

In this chapter, we take a closer look at various instances of the Thermostat example by describing their purpose, configuration, and runtime behaviour. For a more detailed description of the configurations, please read the commented versions in the configuration folder of your ViViVerse installation: `ViViVerse\software\runtime\configuration\education\vvv_thermostat\commented versions`.

8.1 Introduction to the vvv_biotope host program

To explore the Thermostat example, we will use the desktop host program `vvv_biotope`. It is a multiple document interface application, with each loaded configuration representing one document.

Note: the explanations and screenshots in this chapter relate to the Windows version. The Linux version is built on the `ncurses` library and therefore slightly less spectacular.



education
- vvv_thermostat
  - vvv_thermostat_bridge_connector
  - vvv_thermostat_cloud
  - vvv_thermostat_complete
  - vvv_thermostat_ctrl
  - vvv_thermostat_ctrl_cloud
  - vvv_thermostat_ctrl_iot
  - vvv_thermostat_ctrl_with_log_and_www
  - vvv_thermostat_hmi
  - vvv_thermostat_hotcoldtemp
  - vvv_thermostat_iot
  - vvv_thermostat_moba
  - commented versions
    - vvv_thermostat_bridge_connector_commented
    - vvv_thermostat_complete_commented
    - vvv_thermostat_ctrl_commented
    - vvv_thermostat_ctrl_with_log_and_www_commented
    - vvv_thermostat_hmi_commented

Figure 12 The vvv_biotope host program showing the available configurations.

After starting the program, the available configurations are displayed (`vvv_biotope` looks for them in the configuration folder of your VVV installation), clicking on one of them loads the configuration. When the document (configuration) is loaded, a vivid is synthesised based on it. The vivid is dissolved when the document is closed.

8.1.1 Faces and Visages

The ViViVerse knows two types of user interfaces: while we have already introduced Visages – the end-user-HMIs, let us now present Faces. To explain the difference between those two, let us take a look at the hardware world:

In control cabinets (see Figure 13), devices of different manufacturers are mounted and provide access to extended or reduced functionality in manufacturer-specific style and format. The control room, on the other hand, provides access to the whole functionality of the system in a coherent layout.

Faces are like the user interfaces of the devices in the control cabinet: they are directly attached to one organelle and there is only one face per organelle (it can be opened more than once simultaneously though). Visages can be connected to any number of organelles (also in other vivids), and the functionality they provide is rather related to the entire system instead of single organelles – unless e.g., devices need to be configured. And with Visages being organelles themselves, they may also have Faces.

Figure 13 A control cabinet and a control room demonstrate the relationship between Faces and Visages.

8.1.2 Loading configurations and defining the document window layout

When a configuration is loaded for the first time, initially the document window is empty, except for the sentence: 'Click here to start layout'. When doing so, you switch into layout mode in which you can define the contents of the document window.

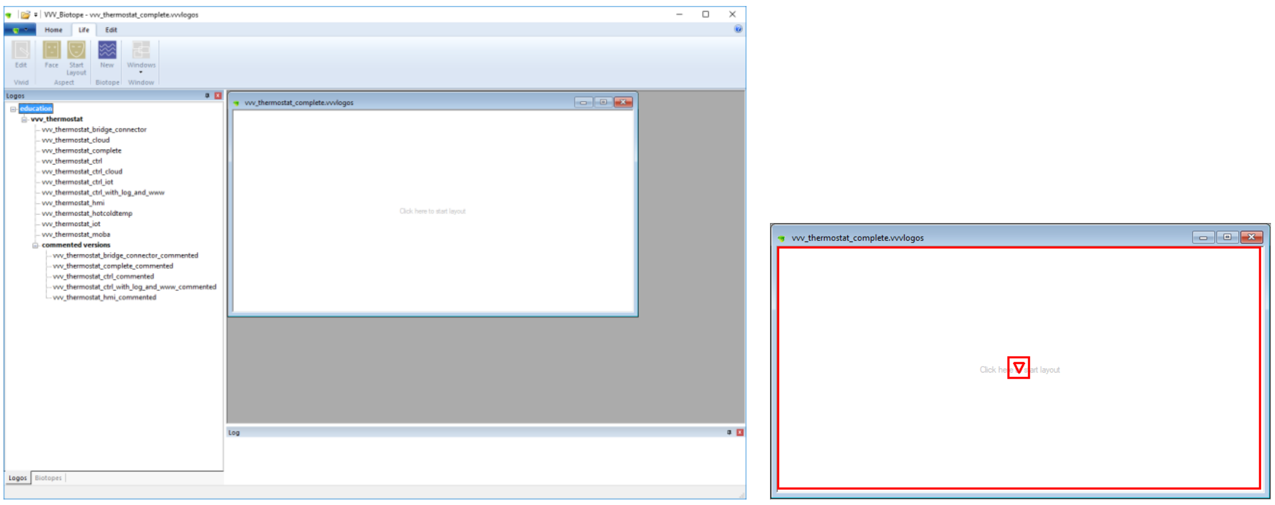

VVV_Biotope - vvv_thermostat_complete.vvvlogos
Home
Life
Edit
Edit
Face
Start layout
New
Windows
Vivid
Aspect
Biotope
Window
Logos
education
vvv_thermostat
vvv_thermostat_bridge_connector
vvv_thermostat_cloud
vvv_thermostat_complete
vvv_thermostat_ctrl
vvv_thermostat_ctrl_cloud
vvv_thermostat_ctrl_iot
vvv_thermostat_ctrl_with_log_and_www
vvv_thermostat_hmi
vvv_thermostat_hotcoldtemp
vvv_thermostat_iot
vvv_thermostat_moba
commented versions
vvv_thermostat_bridge_connector_commented
vvv_thermostat_complete_commented
vvv_thermostat_ctrl_commented
vvv_thermostat_ctrl_with_log_and_www_commented
vvv_thermostat_hmi_commented
vvv_thermostat_complete.vvvlogos
Click here to start layout
Log
Logos
Biotopes
vvv_thermostat_complete.vvvlogos
Click here to start layout

Figure 14 The vvv_biotope host program after loading the vvv_thermostat_complete configuration and the document window after switching into layout mode.

The window area can be split into any number of rectangles in which either Faces or Visages (together known as 'aspects') can be displayed.

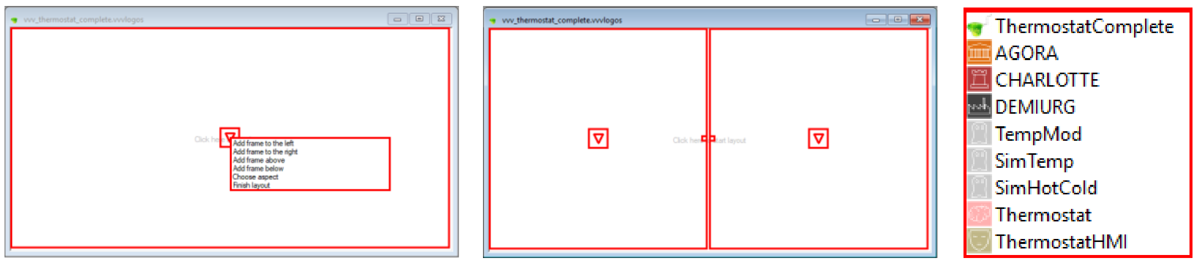


Figure 15 The document window with the layout menu and after a second frame has been added. Clicking choose aspect displays the aspect menu.

The layout of the faces and Visages in the document windows is stored in the same folder as the respective configuration, e.g., `vvv_thermostat_complete__Biotope.xml`.

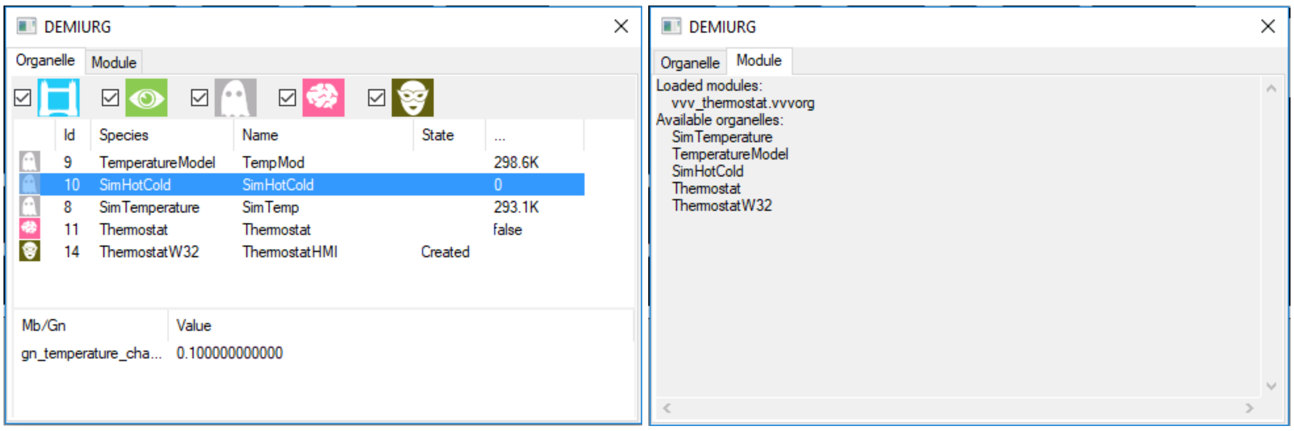
The layout mode can be finished/restarted with the respective button in the ribbon.

If you want to quickly see a Face provided by one of the organelles in the vivid, use the 'Face' ribbon item .

Now we explore the Faces of the three core organelles: Demiurg, Agora and Chaperone. These Faces do help a lot when diagnosing and solving problems. Faces can have one or more 'panes' in which they can display information. One Face can be opened more than once if the information of more than one pane shall be displayed at the same time.

8.1.3 The Demiurg Face

The Demiurg Face displays all synthesised pheno organelles (i.e., the core organelles are not displayed) in the first pane and all loaded organelle modules and available organelle species in the second pane. In fact, this is the only way to find out, which organelles an organelle module contains.



Id	Species	Name	State	...
9	TemperatureModel	TempMod		298.6K
10	SimHotCold	SimHotCold		0
8	SimTemperature	SimTemp		293.1K
11	Thermostat	Thermostat		false
14	ThermostatW32	ThermostatHMI	Created	

Mb/Gn	Value
gn_temperature_cha...	0.100000000000

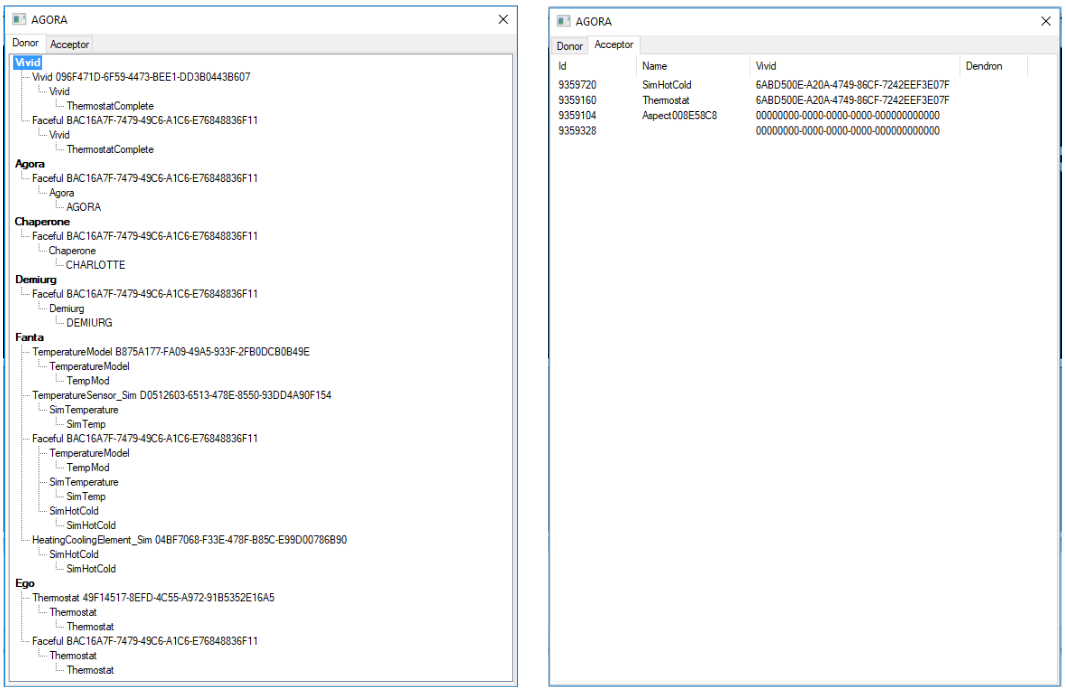
```
Loaded modules:
   vvv_thermostat.vvvorg
Available organelles:
   SimTemperature
   TemperatureModel
   SimHotCold
   Thermostat
   ThermostatW32
```

Figure 16 The two panes of the Demiurg Face.

The organelle pane displays - for each synthesised organelle - its class, id, species, name, and state. The last column shows the value of the first metabolic which the organelle provides. If there are no metabolics, the first gene is displayed. Other metabolics and genes are displayed at the bottom of the pane when the organelle is selected. The pane is dynamic, i.e., when organelles are synthesised or dissolved while it is open, this change is reflected.

8.1.4 The Agora Face

The Agora Face displays all registered donors in the first pane and all registered acceptors in the second pane.


AGORA

Donor | Acceptor

```
Vivid
  Vivid 096F471D-6F59-4473-BEE1-DD3B0443B607
    Vivid
      ThermostatComplete
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Vivid
      ThermostatComplete
Agora
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Agora
      AGORA
Chaperone
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Chaperone
      CHARLOTTE
Demiurg
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Demiurg
      DEMIURG
Fanta
  TemperatureModel B875A177-FA09-49A5-933F-2FB0DCB0B49E
    TemperatureModel
      TempMod
  TemperatureSensor_Sim D0512603-6513-478E-8550-93DD4A90F154
    SimTemperature
      SimTemp
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    TemperatureModel
      TempMod
    SimTemperature
      SimTemp
    SimHotCold
      SimHotCold
  HeatingCoolingElement_Sim 04BF7068-F33E-478F-B85C-E99D00786B90
    SimHotCold
      SimHotCold
Ego
  Thermostat 49F14517-8EFD-4C55-A972-91B5352E16A5
    Thermostat
      Thermostat
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Thermostat
      Thermostat
```

AGORA

Donor | Acceptor

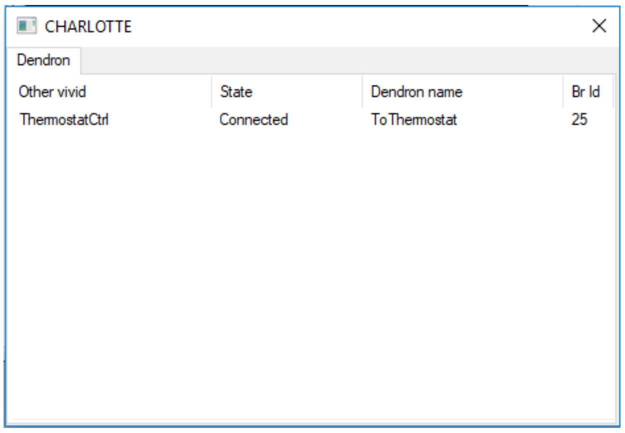
Id	Name	Vivid	Dendron
9359720	SimHotCold	6ABD500E-A20A-4749-86CF-7242EEF3E07F	
9359160	Thermostat	6ABD500E-A20A-4749-86CF-7242EEF3E07F	
9359104	Aspect008E58C8	00000000-0000-0000-0000-000000000000	
9359328		00000000-0000-0000-0000-000000000000	

Figure 17 The two panes of the Agora Face.

One organelle can be registered more than once if it belongs to more than one family. Organelles which have been synthesised in another vivid but have also registered in this vivid through ectoviv donor queries or pheromones also show the dendron through which they were registered. The acceptor pane displays the id (this is not the organelle id), the name and the vivid which contains the acceptor. If the acceptor comes from another vivid, the dendron through which this vivid is connected, is also displayed. The acceptors which do not belong to any vivid (zero guid), are part of the `vvv_biotope` program.

8.1.5 The Chaperone Face

The Chaperone Face shows the other vivids to which the vivid is connected.


CHARLOTTE

Dendron

Other vivid	State	Dendron name	Br Id
ThermostatCtrl	Connected	ToThermostat	25

Figure 18 The Chaperone Face.

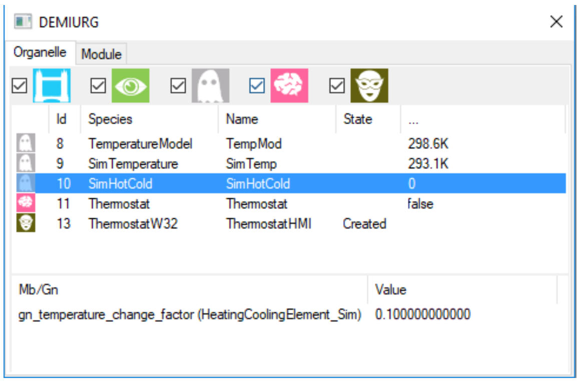
The Chaperone Face displays for each existing dendron the name of the vivid on the other side, the dendron state and name and the organelle id of the bridge used by the dendron. This id can be used to identify the bridge in the Demiurg Face.

8.2 The integrated Thermostat version

This is the simplest and most straightforward version: the configuration `vvv_thermostat_complete.vvvdna` contains all the five organelles which are needed for the full thermostat functionality:

- The temperature sensor
- The temperature-model
- The heating/cooling element
- The thermostat control loop
- The user-interface

All these organelles are loaded into one single vivid on synthesis, which means that they all run on the same computer.



	Id	Species	Name	State	...
	8	TemperatureModel	TempMod		298.6K
	9	SimTemperature	SimTemp		293.1K
	10	SimHotCold	SimHotCold		0
	11	Thermostat	Thermostat		false
	13	ThermostatW32	ThermostatHMI	Created	

Mb/Gn	Value
gn_temperature_change_factor (HeatingCoolingElement_Sim)	0.100000000000

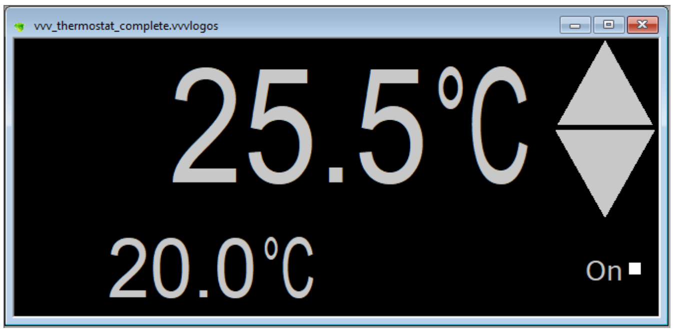
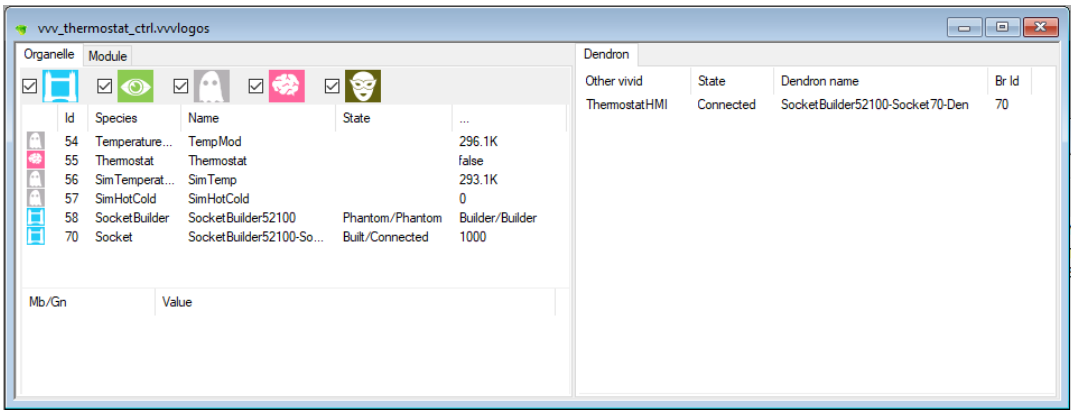


Figure 19 The Demiurg Face and the document window of the vvv_thermostat_complete configuration showing the Visage of the Thermostat application.

8.3 Control loop and HMI separated

When it is required to have the user-interface separated from the control loop, the HMI organelle must be moved into a separate configuration and thus another vivid. This means that the symbioses between the HMI and the other organelles must be performed through a dendron which uses a bridge for communication.

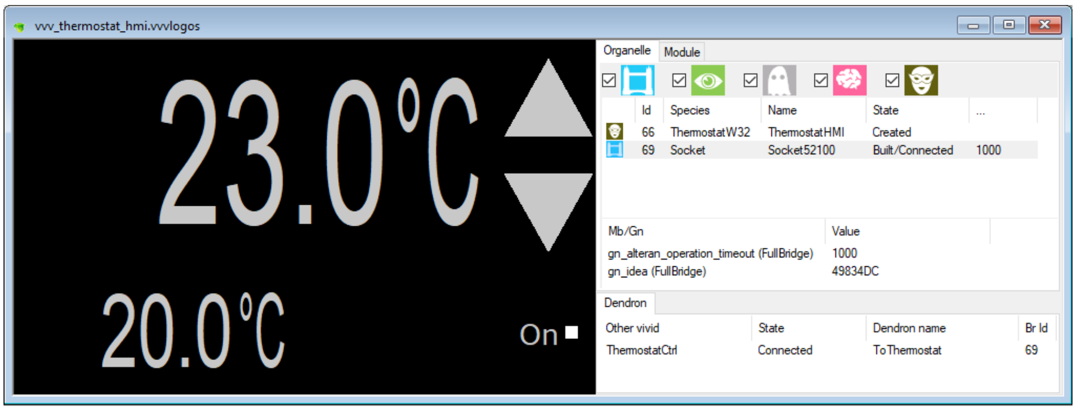
The active side in this setup is the HMI which connects to the control loop vivid. The control loop contains the `SocketBuilder` organelle waiting for socket connection attempts. Any number of HMIs can be connected to the control loop.


vvv_thermostat_ctrl.vvvlogos

Id	Species	Name	State	...
54	Temperature...	TempMod		296.1K
55	Thermostat	Thermostat		false
56	SimTemperat...	SimTemp		293.1K
57	SimHotCold	SimHotCold		0
58	SocketBuilder	SocketBuilder52100	Phantom/Phantom	Builder/Builder
70	Socket	SocketBuilder52100-So...	Built/Connected	1000

Mb/Gn	Value

Other vivid	State	Dendron name	Br Id
ThermostatHMI	Connected	SocketBuilder52100-Socket70-Den	70


vvv_thermostat_hmi.vvvlogos

Id	Species	Name	State	...
66	ThermostatW32	ThermostatHMI	Created	
69	Socket	Socket52100	Built/Connected	1000

Mb/Gn	Value
gn_alteran_operation_timeout (FullBridge)	1000
gn_idea (FullBridge)	49834DC

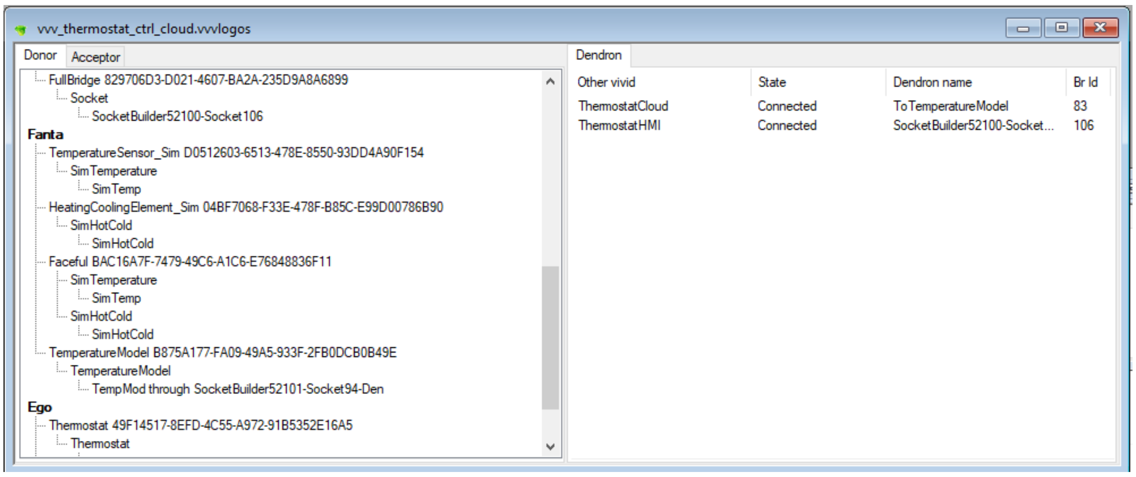
Other vivid	State	Dendron name	Br Id
ThermostatCtrl	Connected	ToThermostat	69

Figure 20 The document windows of the vvv_thermostat_ctrl (above) and vvv_thermostat_hmi (below) configurations. Each document window represents one vivid.

Figure 20 shows various Faces and the Visage in the split thermostat application. In comparison with the integrated setup, the HMI organelle is missing in the control vivid. It has been replaced by the `SocketBuilder` organelle and the `Socket` bridge, which is only synthesised while a connection with the HMI vivid has been established. The HMI vivid only has two organelles: the `ThermostatW32` Visage and the `Socket` bridge. The Chaperone Faces in both document windows show to which vivids the vivid is currently connected.

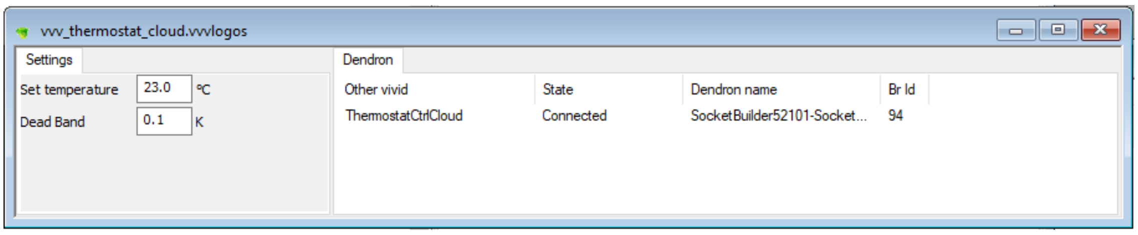
8.4 Cloud computing

Cloud computing, in this context, means the outsourcing of computing resources to other vivids. This, of course, only makes sense, if the communication reliability and performance needs of your application are met. In the Thermostat example, the temperature model is moved into its own vivid and the control loop vivid reaches out for it to find a member of the `TemperatureModel` family. When found, the `TemperatureModel` (the species has the same name as the family) fanta is registered in the Agora of the control loop vivid.


vvv_thermostat_ctrl_cloud.vvvlogos

```
  FullBridge 829706D3-D021-4607-BA2A-235D9A8A6899
    Socket
      SocketBuilder52100-Socket106
Fanta
  TemperatureSensor_Sim D0512603-6513-478E-8550-93DD4A90F154
    SimTemperature
      SimTemp
  HeatingCoolingElement_Sim 04BF7068-F33E-478F-B85C-E99D00786B90
    SimHotCold
      SimHotCold
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    SimTemperature
      SimTemp
    SimHotCold
      SimHotCold
  TemperatureModel B875A177-FA09-49A5-933F-2FB0DCB0B49E
    TemperatureModel
      TempMod through SocketBuilder52101-Socket94-Den
Ego
  Thermostat 49F14517-8EFD-4C55-A972-91B5352E16A5
    Thermostat
```

Other vivid	State	Dendron name	Br Id
ThermostatCloud	Connected	ToTemperatureModel	83
ThermostatHMI	Connected	SocketBuilder52100-Socket...	106



vvv_thermostat_cloud.vvvlogos

Settings

Set temperature	23.0	°C
Dead Band	0.1	K

Dendron

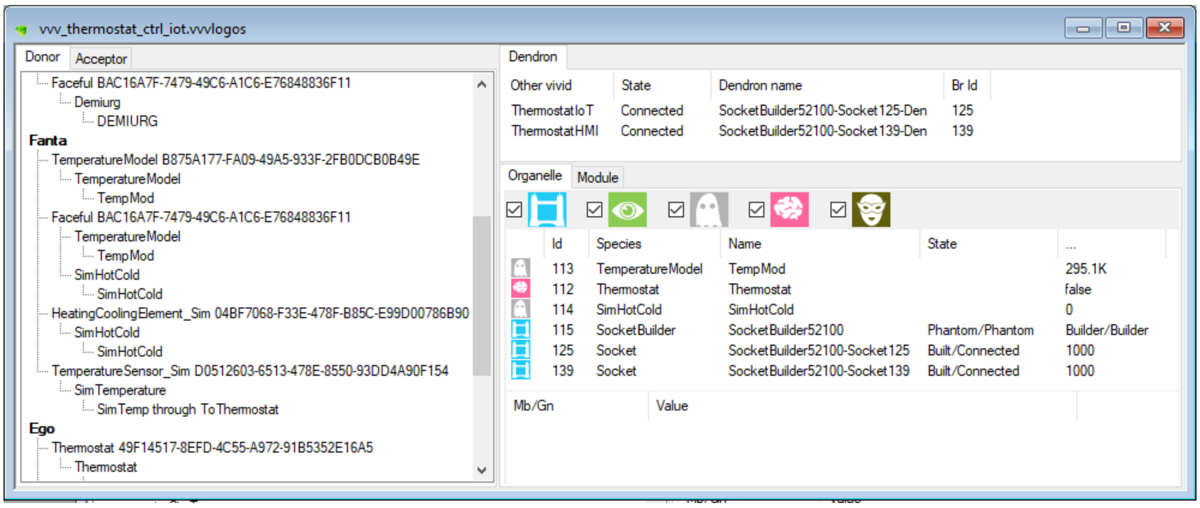
Other vivid	State	Dendron name	Br Id
ThermostatCtrlCloud	Connected	SocketBuilder52101-Socket...	94

Figure 21 The Agora and Chaperone Faces of the ThermostatCtrlCloud vivid (above) and the `TemperatureModel` and Chaperone Faces of the ThermostatCloud vivid (below) in the document windows of their respective configurations.

The Chaperone Face of the control-loop-vivid shows that the HMI is connected as well. However, it is not shown here because it does not differ from the previous example.

8.5 The IoT version

The IoT version of the Thermostat example has the temperature sensor in its own vivid, by this decoupling the location of the sensor device from the application device.



vvv_thermostat_ctrl_iot.vvvlogos

Donor | Acceptor

```
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Demiurg
      DEMIURG
Fanta
  TemperatureModel B875A177-FA09-49A5-933F-2FB0DCB0B49E
    TemperatureModel
      TempMod
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    TemperatureModel
      TempMod
    SimHotCold
      SimHotCold
  HeatingCoolingElement_Sim 04BF7068-F33E-478F-B85C-E99D00786B90
    SimHotCold
      SimHotCold
  TemperatureSensor_Sim D0512603-6513-478E-8550-93DD4A90F154
    SimTemperature
      SimTemp through ToThermostat
Ego
  Thermostat 49F14517-8EFD-4C55-A972-91B5352E16A5
    Thermostat
```

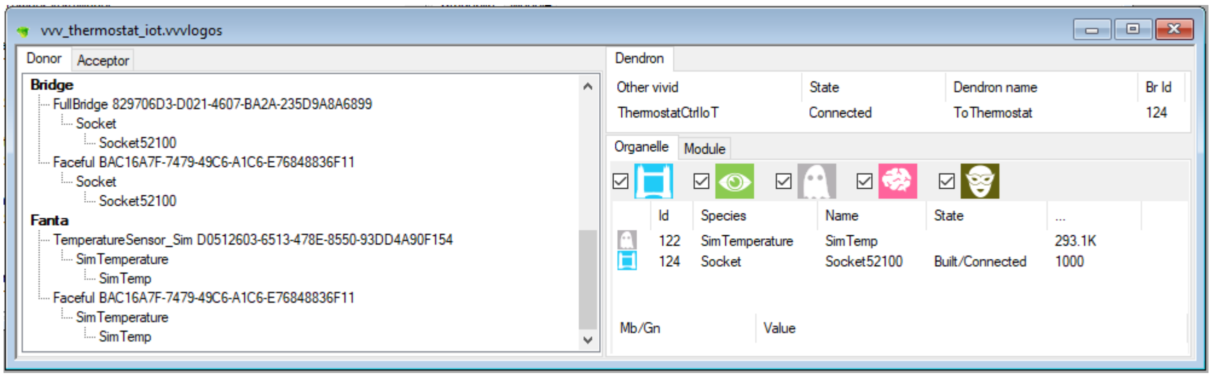
Dendron

Other vivid	State	Dendron name	Br Id
ThermostatIoT	Connected	SocketBuilder52100-Socket125-Den	125
ThermostatHMI	Connected	SocketBuilder52100-Socket139-Den	139

Organelle | Module

Id	Species	Name	State	...
113	TemperatureModel	TempMod		295.1K
112	Thermostat	Thermostat		false
114	SimHotCold	SimHotCold		0
115	SocketBuilder	SocketBuilder52100	Phantom/Phantom	Builder/Builder
125	Socket	SocketBuilder52100-Socket125	Built/Connected	1000
139	Socket	SocketBuilder52100-Socket139	Built/Connected	1000

Mb/Gn	Value



vvv_thermostat_iot.vvvlogos

Donor | Acceptor

```
Bridge
  FullBridge 829706D3-D021-4607-BA2A-235D9A8A6899
    Socket
      Socket52100
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    Socket
      Socket52100
Fanta
  TemperatureSensor_Sim D0512603-6513-478E-8550-93DD4A90F154
    SimTemperature
      SimTemp
  Faceful BAC16A7F-7479-49C6-A1C6-E76848836F11
    SimTemperature
      SimTemp
```

Dendron

Other vivid	State	Dendron name	Br Id
ThermostatCtrlIoT	Connected	ToThermostat	124

Organelle | Module

Id	Species	Name	State	...
122	SimTemperature	SimTemp		293.1K
124	Socket	Socket52100	Built/Connected	1000

Mb/Gn	Value

Figure 22 The Faces of the ThermostatCtrlIoT (above) and ThermostatIoT (below) vivids in the document windows of their respective configurations.

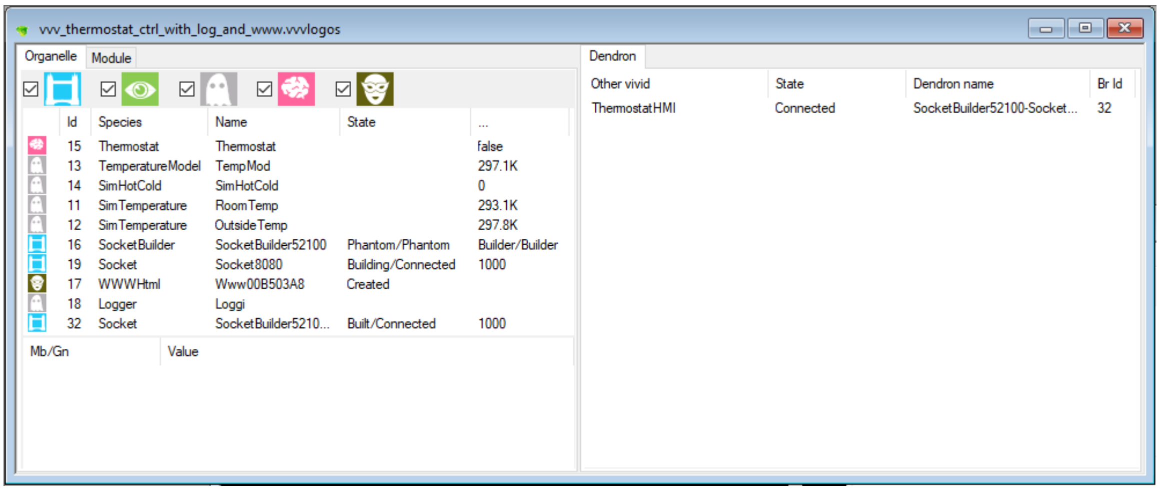
As opposed to the cloud example, this time it is not the control loop vivid which connects to the 'other' vivid, but the iot vivid, the reason for this being that the 'mobile' partner cannot be reached (because prevented by the carriers) and therefore must connect to the application itself.

Again, it can be seen, that the HMI vivid is also connected to the control loop vivid. It is not shown here for the same reasons as in the last chapter.

8.6 The augmented version

One of the shining features of the VVV is the ability to extend the functionality of applications without changing the source code. Suppose you wanted to add logging of not only the room temperature but also the outside temperature. As additional goody, you want to see these values in a browser. Which additional organelles would you need?

- Another temperature sensor (mounted outside)
- A configurable logging facility
- A configurable web server



vvv_thermostat_ctrl_with_log_and_www.vvvlogos

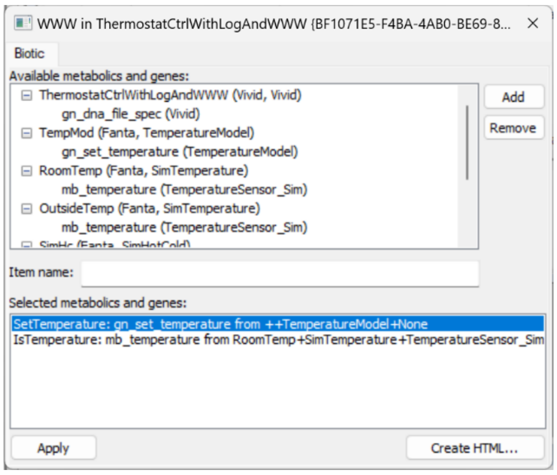
	Id	Species	Name	State	...
	15	Thermostat	Thermostat		false
	13	TemperatureModel	TempMod		297.1K
	14	SimHotCold	SimHotCold		0
	11	SimTemperature	RoomTemp		293.1K
	12	SimTemperature	OutsideTemp		297.8K
	16	SocketBuilder	SocketBuilder52100	Phantom/Phantom	Builder/Builder
	19	Socket	Socket8080	Building/Connected	1000
	17	WWWHtml	Www00B503A8	Created	
	18	Logger	Loggi		
	32	Socket	SocketBuilder5210...	Built/Connected	1000

Mb/Gn	Value

Other vivid	State	Dendron name	Br Id
ThermostatHMI	Connected	SocketBuilder52100-Socket...	32

Figure 23 The Demiurg and Chaperone Faces of the ThermostatCtrlWithAndWWW vivid.

In the above figure, the socket with id 19 is the one opened by web server Visage, while the socket with id 32 is the one opened by the socket builder when the HMI has connected.



WWW in ThermostatCtrlWithLogAndWWW {BF1071E5-F4BA-4AB0-BE69-8...

Biotic

Available metabolics and genes:

```
⊟ ThermostatCtrlWithLogAndWWW (Vivid, Vivid)
      gn_dna_file_spec (Vivid)
⊟ TempMod (Fanta, TemperatureModel)
      gn_set_temperature (TemperatureModel)
⊟ RoomTemp (Fanta, SimTemperature)
      mb_temperature (TemperatureSensor_Sim)
⊟ OutsideTemp (Fanta, SimTemperature)
      mb_temperature (TemperatureSensor_Sim)
⊟ SimHc (Fanta, SimHotCold)
```

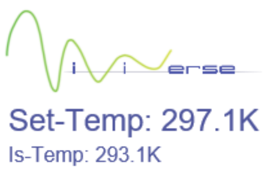
Item name:

Selected metabolics and genes:

```
SetTemperature: gn_set_temperature from ++TemperatureModel+None
IsTemperature: mb_temperature from RoomTemp+SimTemperature+TemperatureSensor_Sim
```

Figure 24 The Face of the www_html organelle. In it, the metabolics and genes in the generated html pages can be configured.

The web server can be configured via its Face. It is even possible to create basic web pages directly, thus publishing information within a minute without any programming knowhow.



Set-Temp: 297.1K

Is-Temp: 293.1K

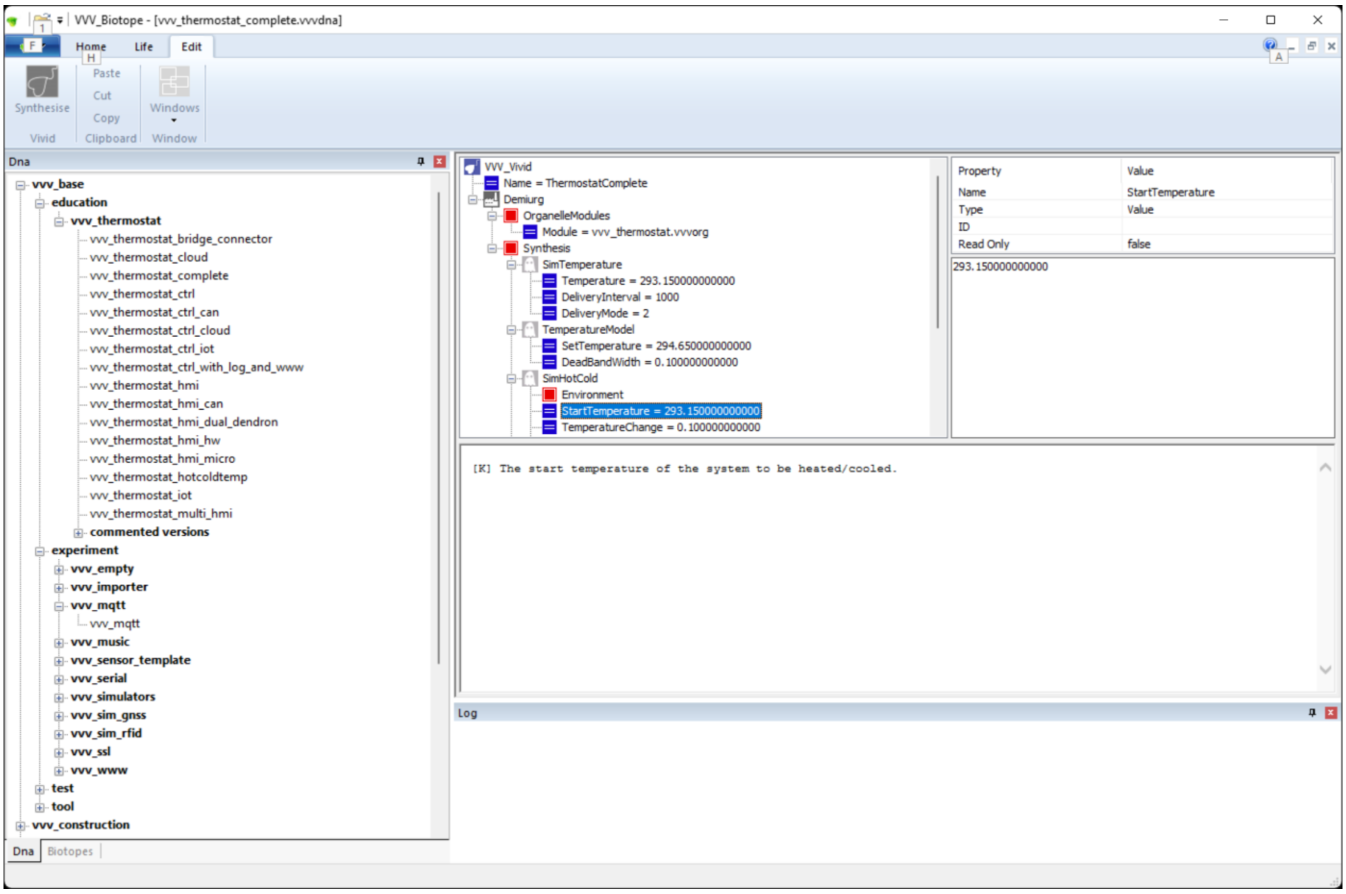
Figure 25 The browser output of the web server Visage after connecting the browser to [http://localhost:8080/.](http://localhost:8080/)

Because the web server is very simple, it cannot do conversions. Therefore, the temperature is displayed in the internal unit, which is Kelvin. Custom conversions and formatting, however, could be done from within the web page.

The logger collects configurable values (metabolics and genes) and writes them to a text file in the folder `runtime\quality\production`.

8.7 Editing DNAs

The `vvv_biotope` host program can edit configurations in place and then directly use them. This way, it is also possible to create a new configuration and immediately use and test it.



```
vvv_base
  education
    vvv_thermostat
      vvv_thermostat_bridge_connector
      vvv_thermostat_cloud
      vvv_thermostat_complete
      vvv_thermostat_ctrl
      vvv_thermostat_ctrl_can
      vvv_thermostat_ctrl_cloud
      vvv_thermostat_ctrl_iot
      vvv_thermostat_ctrl_with_log_and_www
      vvv_thermostat_hmi
      vvv_thermostat_hmi_can
      vvv_thermostat_hmi_dual_dendron
      vvv_thermostat_hmi_hw
      vvv_thermostat_hmi_micro
      vvv_thermostat_hotcoldtemp
      vvv_thermostat_iot
      vvv_thermostat_multi_hmi
      commented versions
  experiment
    vvv_empty
    vvv_importer
    vvv_mqtt
      vvv_mqtt
    vvv_music
    vvv_sensor_template
    vvv_serial
    vvv_simulators
    vvv_sim_gnss
    vvv_sim_rfid
    vvv_ssl
    vvv_www
  test
  tool
vvv_construction
```

```
VVV_Vivid
  Name = ThermostatComplete
  Demiurg
    OrganelleModules
      Module = vvv_thermostat.vvvorg
    Synthesis
      SimTemperature
        Temperature = 293.150000000000
        DeliveryInterval = 1000
        DeliveryMode = 2
      TemperatureModel
        SetTemperature = 294.650000000000
        DeadBandWidth = 0.100000000000
      SimHotCold
        Environment
        StartTemperature = 293.150000000000
        TemperatureChange = 0.100000000000
```

Property	Value
Name	StartTemperature
Type	Value
ID	
Read Only	false

```
293.150000000000
```

```
[K] The start temperature of the system to be heated/cooled.
```

Figure 26 A vivid in Edit mode.

Based on XML schema definitions, the editor provides support for editing.

8.8 More Ideas: multiple HMIs, MQTT and more

Instead of the simulators used in the examples so far, real sensors and actuators could be used.

A second HMI could be added to the configuration which could display weather data – measured locally or downloaded from the internet.

Instead of the web server implemented by the organelle `www_hmtl`, the organelle `mqtt` could be added to the vivid, by this adding the functionality to publish metabolics and genes to an MQTT broker. The Face of the organelle `mqtt` is almost identical to the one of `www_html`, allowing configuration without programming knowhow.

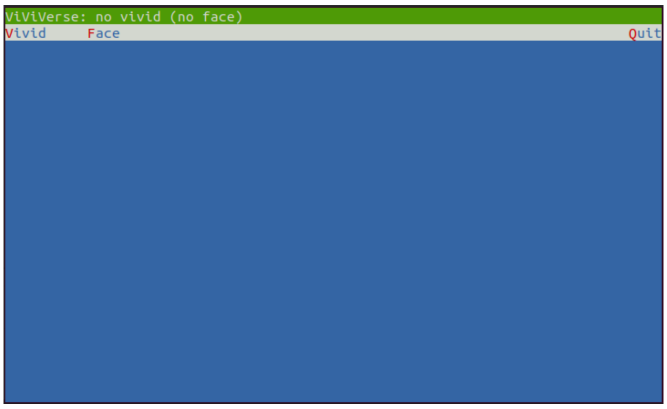
Using the available interoperability layer (C and .NET), it is possible to integrate the ViViVerse into other software like MS Office or MatLab.

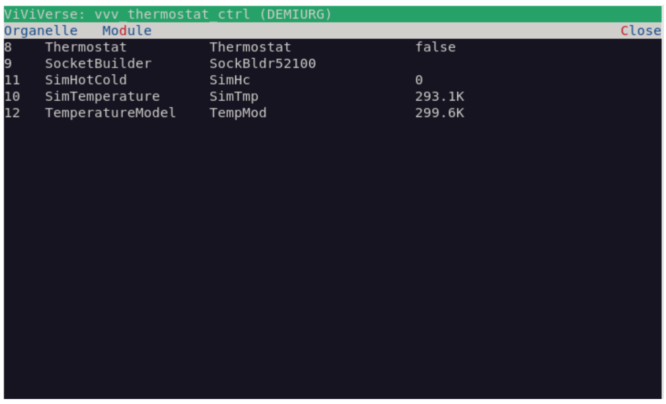
9 Appendix B: Linux

In chapter 8, we have seen a lot of screenshots, mostly taken from the `vvv_biotope` host program (the Faces are not tied to the host program and can be reused). This software is based on MFC and thus cannot be used on Linux and other platforms.

9.1 ncurses

For Linux, the `ncurses` library has been used to create the `vvv_console` host program and Faces. The graphical capabilities of this approach are very limited though.





```
ViViVerse: vvv thermostat ctrl (DEMIURG)
Organelle   Module                                                Close
8    Thermostat        Thermostat                   false
9    SocketBuilder     SockBldr52100
11   SimHotCold        SimHc                        0
10   SimTemperature    SimTmp                       293.1K
12   TemperatureModel  TempMod                      299.6K
```

Figure 27 The vvv_console host program and the Demiurg Face.

One huge advantage is the low bandwidth needed for remote support via SSH.

9.2 Other visual frameworks

Using Qt for a platform independent take on the implementation of Faces and Visages has been considered but discarded because of the delicate nature of the Qt licensing policy. Thus, Qt Faces have not yet been implemented.

If Qt shall be used for Visages, the `vvv_qt` package supports their development. It has already been used in several projects.

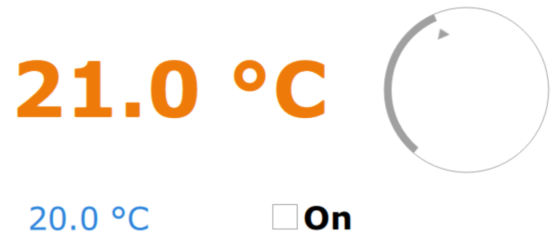


Figure 28 A Visage for the Thermostat application based on Qt. It can run on Linux and Windows.

Other possibilities include the use of HTML or .NET based frameworks, which have not been implemented yet.

10 Appendix C: The ViViVerse dictionary

Organelle

Organelles are the building blocks of the ViViVerse. The **Pheno** Organelles provide the actual functionality to the real world. The **Core** Organelles are present in each Vivid and create the Pheno Organelles and manage their Symbioses.

Biotic

Biotics are the components of the functionality provided in the ViViVerse: **Commands**, **Metabolics** and **Genes**.

Donor

A Donor is a **Symbiont** which provides Biotics to others.

Acceptor

An Acceptor is a Symbiont which uses the Biotics provided by a Donor.

Symbiosis

A Symbiosis is the relationship between a Donor and an Acceptor. If they are in the same Vivid, the Symbiosis is **endoviv**. If they are in different Vivids, the Symbiosis is **ectoviv**.

Dendron

The Dendron links Vivids and enables Symbioses across the whole ViViVerse.

Demiurg

The Demiurg is the generic Organelle factory. It is one of the three Core Organelles present in each Vivid.

Agora

The Agora is the marketplace where Symbionts register and find each other. It is one of the three Core Organelles present in each Vivid.

Chaperone

The Chaperone is responsible for the security within a Vivid by supervising the Dendrons. It is one of the three Core Organelles present in each Vivid.

Face

Faces are optional HMIs bound to an Organelle. They are displayed as part of the host application.

Taxon

A Taxon is a label which helps classifying Organelles in a structured way.

Class

The Class is the Taxon which defines the category of an Organelle. There are 5 Classes: **Bridge**, **Rebus**, **Fanta**, **Ego**, **Visage**.

Family

The Species is the Taxon which defines a specific implementation of a Family.

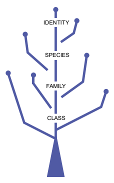
Species

The Species is the Taxon which defines a specific implementation of a Family.

Identity

The Identity is the Taxon which identifies a single specimen in the ViViVerse.

Holotaxon



The Holotaxon is a unique combination of a Class, a Family, a Species, and an Identity Taxon.

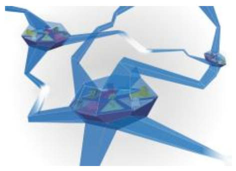
VVVDNA

The VVVDNA holds the information which is used to **synthesise** a Vivid and its Organelles.

Vivid

The Vivid contains the Core Organelles, Dendrons and Pheno Organelles. It is the smallest fully operational unit in the ViViVerse.

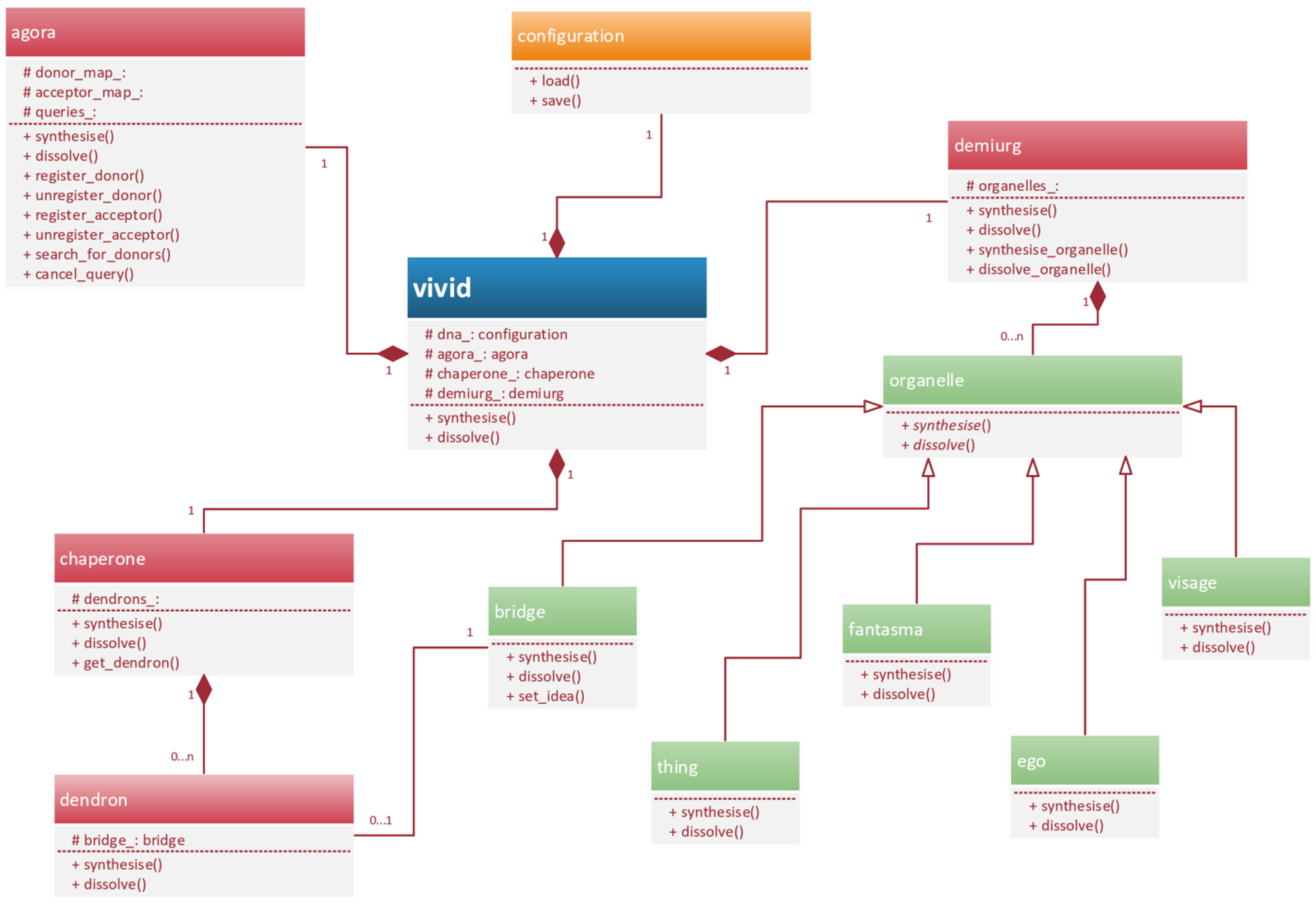
ViViVerse



The ViViVerse is the sum of all Vivids and the Symbioses between them.

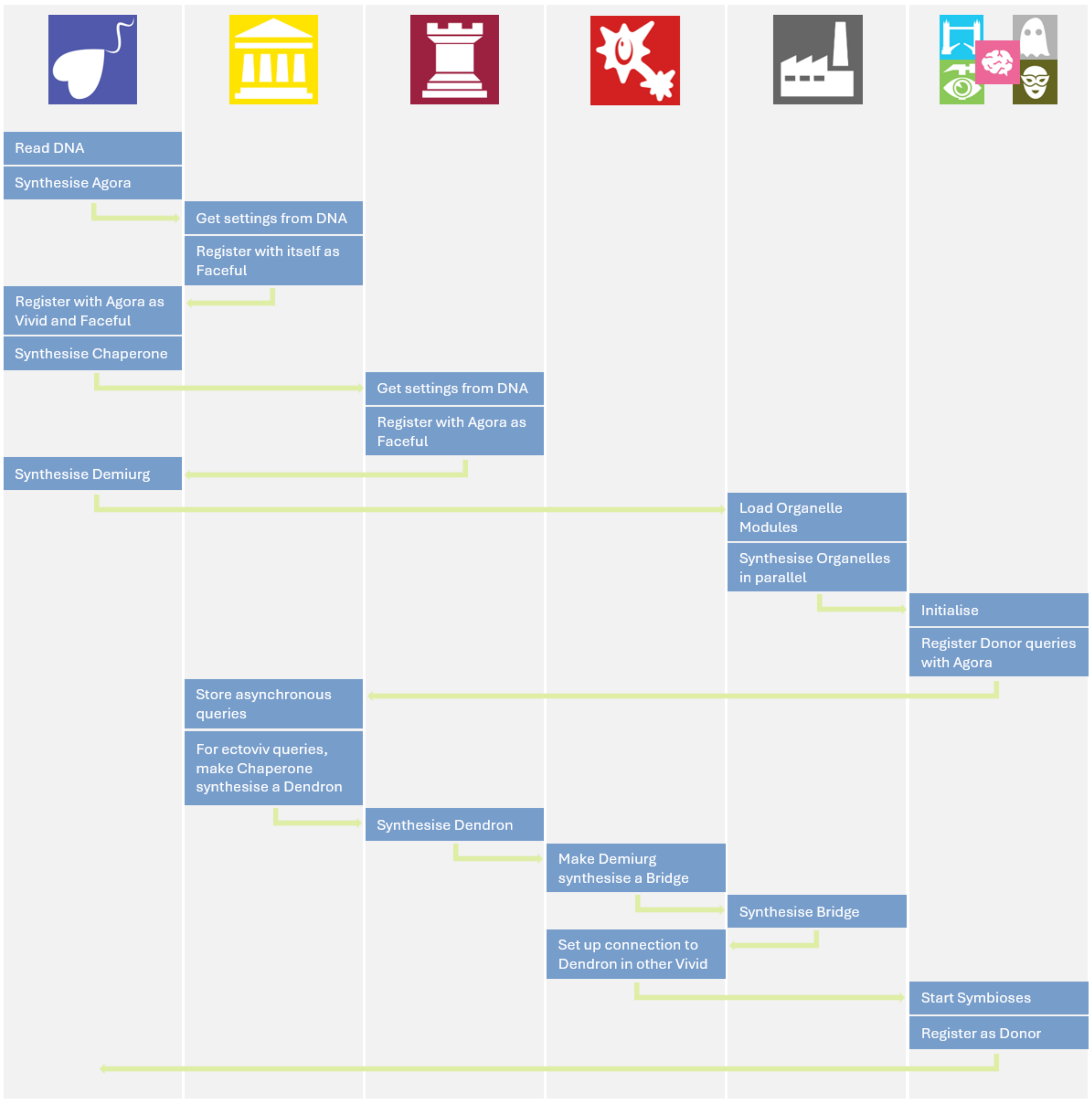
11 Appendix D: C++ Class diagram

The following diagram shows the main classes of the ViViVerse framework and their relationships.



12 Appendix E: Vivid synthesis

The following diagram shows what happens during the synthesis of a Vivid.


Read DNA
Synthesise Agora
Get settings from DNA
Register with itself as Faceful
Register with Agora as Vivid and Faceful
Synthesise Chaperone
Get settings from DNA
Register with Agora as Faceful
Synthesise Demiurg
Load Organelle Modules
Synthesise Organelles in parallel
Initialise
Register Donor queries with Agora
Store asynchronous queries
For ectoviv queries, make Chaperone synthesise a Dendron
Synthesise Dendron
Make Demiurg synthesise a Bridge
Synthesise Bridge
Set up connection to Dendron in other Vivid
Start Symbioses
Register as Donor