Design Documentation

Release 8.0

The AMUSE Team

May 23, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

In this document we will describe the high level design of AMUSE. During the development period of AMUSE this document will be a “Work in Progress”. It will be updated to state latests idead about the design and reflect the current implementation. More detailed documentation can be found in the reference documentation

1.1 AMUSE

AMUSE combines existing astrophysical numerical codes into a single system.

1.1.1 Goal

To develop a toolbox for performing numerical astrophysical experiments. The toolbox provides:

- A standard way of input and output of astrophysical data.
- Support for set-up and management of numerical experiments.
- A unique method to couple a wide variety of physical codes
- A legacy set of standard, proven codes. These codes will be integrated into AMUSE as modules. Each module can be used stand-alone or in combination with other modules
- A standard way for adding new modules to AMUSE.
- Examples to show the use of each module and possible couplings between the modules.
- Documentation containing introduction, howto’s and reference documents.

1.1.2 Development

AMUSE development is planned to take place in a 2.5 year period. AMUSE is developed at the Leiden Observatory. The Leiden Observatory is a faculty of the Leiden University in the Netherlands. Funding is provided by a NOVA grant.

Universiteit Leiden
The Netherlands


Sterrewacht
Leiden


NOVA

CHAPTER

TWO

ARCHITECTURE OVERVIEW

2.1 Layers

The AMUSE architecture is based on a layered design with 3 layers. The highest layer is a python script, written for a single problem or set of problems. The next layer contains the AMUSE code, this layer provides a library of objects and function to use in the python script. The last layer contains all the existing or legacy codes. In this layer the physical models are implemented.

Each layer builds upon a lower layer, adding functionality or ease of use to the previous layer: Each layer has a

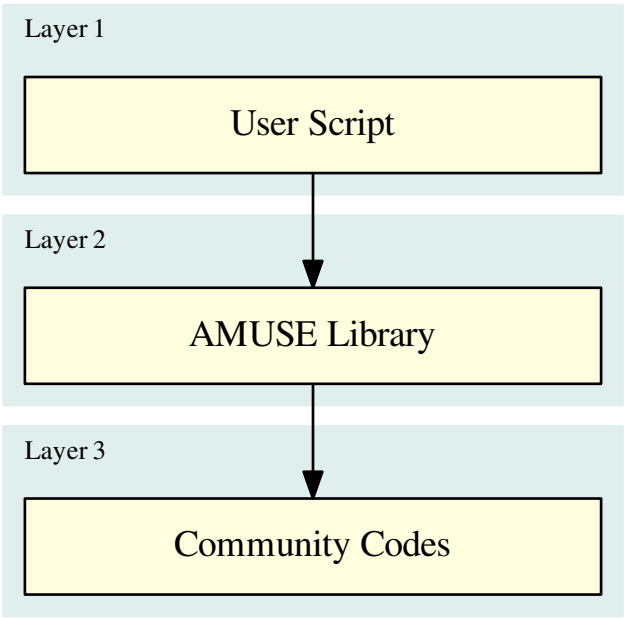


Figure 2.1: The 3 layers in AMUSE

different role in the AMUSE architecture:

1. **User Script layer**. The code in this layer implements a specific physical problem or set of problems. This layer contains the example scripts and scripts written by the user. This layer is conceptually comparable to

a User Interface layer in applications with a GUI. Coupling two or more codes happens in this layer (with the help of support classes from the *AMUSE Library Layer*.

2. **AMUSE Library layer**. This layer provides an object oriented interface on top of the legacy codes. It also provides a library of functionalities, such as unit handling and data conversion. The role of this layer is very generic, it is not specific for one problem or for one physical domain.
3. **Community Codes layer**. This layer defines the interfaces to the community codes and contains the actual codes. It provides process management for the community codes and functional interfaces to these. The code in this layer is generic in respect to problems, but specific for different physical domains.

The following sections contain a detailed explanation of the layers, starting with the lowest layer to the highest. Some details are further worked out in other chapters or in the reference manual.

2.1.1 Community codes layer

The **Community Codes layer** contains the actual applications and the functionality to communicate with these applications. This layer exposes every community code as a set of functions. These functions are grouped in one class per code.

The AMUSE framework code and the community codes are designed to be run as separate applications. The AMUSE framework code consists of a python script and the AMUSE library. The community codes consist of the original code-base of a scientific code extended with a new main application that handles messages send to it from the python library. Function calls into the community codes are send via a *message passing framework* to the actual running codes. The number of applications started and the machines on which these run can all be

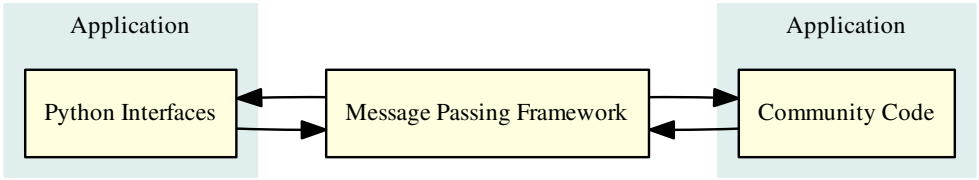
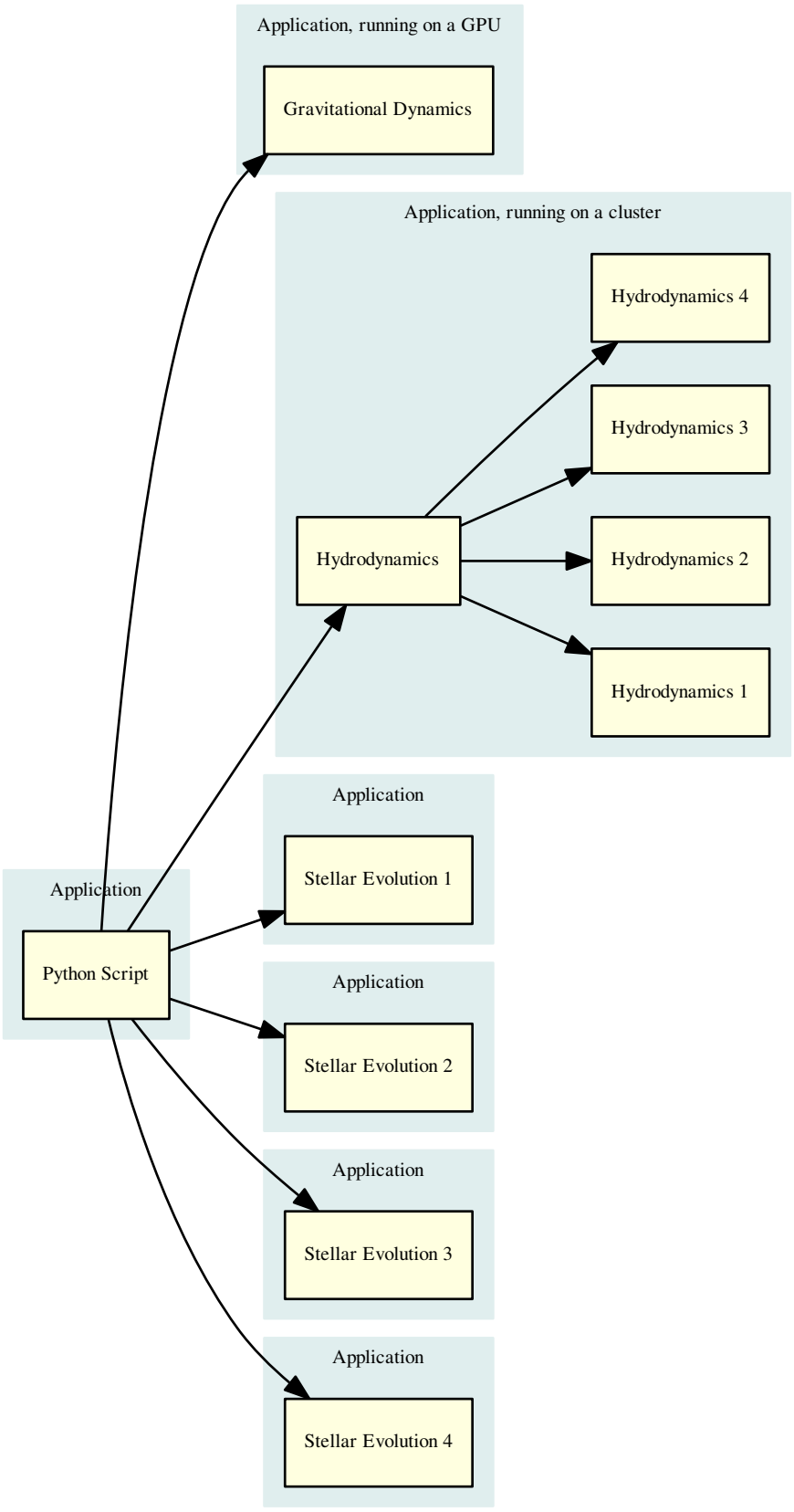


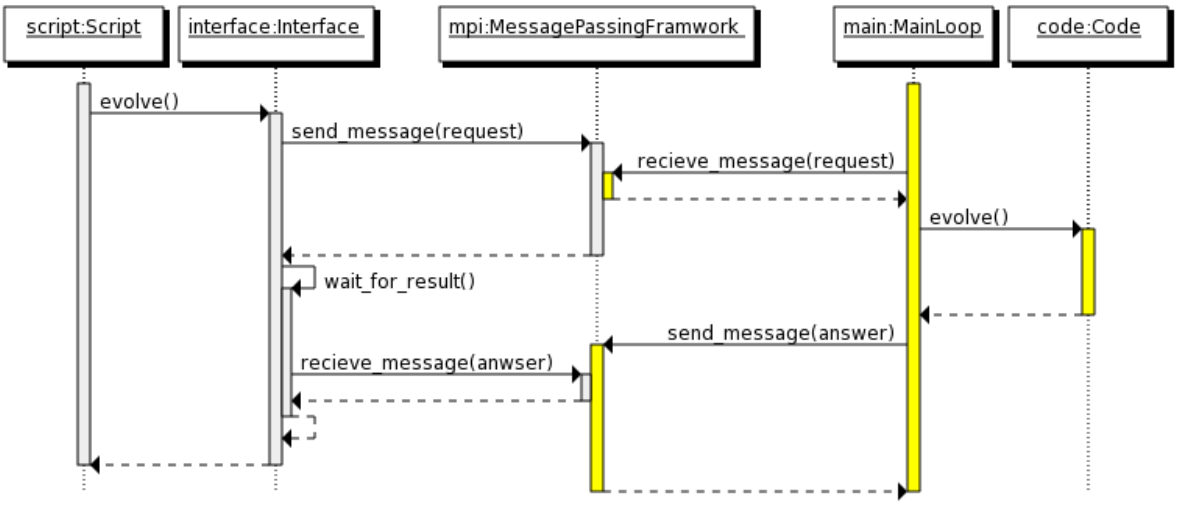
Figure 2.2: The AMUSE script and community codes are separate applications. The application communicate using a message passing framework

set dynamically in AMUSE. Depending on the problem a researcher can run all of AMUSE on a single desktop computer or in a mixed environment with clusters of computers. Every AMUSE run starts with one python script. This script can in turn start a number of different community codes (as separate applications). A complete run can consist of multiple applications running in parallel or in sequence and managed by one python script.


Application, running on a GPU
Gravitational Dynamics
Application, running on a cluster
Hydrodynamics 4
Hydrodynamics 3
Hydrodynamics
Hydrodynamics 2
Hydrodynamics 1
Application
Stellar Evolution 1
Application
Python Script
Application
Stellar Evolution 2
Application
Stellar Evolution 3
Application
Stellar Evolution 4

Message passing

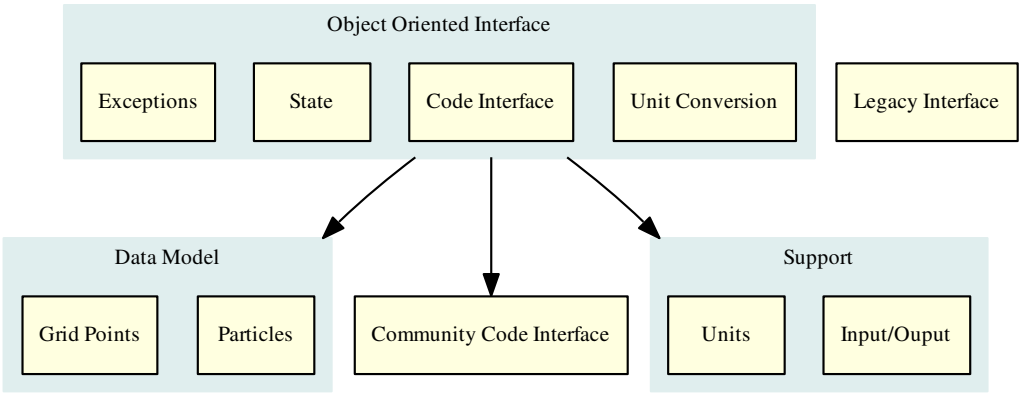
The amuse framework interacts with legacy codes via a message passing framework. Function calls in the python scripts are translated to messages and these messages are send to the community codes using the message passing framework . The community codes wait for message events and will decode the message upon arrival and perform the requested function. The results will be send back using a similar message.



2.1.2 AMUSE Library layer

The **Library layer** is responsible for providing an object oriented interface to the community codes. It also provides extra functionality to help write a user script, such as file input and output of common file formats and unit conversions. These extra functionalities can be used independent of the community codes.

Every community code has a *low-level* interface (defined in the community interface layer) and an *object-oriented* interface. The *low-level* interface is defined as as set of functions. The *object-oriented* interface uses these functions and combines these with models for state-transitions, units and data sets to provide an interface that is easier to use (less error prone) and easier to couple with other codes.



Model of a community code

The community codes of every module in all physical domains are modelled using the same template. The template consists of attributes and wrappers. **Attributes** provide a common interface for sub-parts of the code, for

example the *particles* attribute provides an interface to add, update and remove the particles in a code. Attributes combine several functions in a legacy interface into one object. **Wrappers** are defined on top of the community functions and add functionality to existing methods. For example for every method the units of the arguments and return values can be defined in a filter. Wrappers add functionality to individual methods.

Attributes

The template divides the interface object of a code into a number of attributes. Each attribute refers to an object implementing a specific sub-interface of the code. For example a code can have a *parameter* attribute, this attribute implements the *ParameterSet* sub-interface. The *ParameterSet* sub-interface defines how to interact with the parameters of a code (in this case each parameter can be set or queried from the set by name using normal python attribute access).

The *template* for all codes is divided into the following sub-interfaces:

parameters Parameters influence how the code works. Parameters are usually set just after creating a code. Parameters should be read-write or write-only.

properties Properties inform the user about the state of the code. The current model time is a property. Properties are always read-only.

particle sets Particle sets provide a common interface for a set of particles in the code. A code can have multiple particle sets defined under different names (for example gas, stars and dark matter)

grids Grids provide access to multi-dimensional data. A code can have multiple grids defined in a hierarchy (for AMR or SMR codes)

Wrappers

Wrappers decorate a method. Wrappers can do pre- and post-processing of the arguments or decide if a method can savely be called.

units and error code Defines a unit for each argument of the wrapped method. When called the arguments will be converted to numbers in the correct unit. The return values will be converted to quantities (numbers with a unit).

state The state of a code determines which functions are valid to call and how the code can transfer from one state into another. For example, a code might give incorrect answers if the potential energy is requested before the particles are entered into the code, the state model will rais an error to inform the script writer of this problem.

Implementation

The implementation of the *object-oriented* interface is based on the adaptor pattern. A *Community Code Interface* class is adapted to create a class which provides *"parameters"*, *"particle sets/gridpoints"* , *"methods with units"* , *"properties with units"* , *"state control"* and *"Unit conversions for incompatible unit systems"*. Each functionality has the same interface for all codes in the system.

2.1.3 User Script

The final layer is the **User Script Layer** this layer contains all the scripts written by a researcher for a specific problem or set of problems. These scripts are always written in *python* and can use all the functionality provided by the two lower layers in the AMUSE framework. The scripts don't need to follow a fixed design.

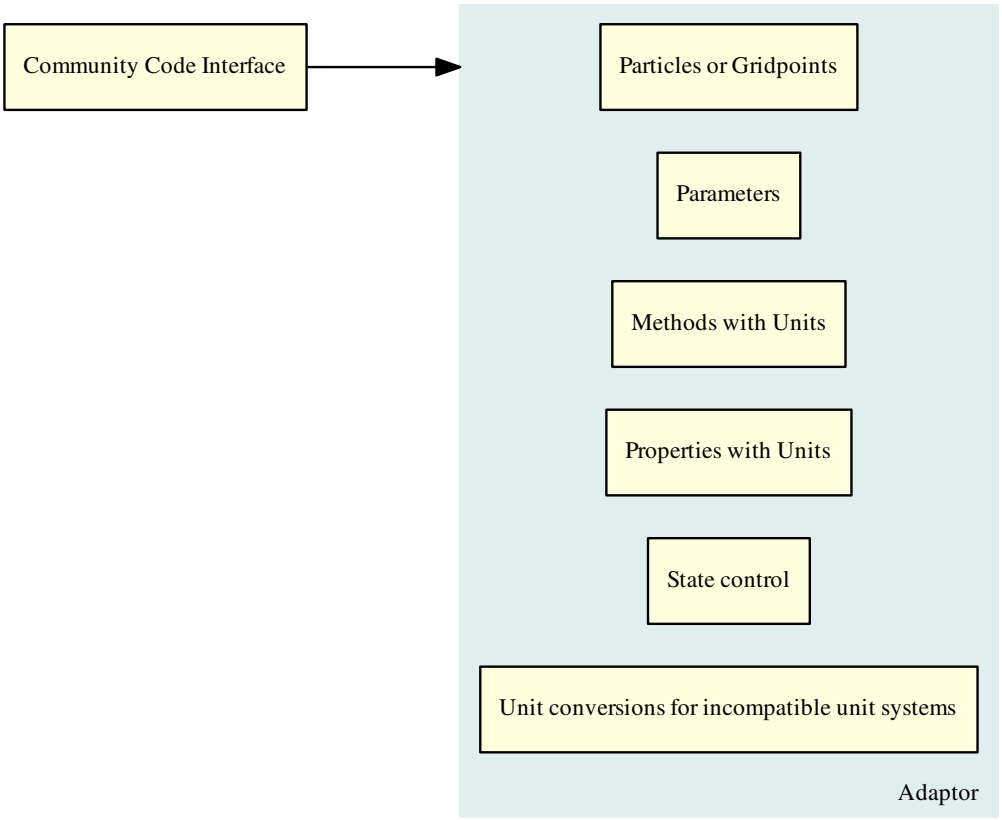
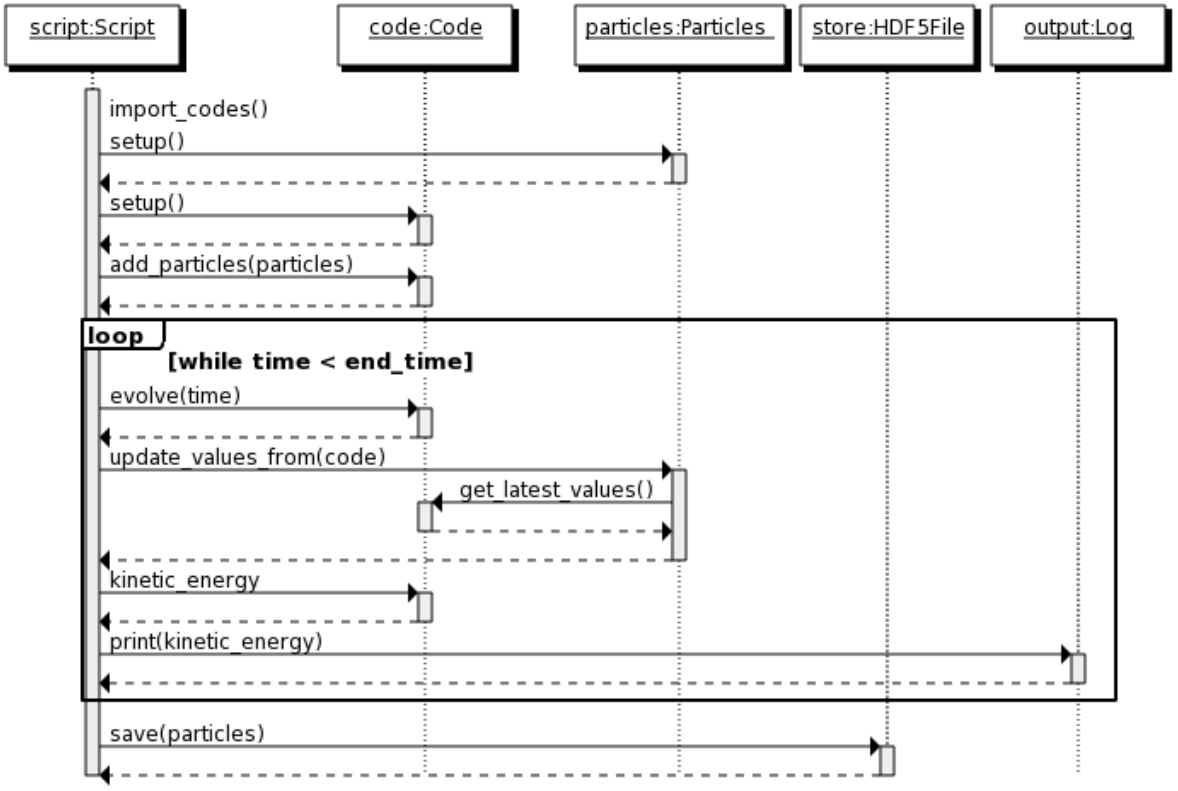


Figure 2.3: A legacy interface is adapted to provide an object oriented interface and more functionality.


script:Script
code:Code
particles:Particles
store:HDF5File
output:Log
import_codes()
setup()
setup()
add_particles(particles)
loop
[while time < end_time]
evolve(time)
update_values_from(code)
get_latest_values()
kinetic_energy
print(kinetic_energy)
save(particles)

CHAPTER

THREE

COUPLING CODES

The design for coupling codes in AMUSE is based on providing the same set of functions for every community code and using these to devise different coupling methods. As the coupling methods are not fixed and can change on a per problem basis the functions to be very generic. The AMUSE library defines three sets of functions to support coupling codes:

particle or gridpoint manipulation Most properties of particles (or gridpoints) can be queried and updated during the run, providing a direct method of manipulating the data of a community code. Further most codes support removing and adding particles during the run.

stopping conditions Stopping conditions are designed to interrupt a code during model evolutions. Stopping conditions are triggered when a code encounters a predefined state (for example a particle escaping out of the bounding box).

services Services are functions added to a code that use the model of that code to provide data for other codes. For example a smoothed particle hydrodynamics code can provide the state of the model at any random point (not just on the particles) which can be used to create a grid from an particle model.

CHAPTER

FOUR

PYTHON PACKAGES

Like all large python projects, the AMUSE source-code is structured in packages, subpackages and modules. Packages and subpackages are directories in the filesystem and modules are python files (A detailed explanation can be found in *Modules <http://docs.python.org/tutorial/modules.html>*). In this chapter the most important packages are named.

The sourcecode of the **AMUSE Code** and **Legacy Codes** layer is combined in one package. The package name is **amuse**.

amuse Root package of the AMUSE code, does not contain test files, the build system or the test system.

The **amuse** and the package is further divided into three subpackages:

amuse.support Contains the code of the **AMUSE Code** layer. The units system, data io and model and all base classes can be found in this package.

amuse.legacy Contains the code the **Legacy Codes** layer. The legacy codes can be found in this package as well as support code for generating the script to legacy code messaging framework.

amuse.ext Contains **extra** and/or **extension** code. For example, making initial data models is not one of the main functionalities of AMUSE, but it is very useful to include this into the codebase.

CHAPTER

FIVE

DATAMODEL

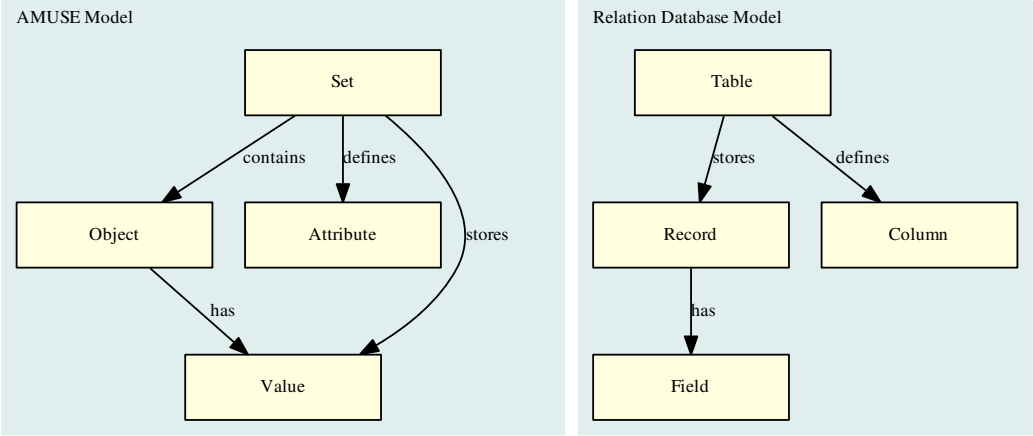
5.1 All data is stored in sets

In the datamodel of AMUSE all data are stored in sets. The sets store objects and the values of attributes belonging to the objects. All objects in a set have the same attributes, but not same values for these attributes.

set	attribute a	attribute b	..	attribute z
object 1	value 1.a	value 1.b	..	value 1.z
object 2	value 2.a	value 2.b	..	value 2.z
..	..	..	..	..
object n	value n.a	value n.b	..	value n.z

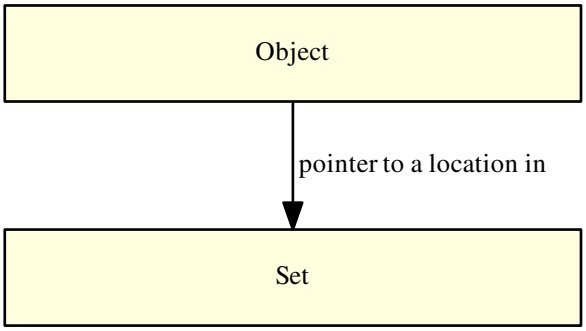
5.2 Like the relation database model

For every object in a set, the set will store the values of the attributes of the object. This model is like a relation database with Tables (sets in AMUSE), Records (an object in the set) , Columns (an attribute of an object) and Fields (the value of an attribute of an object).

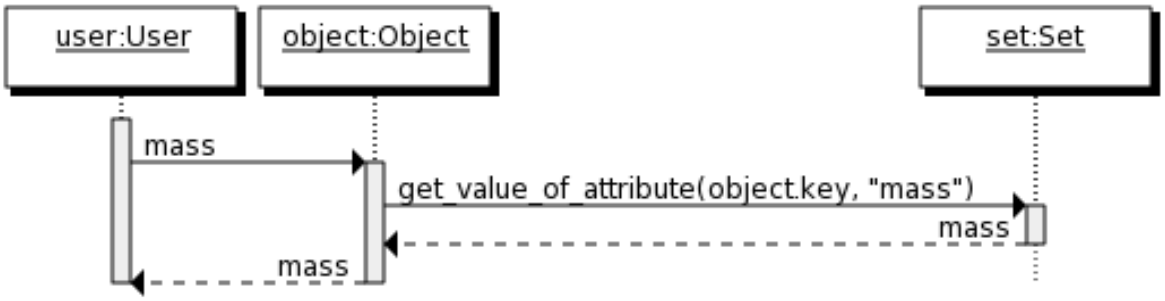


5.3 Objects are views

Objects from a set do not store any values, instead they defer to the set to provide their attribute values. In a sense these objects are pointers to a location in the set. When comparing to the relational database model an object is like a cursor. It can be used to access the values of the attributes belonging to the object stored in the set.



When a user asks an object for its mass the object will query the set for the stored value and return the answer to the user.

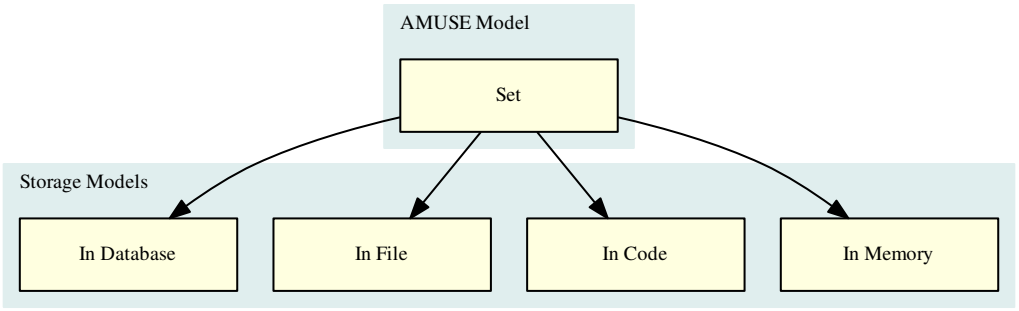


5.4 Objects have a key

The objects in a set can be identified with a unique key. All objects having the same key are seen as the same object by the AMUSE system. The same object can exist in multiple sets. In each set this object can have a different value of an attribute or different attributes. Or, in each set a different version of the object can exist.

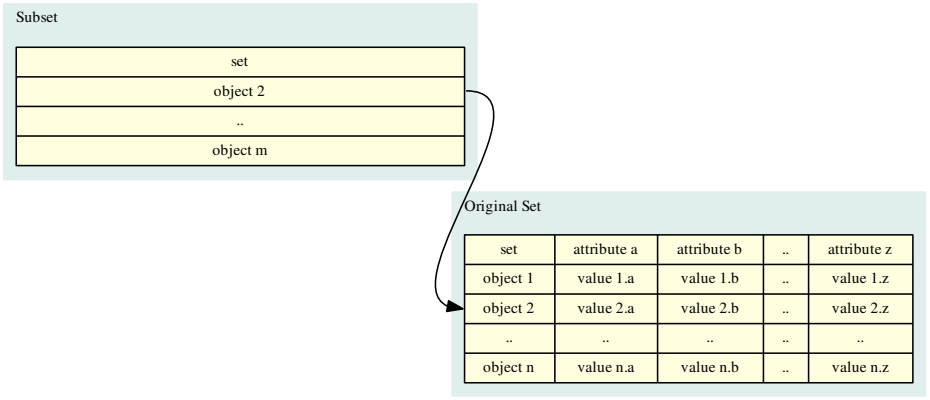
5.5 Sets use Storage Models

The actual storage of attribute values in a set is provided by a storage model. The set provides the interface to the script writer, the storage model manages the data. Each storage model must decide how and where to store the data. All data can be stored in the memory area of the script or in the mememory area of the code or on a file or in a relational database.



5.6 Selections on the set

The datamodel provides subsets to handle a selection of the objects in a set. When comparing to the relational database model an subset is like a view. The subset does not store any data, all the data is stored in the original set. When an attribute is updated in a subset, the attribute is also updated in the original data.



CHAPTER

SIX

MPI INTERFACE

The interface between the AMUSE python core and the legacy-codes is based on the MPI framework. Choosing MPI and not SWIG (or any other direct coupling method) has several advantages:

- MPI is a well-known framework in the astrophysics community. Other coupling methods are less well known (like SWIG)
- Legacy code does not run in the python space (memory usage, names)
- Multiple instances of the same legacy code can easily be supported (not so in SWIG / f2py couplings)
- Multi-process support taken into account at the start of the project.
- Coupling is much looser.

There are also be some disadvantages:

- Need to define a protocol over MPI
- More "hand-work" needed to couple code. Other frameworks, like SWIG and f2py, generate an interface based on the appliction code.
- More overhead for every call, slower calls

These disadvantages are mitigated by creating a library that handles most of the coupling details. This library has a Python, C++ and Fortran version. It implements the protocol and generates hooks to connect with the legacy codes.

The overhead per call may be an important factor in the speed of the framework. This will be tested during development of the first codes. It should be possible to limit the overhead by sending a lot of data per call. For example, setting the properties of a lot of stars in one call. Calling a lot of methods with limited data will be compared to sending one method with a lot of data.