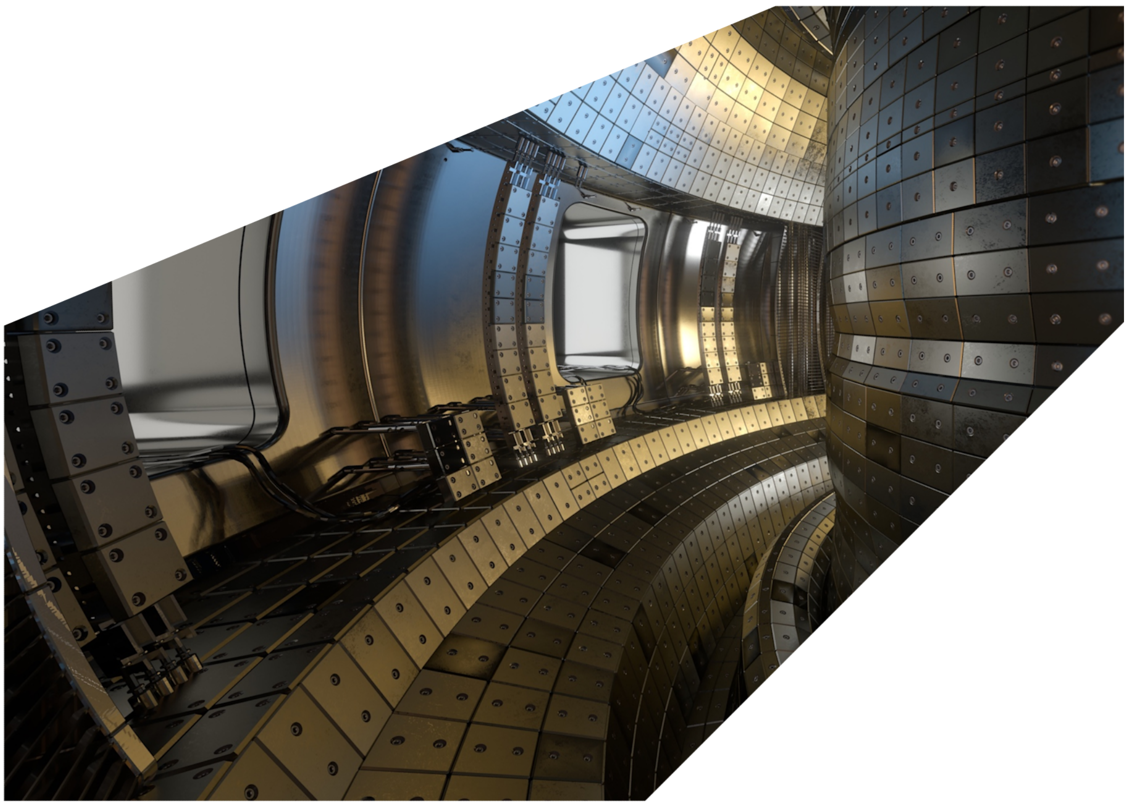
UK Atomic Energy Authority

ExCALIBUR

Module Guide

D3.3

Abstract
The report describes work for ExCALIBUR project NEPTUNE at the point of Deliverable 3.3. This report provides a discussion for the NEPTUNE project of both very high-level and very low-level options for the basic module structure, together with input/output and fine-scale data structures. It complements the description provided by the previous milestone reports [1, 2, 3]. This report completes D3.3 to produce a background against which final decisions may be taken regarding decomposition of NEPTUNE software into modules, and the grouping of modules into packages. It recommends use of the Unified Modeling Language (UML 2) to describe software structure.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0032
	Issue:	1.0
	Date:	March 15, 2021

Project Name: ExCALIBUR Fusion Modelling System

	Name and Department	Signature	Date
Prepared By:	Wayne Arter Ed Threlfall Joseph Parker Will Saunders BD	N/A N/A N/A N/A	March 15, 2021 March 15, 2021 March 15, 2021 March 15, 2021
Reviewed By:	Rob Akers Advanced Computing Dept. Manager		March 15, 2021
Approved By:	Martin O'Brien MSSC		March 15, 2021

1 Introduction

This report comes as D3.3, a deliverable addressing the problem as to how the software for project NEPTUNE should be structured in order to facilitate the development. The original planning of the project envisaged that by this stage there would have been time for more extensive interactions with the wider community, who will after all be largely responsible for the provision of code. However, such discussions have not been possible, owing to the delays in the awarding grants and building the community due to the COVID-19 pandemic. Thus the material presented here although based on the extensive work described in the previous milestone M3.3.x reports [1, 2, 3], has to be regarded in part as a basis for further negotiation with the community. Since it remains unclear how the software may be best written so as to execute well on a range of different machine architectures – and indeed which architectures will successfully operate at the Exascale – a choice of a parallelism abstraction layer (between eg. RAJA, Kokkos and SYCL) continues to be deferred [3, § 3.2]. In the former case, details are expected to change in the light of experience developing the proxyapps, when hopefully the latter situation regarding the choice of parallel library will have become clearer. For the present, only technical considerations to guide the selection are presented.

The survey of software patterns in the M3.3.x reports supports the contention that, in the context of scientific software, patterns should be viewed largely as a way of communicating the functionality of a module or subroutine. Hence, as an important ultimate aim of the software is incorporation into a engineering design workflow for a fusion reactor or its control systems, this report veers more to the model-based systems engineering (MBSE) viewpoint. A widely recommended language for engineering design is SysML^{TM}. [4]. Conveniently SysML is underlain by the Unified Modelling Language (UML [5]), already used in refs [2, 3] to describe software patterns, and since NEPTUNE is a software project, more than adequate for present purposes.

Employing preferentially the UML nomenclature from UML 2,, this report aims to provide a complete background against which to design the NEPTUNE software, complementing the M3.3.x reports. Section 2.1 discusses high-level constraints on the structure of software. where the concept of division into packages and modules (which may represent libraries) is promoted. Section 2.2 explores the implications of these constraints for NEPTUNE. At the opposite extreme to Section 2.1, Section 2.3 describes the desirable contents for a single module, and the last Section 2.5 the smallest scale data structures relevant to selection of an abstraction layer for machine parallelism. Section 2.4 discusses the question of what best to output when developing software for the Exascale.

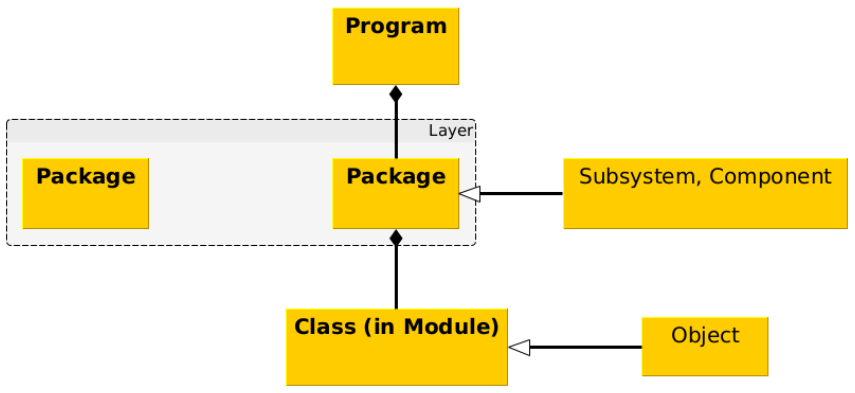


Figure 1: Sketch of relationship between units of large code.

2 Discussion

2.1 General Considerations

It is important that individual units of code are manageable and the overall structure is comprehensible, so that developers and users can navigate the codebase and determine where new work is to be located. This simple consideration implies that NEPTUNE software should be divided into units that it will be convenient to refer to as modules, sections of code containing everything necessary to perform one (and only one) aspect of the desired functionality.

The suggested content of a module describing a single class in the UML sense (see Section 2.3, Table 1) implies a minimum of 13 methods described by separate Fortran subroutines, with examples extending up to 50, although the latter is there regarded as an excessively large number of methods.. Many small software libraries also fall into this size range. Supposing that each subroutine is of a length to fit within one computer window, ie. up to about 60 lines, the desirable maximum length of a well-designed module file is $30 \times 60 = 1\,800$ lines which is a manageable size of file given editing software that remembers on restart, the last line accessed (so that it possible to return immediately to a particular subroutine). The magnitude of the D3.3 exercise follows from the fact that comparable software packages probably of somewhat lesser complexity than NEPTUNE written in high level languages such as C++ range in size up to $1\,000\,000$ lines. Clearly 400 separate modules is too unwieldy, and there is a need to organise further into packages which might contain in turn 10-50 modules. As a way of providing further indication to developers, it is helpful to talk of packages as being arranged into layers, as discussed in ref [3, §§2.4,3.2], see Figure 1. Then, as prefigured in ref [3, Annex A], it is possible to encapsulate the necessary complexity in one, albeit large diagram.

2.2 Considerations for Scientific Software

2.2.1 Structural Considerations

In both refs [1, 2], a figure from Dubey et al [6] is reproduced that illustrates how scientific software may be developed by beginning with an "infrastructure" capability into which initially exploratory scientific software is integrated as its worth is established. Unfortunately for NEPTUNE, it is not clear initially what aspects of the infrastructure will be durable and stable, although once the software is more mature, Dubey et al's methodology appears attractive.

The literature referenced in ref [3] and indeed the local practice of the SMARDDA development indicates that scientific software should be produced by aggregation, but are less helpful as to what is to be aggregated, i.e. the modular decomposition as to what should constitute a single module etc. Surprisingly little detailed discussion was found in the literature after the early book by Booch [7], as to how to create classes in the scientific and engineering context, with the exception of Douglass [8, § 5]. The ideal would be a way to create classes that enabled rapid development of code that executed efficiently but was easily re-used. Douglass [8, § 5] does not specifically address these last points, but they could guide choice of objects in his approach which is reproduced as Appendix Section A.

The NEPTUNE development will proceed as a series of proxyapps. Thus there is a chance to refine and redefine an initial modular decomposition with each successive proxyapp, recording the results as a UML structure diagram. When generating the corresponding sequence diagram (i.e. procedural description), the proxyapp-based units should be preserved to facilitate the use of the simpler ones as surrogates for the later more complex proxyapps.

To help understand the aggregation of the software, it should be layered in the obvious manner, with the higher layers corresponding to greater numbers of aggregated objects. It is also expected to be useful to classify the modules. The modules should be arranged into a relatively small number of packages according to, for example, whether they treat geometry generation, matrix coefficient calculation, the main matrix solution, or visualisation.

2.2.2 Interactions between Modules

Concerning logical interconnections between modules, the use of a directed, acyclic graph (DAG) structure might be thought mandatory, particularly to process the input data in order to specify coherently the construction of the elements of the solution matrix. However, as prefigured in ref [3] for the gyrokinetics code GS2, the tightly coupled nature of the central edge physics problem means that input is more about gathering *all* the data, for only at that point can fields be computed and only after that may matrix coefficients be computed.

2.3 Design of a Module

2.3.1 Notation

It should be noted that when discussing software in text documents, the following conventions in LaTeX are used

- SMALL CAPITALS denote a package name
- *Italics* denote a program name
- `Fixed` width font denotes any code name or fragment which is not otherwise obviously source code
- **Bold** denotes a shell script name

However special fonts are not employed if the class is identified by a suitable suffix, thus "_m" for a module containing executable code defining methods, "_h" or ".h" for class attributes (equivalently derived type or namespace code), ".cpp" for name of file containing C++ source, ".exe" for an executable, etc. UML nomenclature is preferred herein, for which Table 1 provides a limited translation into C++ and Object Fortran.

Table 1: UML interpreted as Object Fortran and C++

UML	Fortran 2003	C++
Class	Derived type	Class
Part	-	Component
Attribute	Component	Data member
Method	Type-bound procedure	Virtual Member function
Feature	Component and Type-bound procedure	Data and Virtual Member function
Structured class	Extended type	Subclass
Specialisation	Class	Dynamic Polymorphism
Generalisation	Generic interface	Function overloading

Further insight into UML terminology may be gained from the description of the patterns in refs [2, 3].

2.3.2 Module design

The layering and the aggregation used in the SMARDDA modules was described in the Report [3, § 2.4]. The focus herein is the structure within a module. Specifically, the module describes a single class or object (strictly speaking, in UML objects are instances of a class), which is fundamental in that it is defined without use of aggregation. As mentioned, the modules in SMARDDA are implemented in Object Fortran, however it is expected that the same functionality would be required in any language. In fact, the software style - consistent with Clerman and Spector [9, § 11] - that is promoted by Arter et al [10], recognises two sizes of fundamental class and it is easier to start by considering the smaller, denoted smallobj_m.

Module smallobj_m does not have a separate attributes file, but will typically still use or access three or more yet more fundamental classes, namely

- log_m for logging errors or warnings in code execution, and checkpointing values of selected variables
- const_numphys_h for numerical values of important mathematical and physical constants relevant to plasma physics
- const_kind_m to specify precision of representation of real and integer values, together with formatting to be used for their output (in fact contains no executable code)
- date_time_m to return the date and time in either a verbose or minimal format
- misc_m to form miscellaneous operations found to be common to many modules

In outline, the methods or operations associated with smallobj allow data to be read from a named file, used in the construction of an object, and output to disc file. The file is given a numeric identifier ninso (file unit) when it is opened. Data used to construct the object forms a single Fortran namelist, viz. a list of arbitrary code variables that may (or not) be assigned values in the input file using a attribute-value notation. Namelist variables should have long names that promote a good user interface, and be given default values in case they are not input. The style encourages checking that inputs have acceptable values, for example lie in expected ranges, but there is no equivalent of eg. the Cerberus Python data validation package [11], users must explicitly code checks and calls to log_m if the values are questionable or erroneous. After successful checks, the values in the namelist are copied into the data type sonumerics_t, whence it is supposed that a single subroutine then instantiates the smallobj_t object. Another routine performs output of this object, either to a directly specified file unit, or to the standard output by default, in the simple text format of a variable name followed by its value on the following line. As might be expected, there are also routines to 'delete' the object, ie. to deallocate any constituent arrays, and to close the input file. The precise list of member functions (UML methods) is

- initfile, open input file
- readcon, read data from input file
- generic, generic subroutine to instantiate object
- write, write out object to standard output, or to file opened by another object
- delete, delete object
- close, close input file

Module bigobj_m has a separate file for its attributes (*aka* namespace), which it will normally still use or access like the yet more fundamental objects listed for smallobj_m. However the data types defined in bigobj_h are of the same kind as those in smallobj_m, viz. bonumerics_t to hold input data which is used to define bigobj_t by calling the solve subroutine (rather than the subroutine generic in the case of smallobj_m). Apart from this last exception, the methods in bigobj_m are a superset of those in smallobj_m. The additional methods recognise that instantiation may involve more than one routine and in particular may involve use of a specially defined function fn, demonstrating the Strategy Pattern or possibly Template Pattern. This function may be determined by a formula identified in the input, giving the option for a developer or determined user of adding their own code to define a new function. The range of allowable outputs is much extended. Thus there is a separate routine provided for output to the log file using what will probably be a lengthy list of calls to log_value_m. More likely to be useful is a routine to open an .out file on a file unit given the number noutbo on opening, which becomes the default unit for writing out the object in bigobj_write. There are also provided separate skeleton routines intended to provide output in a format suitable respectively for visualisation by GNUPLOT and PARAVIEW, and of course routines to close files and delete the object. The precise list of member functions for bigobj_m is as follows where those also found in smallobj_m are enclosed in parenthesis:

- (initfile), open input file
- (readcon), read data from input file
- solve, generic subroutine to manage instantiation of object
- userdefined, user-defined method or function
- fn, general external function call
- dia, write object diagnostics to log file
- initwrite, open new file, making up name which defaults to having .out suffix
- (write), write out object to standard output, or to file opened by bigobj
- writeg, write out object suitable for visualisation by GNUPLOT
- writev, write out object suitable for visualisation by PARAVIEW
- (delete), delete object
- (close), close input file
- closewrite, close write file opened by initwrite

Thus the skeleton object is defined by a formula plus data input from disc, and since both are saved internally as features, the instantiation may be deferred as necessary, so-called 'lazy initialisation'. It will be noticed that other variables in bigobj_m such as unit number are hidden, ie. cannot be accessed by other modules. In fact a common addition to the default modules is a method function getunit that returns ninbo, illustrating the approved way of accessing such data.

The source as mentioned in ref [3] may be obtained by download of program *smardda-qprog* [12]. The reason for the emphasis on input and output from and to disc (I/O) then becomes apparent from the main program qprog.f90, viz. facilitating the construction of a test harness for an objects in the style of bigobj_m, and indeed the aggregations of such objects. The use of “attribute-value” in I/O gives flexibility to the developer, since new variables may be added without the need to modify existing input files or output processing. Output processing is further discussed in Section 2.4.

In the course of a software development, the source code relating to an object naturally grows with the addition of further subroutines in the module file. Since NEPTUNE is anyway going to have to describe the equilibrium magnetic field $\mathbf{B}$ it is worth examining what such growth led to the beq_m SMARDDA module containing, besides the standard routines listed above. In fact beq_m grew to 50 routines (the number probably indicating a need for refactoring) so they will not be listed in full. 13 of the new routines are concerned with either reading or writing from disc, reflecting the ability to handle a range of formats of the so-called ‘eqdsk’ type, plus an ability to define the magnetic field analytically. The eqdsk and analytic description assume axisymmetry with the field defined by the magnetic flux function $\psi(R, Z)$ where (R, Z) are cylindrical polar coordinates centred on device major axis. Much of the module complexity is accounted for by the needs either to verify that the field has been correctly read in as the ‘eqdsk’ format is not standardised, or to return $\mathbf{B}$ in either flux, polar or Cartesian coordinates, in formats avoiding storage on a whole 360^o mesh. Onto this structure was subsequently bolted capabilities to superimpose a non-axisymmetric vacuum field (which is not strictly an equilibrium field) and to track through the ψ-landscape for various purposes.

Representative routines in beq_m are

- fixorigin, which fixes a problem encountered when the data in the eqdsk extends to $R = 0$ because eg. the $Z-$component of $\mathbf{B}$ is defined as $(1/R)\partial\psi/\partial R$.
- sense, which returns the sense of helicity of the field
- psilt, to calculate the actual range of values of ψ by resolving possibly conflicting inputs
- ctrackc, return the track in (R, Z) running through the plasma centre of the extremum of ψ
- init, which initialises the field, the equivalent of *solve* and/or *generic*

Arguably ψ should be disaggregated as a separate class for the purposes of tracking extrema etc. In one sense, it is already separate because it constitutes a 2-D spline class, being of type spl2d_t defined in modules spl2_m. However the latter is conceptually a mathematical construct, whereas the analyses in beq_m are physically conceived, which is why they were placed there. This illustrates one of the difficult dilemmas that may arise from the object-oriented approach in scientific applications. Another feature of scientific work is the use of simple analytic formulae either for testing purposes or to include an effect such as field ripple to get a qualitative feel for its influence, rather than always using detailed machine data to get a full quantitative evaluation.

2.4 Processing Module Output

The number of I/O subroutines can be criticised as leading to excessive length of a module. Some might argue that for many debugging purposes an interactive debugger is adequate, and for most others that one output file is sufficient provided it is in a attribute-value format such as JavaScript Object Notation (JSON) or in a self-documenting format such as the Network Common Data Form (netCDF), to be interpreted by any suitable visualisation software.

The problem with scientific software, worse at Exascale, is the volume of data to be handled so that the ability to visualise large arrays as well as large numbers of small arrays is essential (no debugger as far as is known has such a visualisation feature). Moreover, the actionable aspect of NEPTUNE further means that any postprocessing of a generic file must also be carefully performed. Thus while a generic output file could be processed by say Linux awk script into a form suitable for GNUPLOT processing, this gives rise to a need for providing documentation and provenance for the script, which is at minimum a nuisance. Worse is the risk that the amount of data to be processed may be so large as to lead to significant delay and maybe even system or other issues not handled well if at all by the script, all of which may be extremely time-consuming to resolve. Other conversion software may not be available or properly implemented on the target machine. Thus even for debugging purposes, it seems desirable that as much as possible of the processing is done within the higher level language, and for production runs, output in a format directly readable by say PARAVIEW should also speed the post-processing. Fortunately since netCDF may be read directly by PARAVIEW, this is often the best option for writev.

2.5 Parallelism Abstraction

To exploit parallelism most effectively on any given architecture, data must be arranged in arrays to which the same operations can be applied to many (N_{adj}) adjacent elements. The arrangement of data describing, say, the magnetic field or a particle distribution function can nonetheless make a big difference to ultimate speed of execution which can depend sensitively on N_{adj}. Thus a good API could be defined at the array level, taking away from the developer the decision as to whether the data is arranged as $n_x \times n_y \times n_z$ or $n_z \times n_x \times n_y$, ie. as to which array index runs the fastest. Further, extremely large first indices n_x might for example be factored so that the first index is of order 64 to exploit caching, whereas the final index might be used to map array contents to different nodes of the machine.

In a physics modelling code, it seems reasonable that physics should have a say as to how the data is arranged, with the special implication that all information relating to a particular position in space should be as close together as possible. However, particularly for edge physics, this may lead to conflict with an array level API. There are two main problems, namely that at a given spatial point (1) some species may be represented as particles and others as finite elements and some as both, and (2) not all species need be present at a given point. The issue at (1) arises when the species collisionality varies so that a fluid and a high-dimensional representation that accounts more accurately for non-Maxwellian effects are needed in different spatial regions, with the different representations allowed to overlap. Situation (2) may occur with a neutral species that becomes fully ionised with distance into the plasma, or when say singly-charged ions of a certain species are only present in the divertor. The problem is intensified when p-adaptive finite elements are used such that adjacent elements may have different orders of polynomial discretisation. It may also be desirable when working with ensembles to have samples from different solutions but for the same spatial region to be physically close together in storage.

The plasma physical constraint may be met by domain decomposition in position space, so that within each subdomain, fluid species can be represented by one set of arrays, one per species, and particles or other high-dimensional representations as other set(s) of arrays. The optimality of this arrangement, and certainly the size of subdomain, depends on machine architecture. For example, on a node with both conventional CPU cores and a GPU, it might be good to store finite elements adjacent to the CPU and use the GPU for particles. Another option might be to take the localisation concept to its extreme, and arrange together quantities that are close in the 6-D phase velocity and position space, perhaps using an hierarchy of elements in velocity space. Fluid species might be represented by pointers in these elements, without too much wastage of store, even if there is only one species that requires a high-dimensional representation.

Since the main work of a NEPTUNE solver is expected to be the numerical inversion of a large matrix to obtain field values at a new time or iteration, there is even a question mark over how much weight should be attached to the localisation constraint. At the Exascale, the matrix and especially its preconditioner must be virtual in the sense that it will be too large to store all the coefficients simultaneously, given the size of field discretisation. Hence the ease of computation of the coefficients of the matrix may be more important for performance.

Acknowledgement

The support of the UK Meteorological Office and Strategic Priorities Fund is acknowledged.

References

[1] L. Anton, W. Arter, and D. Samaddar. NEPTUNE: Background information and user requirements for design patterns. Technical Report CD/EXCALIBUR-FMS/0015-1.00-M3.3.1, UKAEA, 2020.

[2] W. Arter, E. Threlfall, J. Parker, and S. Pamela. Report on design patterns specifications and prototypes. Technical Report CD/EXCALIBUR-FMS/0023-M3.3.2, UKAEA, 2020.

[3] E. Threlfall, J. Parker, and W. Arter. Design patterns evaluation report. Technical Report CD/EXCALIBUR-FMS/0026-M3.3.3, UKAEA, 2020.

[4] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Elsevier, Amsterdam, 2011.

[5] OMG: Unified Modelling Language. `https://www.omg.org/spec/UML/`, 2021. Accessed: March 2021.

[6] A. Dubey and L.C. McInnes. Idea paper: The lifecycle of software for scientific simulations. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2016.

[7] G. Booch. *Object Oriented Design with applications*. Benjamin/Cummings, Redwood, 1991.

[8] B.P. Douglass. *Real Time UML Workshop for Embedded Systems*. Elsevier, Amsterdam, 2006.

[9] N.S. Clerman and W. Spector. *Modern Fortran. Style and Usage*. Cambridge University Press, 2012.

[10] W. Arter, N. Brealey, J.W. Eastwood, and J.G. Morgan. Fortran 95 Programming Style. Technical Report CCFE-R(15)34, CCFE, 2015. `http://dx.doi.org/10.13140/RG.2.2.27018.41922`,`https://scientific-publications.ukaea.uk/wp-content/uploads/CCFE-R-1534.pdf`.

[11] Cerberus Python data validation. `https://github.com/pyeve/cerberus`, 2021. Accessed: March 2021.

[12] QPROG, simple demonstration of software design by aggregation. `https://github.com/wayne-arter/smardda-qprog`, 2015. Accessed: December 2020.

A Object Identification

Douglass [8, § 5] has a description of object analysis which is well-suited project NEPTUNE. His approach is to take the use cases, which for this purpose should include the proxyapps separately, and treat them carefully one after the other using the strategies indicated in Table 2. Each proxyapp should be carefully analysed and classes produced from the list of objects before proceeding to the next.

Table 2: Key Strategies for Object Identification. After Table 5.1 from ref [8], slightly amended. All the strategies except the last, are concerned with identifying the objects listed.

Strategy	Description
Nouns	Used to gain a first-cut object list, the analyst underlines each noun or noun phrase in the problem statement and evaluates it as a potential object, class, or attribute.
Causal agents	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Messages and information flow	Messages must have an object that sends them and an object that receives them as well as, possibly other objects that process the information contained in the messages. There are many ways to identify the objects within a collaboration.
Real-world items	Real-world items are entities that exist in the real world, but are not necessarily electronic devices. Examples include objects such as gases, forces, blanket modules, etc.
Physical devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. The resulting objects are almost always the interfaces to the devices. Note: this is a special kind of “Identify real-world items”.
Key concepts	Key concepts may be modeled as objects. Physical theories exist only conceptually, but are critical scientific objects. Frequency bins for an on-line autocorrelator may also be objects. Contrast with the “identify real-world items” strategy.
Transactions	Transactions are finite instances of interactions between objects that persist for some significant period of time. An example is queued data.
Persistent information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Visual elements	User-interface elements that display data are objects within the user-interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Control elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects and relations.