



UK Atomic
Energy
Authority



ExCALIBUR


Design Patterns Evaluation report - M3.3.3

Abstract

This report describes work for ExCALIBUR project NEPTUNE at Milestone 3.3.3. The report identifies software design patterns for use by specific NEPTUNE proxyapps, with reference to existing C++ and Fortran simulation codes. The C++ language is represented here by the Nektar++ and BOUT++ frameworks; the Fortran language by the GS2 gyrokinetics code and the SMARDDA modules framework. A further discussion of overarching design issues for integration of the proxyapps into a multiscale, multiphysics framework follows. One annex gives an indication of the design process in the context of game engine software, expected to possess comparable or greater complexity than NEPTUNE. The second annex gives an overview of an existing set of standards for exascale proxyapps, developed by the US Exascale Computing Project.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:		
	UKAEA Reference:	CD/EXCALIBUR-FMS/0026	
	Issue:	1.00	
	Date:	11 December 2020	
Project Name: ExCALIBUR Fusion Modelling System.			
	Name and Department	Signature	Date
Prepared By:	Ed Threlfall Joseph Parker Wayne Arter BD	N/A N/A N/A	11/12/2020
Reviewed By:	Rob Akers	N/A	11/12/2020
Modified By:	N/A		
Approved By:	Rob Akers Advanced Computing Dept. manager BD		18/12/2020

1 Introduction

This report has the goal of consolidating earlier work regarding software design patterns [1] by attempting an initial identification of specific patterns to be applied in the forthcoming NEPTUNE proxyapps, and also summarizing some other general design concerns.

Section 2 identifies patterns relevant to specific core NEPTUNE proxyapps. It is noted that two of these proxyapps are to be based on pre-existing object-oriented (C++) frameworks and the task is made easier by the explicit identification of some of the relevant patterns in those frameworks' accompanying documentation. Those are the proxyapps that are to treat continuous fields; an additional part describes one of the proxyapps designed to treat particles, which at the time of writing is intended to be based on a procedural code written in Fortran. Further insight into the use of design patterns in Fortran, and more generally the construction of complex modular frameworks, can be gleaned from the study of existing codes; to this purpose, a discussion of the SMARDDA simulation framework is included. Following sketches of the proxyapps, a shortlist of relevant design patterns is given.

Section 3 presents some overarching concerns relevant to the initial proxyapp ecosystem as a whole, viewed here as a fledgling multiphysics ecosystem and keeping in sight the likely structure of exascale target hardware. The first annex A demonstrates one tangentially relevant approach to handling a complex software ecosystem. As in earlier work, a general dearth of published material treating patterns in scientific programming is noted, indicating that room is left for more definite strategies for the overall NEPTUNE framework, once the initial implementations of the proxyapps become available.

A second annex B contains a precis of a set of standards for scientific proxyapps and a framework for benchmarking the same proposed by the US Department of Energy's Exascale Computing Project.

2 Design patterns for individual NEPTUNE proxyapps

In this section, descriptions of the proxyapps that will be created under three of the project NEPTUNE calls are given: in subsection 2.1 the proxyapp for Call 1, “Examining the performance of Nektar++ for fusion applications”, in subsection 2.2 the proxyapps for Call 6, “Fluid referent models” and in subsection 2.3 the proxyapps for Call 8, “Development of a gyro-averaged referent model”. A further subsection 2.4 contains a discussion of the existing SMARDDA modules framework. Finally, in subsection 2.5, a discussion of the different design patterns employed in the proxyapps is provided.

2.1 Software design patterns for Call 1 Proxyapp

The proxyapp specified in Section 2.1.5 of Contract Ref. T/NA078/20, *Proxyapp instantiating a 2D model of anisotropic heat transport*, is designed to investigate and quantify the performance of spectral element methods [2] in modelling anisotropic heat transport in close proximity to a complex first wall geometry. The initial remit is for a two-dimensional solver only. It is worth noting that a rather generic requirement for next-generation algorithms is spectral accuracy, not least because it is suited to the current HPC landscape - hence, some of the design patterns used in this proxyapp are expected to bleed into other NEPTUNE work.

The proxyapp will be built within the pre-existing *Nektar++* framework for spectral / hp element PDE solvers [3] and will extend current capabilities to include equations governing the scrape-off layer plasma. The higher-order methods used by *Nektar++* generically involve a large amount of arithmetic for a given quantity of data, a computational pattern that is well-suited to today’s HPC landscape. The scope of the problem encompasses also techniques for generating two- and three-dimensional meshes capable of conforming to a reactor wall and also to the geometry of local magnetic field lines representing, for example, the tokamak X-point and this capability will be provided by *NekMesh*, a meshing framework capable of importing computational meshes from popular CAD formats as well as generating its own meshes and which is integrated with *Nektar++* [3].

The proxyapp has yet to be implemented but in view of the facts that it will leverage existing *Nektar++* code and that codes of this type are likely to have certain common and well-defined characteristics, a description of the main software design patterns used in the *Nektar++* framework is presented (for more details, see the discussion in Section 3.5 of [4]).

2.2 Software design patterns for Call 6 Proxyapps

The Contract Ref. T/NA083/20 specifies the creation of five proxyapps to mirror the development of the referent fluid model. This model will capture the physics in the tokamak edge, including the separatrix but away from the core and from the metal surfaces of the device. It is to include the multi-physics phenomena relevant to the tokamak exhaust (bulk plasma species, impurities and neutrals). The code itself must be performant and scalable to Exascale, easy to deploy upon different architectures, and actionable (i.e. include treatment of uncertainty quantification, UQ). More-

over it must couple efficiently to the neutral gas and impurity model, and the 5D gyro-averaged model developed in other work packages.

The five proxyapps are largely independent, focussing on separate aspects, namely:

1. 2D elliptic solver in complex geometry
2. 1D simplified fluid model with UQ and realistic boundary conditions
3. 1D model incorporating velocity space effects
4. 1D multispecies plasma model
5. 2D model incorporating velocity space effects

The two proxyapps focussing on velocity space effects (proxyapps 3 and 5) will draw on work on kinetic solvers, and will be written in a high-level language such as Python or Julia. For each of the remaining proxyapps, two versions will be developed for comparison, one in BOUT++ [5] and one in Nektar++. As mentioned above, Nektar++ is a framework for solving PDEs with spectral/hp, which will be extended as part of Call 1 to include terms relevant for plasma physics. In contrast, BOUT++ is a framework for solving plasma and fluid-like equations in curvilinear coordinates using finite-difference approaches, but lacks the complex meshing and finite element approaches of Nektar++.

Proxyapps 2 and 4 above investigate the feasibility of including new physics (namely realistic sheath boundary conditions and multiple species). For this, new terms will be added to the existing BOUT++ physics modules, SD1D and Hermes-3. Owing to BOUT++'s domain specific language, these additions will be relatively straightforward (from a software development perspective).

The remaining proxyapp (number 1) will be implemented by coupling BOUT++ to elliptic solvers provided by PETSc and Hypre. BOUT++ uses the strategy pattern (see section 2.5.3 below) to implement different elliptic solvers. BOUT++ also makes use of the template method (section 2.5.1) and abstract factories (section 2.5.2).

2.3 Software design patterns for Call 8 Proxyapps

The Contract Ref. T/NA085/20 focusses on the development of a referent gyrokinetics model. This work package will investigate a novel approach to deriving the gyrokinetic equations, where the local three-dimensional fluid quantities (density, momentum and energy) are embedded into the five-dimensional particle distribution function. This leads to a version of gyrokinetics with both a kinetic equation for the modified distribution function and a set of fluid equations for the local moments. This allows one to use a single model to study both the core (where one solves the whole equation system) and for the scrape-off layer (where one solves only the fluid equations). Alternatively, the model allows a very natural coupling of the gyrokinetic equations to any set of fluid equations for the scrape-off layer.

Because of the novelty of this approach, the work package will first derive drift kinetic equations (the simplified, long-perpendicular wavelength limit of gyrokinetics). It will also develop proxyapps

to compare the numerical performance of the new model to a more standard drift kinetic model for a variety of problems. It will determine potential bottlenecks in the numerical treatment of the drift kinetic models, and, in addition, develop appropriate numerical algorithms for treating the wall boundary.

The proxyapp will be developed from scratch using Julia, which combines rapid prototyping in a high-level language and access to computational libraries, with performance that approaches that of statically-typed languages. While there is no design pattern yet specified for the proxyapp, the developers are also early contributors to the gyrokinetics code GS2 [6]. It therefore seems likely that the proxyapp will share design features with GS2.

GS2 is written in Fortran 90, using a modular design pattern (see section 2.5.6). It utilizes lazy initialization (see section 2.5.7) as a technique to passively manage module dependencies. The modular design is a result of the age of the code. Object-oriented techniques have been introduced, but in an incremental fashion that allows modular design to coexist with encapsulated objects. To achieve this, dependency injection (section 2.5.8) was adopted, with module-level variables being phased out in favour of large `state` objects. Having a small number of objects with many members has proven convenient for developers, and it seems likely this proxyapp will retain this approach from GS2, with perhaps a few `state` objects representing the solution, the diagnostic outputs and the simulation parameters.

2.4 SMARDDA modules framework revisited

The SMARDDA modules framework was discussed briefly in the M3.1.2 report [7], where particularly its layered structure was noted as relevant to NEPTUNE. Exploration of the literature since, as described in the earlier Section 2.1–Section 2.3, has indicated that it is both common and commended by many workers to construct complex software through aggregation of smaller objects that may themselves be the basis for physically separate codes. Since the SMARDDA software is also built using aggregation in this way, a more detailed description of this is provided here.

As previously mentioned [7], the SMARDDA software is coded in a Fortran language style documented in a Culham report [8] (which is in fact very similar to other styles recommended in the meteorological community) exemplified by templates which have been posted on github, as program `smardda-qprog` [9]. The new material presented here treats *qprog* as a standalone, independent of the SMARDDA modules, to emphasise its structure. Note that the name ‘qprog’ has been deliberately chosen so as not conflict with other software, the ‘q’ indicating that it poses all the questions to the developer, since it is purely a skeleton framework accepting nominal inputs to produce nominal outputs.

Layering is represented by the presence of a subdirectory LIB, containing a subroutine library compiled from Fortran conforming to the older FORTRAN-77 standard. In practice if not in implementation, the Fortran 95 utility routines which duplicate many of the functions of the old OLYMPUS library [7] also share this layer. The most widely used of these utilities is `log_m` for logging progress made by the code including errors, and timing information for which modules `date_time_m` and `clock_m` are also needed.

There is a script `qprog.bash` in the repo [9] which will set up the files, see Table 1, making up

a conforming program, and indeed try to compile, link and run it. (It does not need SMARDDA installation if `-l` for locally complete is set as the first option.) It is perhaps easiest to start understanding the framework from the documentation of the script. The program name, its abbreviation (typically one or two letters) and description are defined using the keys QPROG, Q and STR. The name of a skeleton object module to be tested, its abbreviation and description are defined using the keys BIGOBJ, BO and BSTR:

```
qprog.bash [-l] \  
QPROG=xpc Q=xp STR="transport_coefficients" \  
BIGOBJ=clcoef BO=cc BSTR="classical_coefficients"
```

The example is that of a code `xpc` to calculate transport coefficients, where the program name also corresponds to an object, which in this case is the set of transport models. In the example, for simplicity's sake, only one physical model, giving the so-called classical or Braginskii coefficients is considered, as object `clcoef`. The relation between the objects is shown in Figure 2. In the example, all objects are 'big', ie. the object description is separated in a file with name ending "`_h`" from the subroutines which operate on the object in a file with name ending "`_m`", thus `clcoef_h` and `clcoef_m`. For a small object, object and associated subroutines are both combined in one module ("`_m`") file. Whereas the full name is used to describe the namelist QPROGparameters associated with the object (namelist is a Fortran language feature for user-friendly keyed input of values for code variables), the abbreviation is used to produce type names associated with the object, notably type `Qnumerics` (or `BOnumerics`, `xpnumerics` and `ccnumerics` respectively in the example). The latter separates out the data needed to define the object, making it possible for instantiation to be deferred ('lazy initialisation'). There is also produced a control module `xpcontrol_m` for the program object.

As the templates indicate, each object is largely self-contained within its module, with subroutines for opening a file which contains input control data, reading from it and closing it, together with output routines. These latter routines manage files which might contain numeric dumps of the object variables and descriptions of the object in plotting formats defined for *gnuplot* and *ParaView*, to be used in post-processing and to visualise aspects of the object in up to 3-D.

The calling tree shown in Figure 1 indicates how the calls, starting with `xpcontrol_read` (note the use of the abbreviation in the subroutine name), are arranged so that input data defining all the objects can be read from a single file. The 1, 2, 3 layout is used by the main program `xpc` [7] where in sequence order, 1 is initialisation, 2 is compute and 3 is output and closedown, but this layout is not generally suitable for use by the other objects.

It is worth noting that there is a `.log` file with a special status designed to ensure that as far as possible diagnostics are captured even should serious errors occur.

It will be seen that, assuming the coefficients of the various transport models combine to give the total transport by the plasma, the code will in essence implement the puppeteer pattern despite the constraint of a single control file, in that the `clcoef_m` module (and consequently other modules that might be written to define other sets of coefficients) knows nothing about the `xpc` object. Information needed by `clcoef` and sibling modules, such as details of the plasma composition, is to be passed to them in module `xpcontrol_m`. The present implementation of this last in `xpcontrol_m` is inelegant, and only adequate for a simple example. For more realistic applications, understanding and mapping the dependencies between the various inputs will likely require a graph-based

elaboration on the present code design.

Extension to other sibling models such as anomalous transport should be straightforward. All that has to be arranged is for `xpcontrol_read` to call `anomcoeff_readcon`, `xpc_solve` to call `anomcoeff_solve` and the various output routines `xpc_writev`, `xpc_writeg` and `xpc_write` to call their equivalents in the transport coefficient objects.

It is also worth mentioning that use of templates enabled the production of over 1400 lines of potentially tricky code logic from about 50 lines of pseudo-code defining variables to be found in the repo files `qprog.txt` and `clcoef.txt` plus the 2 lines defining `QPROG`, `Q`, etc. above, using a lengthy but relatively straightforward script, and the total line count allowing for the utilities is nearly 4000. In fact the bulk of the remaining code to evaluate classical transport coefficients to be found on the web-site [10] was produced with the help of the `reduce-algebra` package [11] which can output mathematical expression in a Fortran syntax.

Table 1: Files and principal input functions of the QPROG code (`xpc` example). The `.f90` suffix has been omitted from the filenames.

File	Description
<code>QPROG</code>	main program, see its sequence diagram in Figure 1
<code>QPROG.h</code>	parameters collected in type <code>Qnumerics_t</code> describing how to construct (at least one) object <code>BIGOBJ</code> which is to be combined with other object data structures such as <code>BIGOBJ_t</code> into type <code>QPROG_t</code>
<code>QPROG.m</code>	calls <code>QPROG_readcon</code> , which reads namelist variables listed in <code>QPROGpa-</code> <code>rameters</code> and copies them to type <code>Qnumerics_t</code> variables
<code>Qcontrol.h</code>	data structure of generic top level controls for QPROG (abbreviated as <code>Q</code>)
<code>Qcontrol.m</code>	object of generic top level controls for QPROG (abbreviated as <code>Q</code>), calls <code>BIGOBJ_readcon</code> via <code>Qcontrol_read</code>
<code>BIGOBJ.h</code>	data structure of controls for <code>BIGOBJ</code> forming type <code>BOnumerics_t</code> , also combined type <code>BIGOBJ_t</code>
<code>BIGOBJ.m</code>	object which reads in object level controls

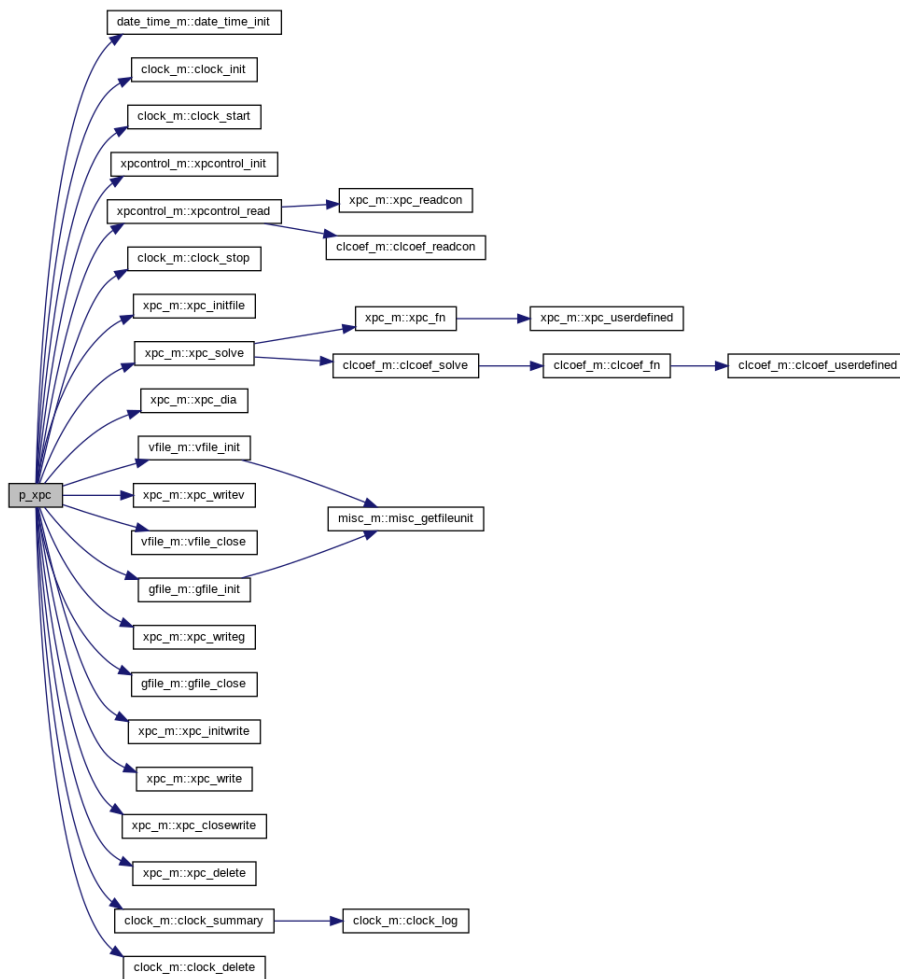


Figure 1: Calling structure for SMARDDA style program, plot produced by doxygen which in this example largely imitates a UML sequence diagram. The important subroutines to note are `xpccontrol_read` which coordinates the input of data defining the two objects `clcoef.t` and `xpc.t`, and `xpc_solve` which calls subroutines operating on both objects to do the work of the code. (Calls to `log_m` subroutines have been suppressed for clarity.)

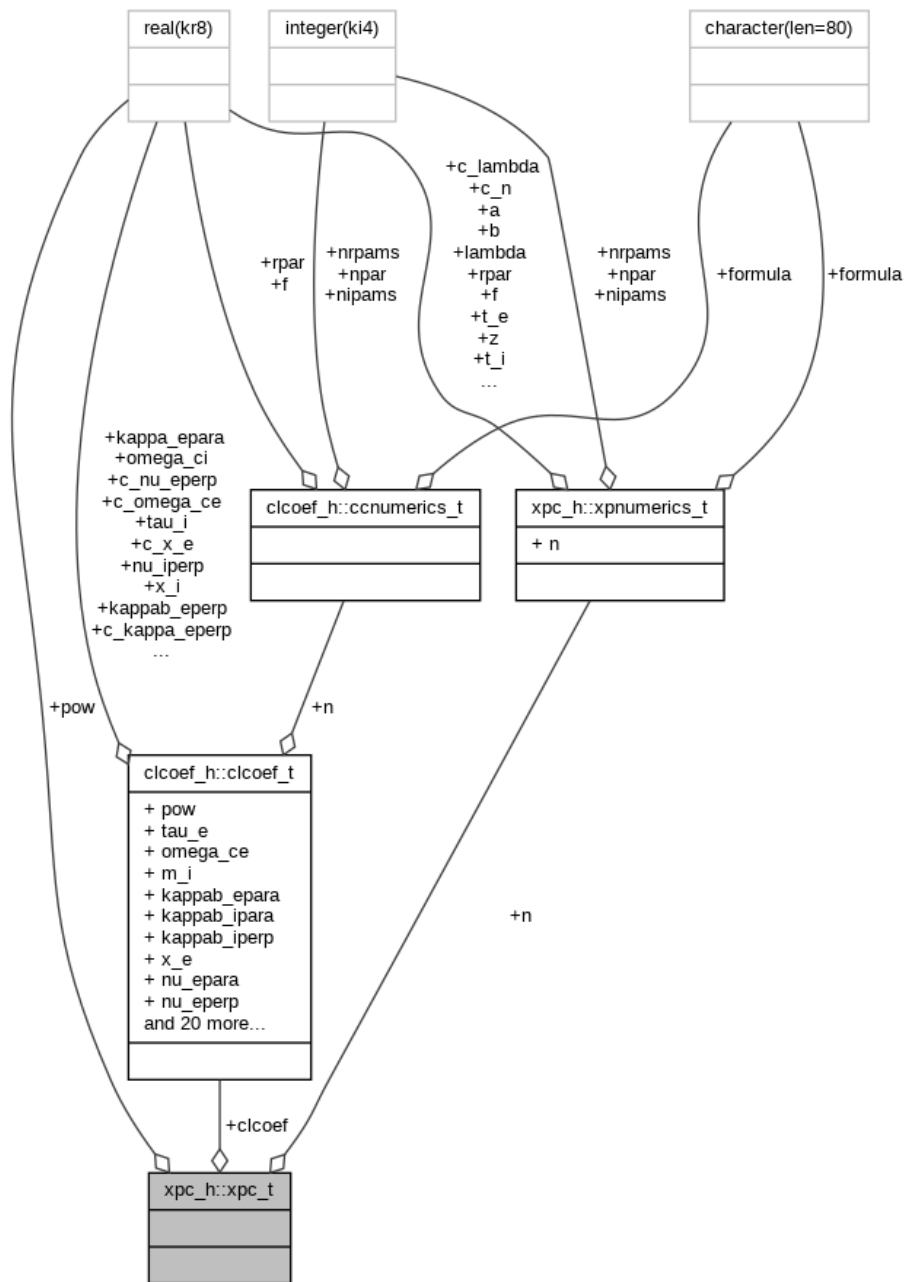


Figure 2: Data structure for SMARTDA style program. The plot produced by doxygen unhelpfully links variables to their simple types (real, integer and character), however it should be clear each object is associated with a numerics type that defines how it is to be generated. The figure also indicates how objects are aggregated, in this instance there is only one object clcoef.t aggregated with the code level controls to form the code object xpc.t.

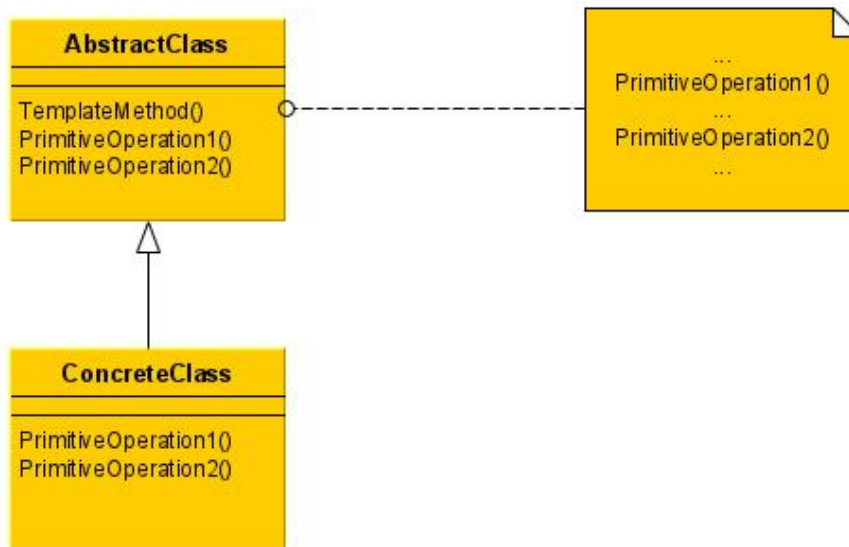


Figure 3: UML diagram for the Template Method pattern.

2.5 Design patterns

2.5.1 Template Method

The Template Method involves the implementation of an algorithm as a carcass with interchangeable components, deferring some of the components to subclasses; thus steps in an algorithm can be redefined while retaining the overall algorithm structure. Code outside the class hierarchy sees a common interface: the public function in the base class, the implementation of which contains the invariant parts of the algorithm, thereby avoiding code duplication. A selection of protected virtual component functions may be called by other classes in the inheritance hierarchy.

This design pattern is such a fundamental technique for code reuse that it is likely to be applicable to an extremely wide range of object-oriented codes. In *Nektar++* it is used, to take just one example, to enable the quadrature of functions over the regions defined by differently-shaped finite elements. In *BOUT++*, this is used to perform the same operation on arrays of different type, so that there is no need to repeat the definition of, say, elementwise addition for `int`, `float` and `double`.

A UML representation of this pattern can be seen in Fig.3 (note that all UML - and other - diagrams used in this report were constructed using the free graphing software *yEd* [12]).

2.5.2 Abstract Factory

Factories are used to create instances of classes using class-specific creator functions; the objects created are of some derived type but are returned as a base-class pointer. This has obvious benefits, for example that the users of factory-created objects need only focus on those objects'

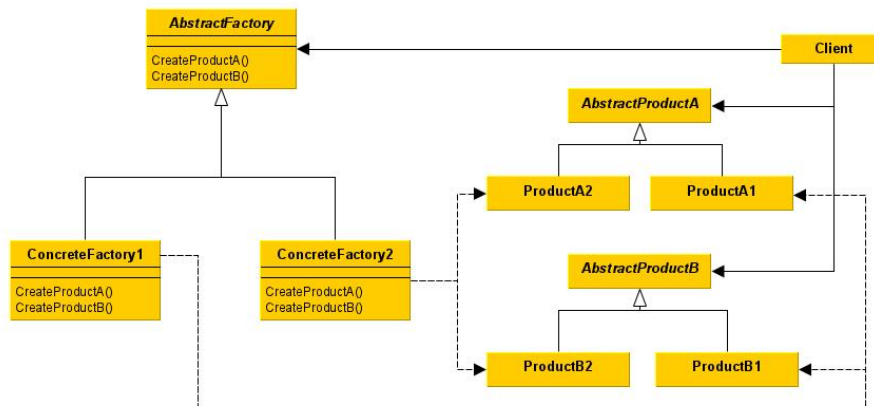


Figure 4: UML diagram for the Abstract Factory pattern.

interfaces and not the specifics of individual implementations; also, header-file dependencies are reduced, improving compile times.

Another very useful thing this enables is the disabling of certain implementations (e.g. relating to third-party libraries) during the build process; one simply avoids compiling unwanted implementations, rather than needing preprocessing flags to be insinuated throughout the code.

Abstract Factories are used in BOUT++ to allow different computational meshes. This allows the details and implementation to vary quite dramatically (compare, for example, a uniform grid and a completely unstructured mesh), while still exposing fundamentally the same `mesh` object to the rest of the code.

2.5.3 Strategy

The Strategy pattern uses delegation to vary an entire algorithm (compare the Template Method described above, which varies only part of the algorithm, in a particular way).

One salient example of this is an interchangeable matrix-free Laplacian kernel intended to supersede the existing (non-matrix-free) x86 implementation in *Nektar++*: upcoming work will focus on implementing vectorized x86, GPU, and ARM versions of the compute kernel. Here, the Strategy pattern provides an alternative to a layered architecture in enabling performance portability.

This pattern is also used in BOUT++ to implement different elliptic solvers. Thus a PETSc or HyPre solver will be implemented in the proxyapp as a new “concrete strategy” in BOUT++’s `Laplacian` class; essentially an inherited class that provides an overload to the function `Laplacian->solve` which inverts the elliptic operator using the library solver.

2.5.4 Flyweight

The Flyweight pattern is a means to allow object fine-graining whilst avoiding prohibitively large storage costs. A reference to a shared object (or the shared parts of an object) are used instead

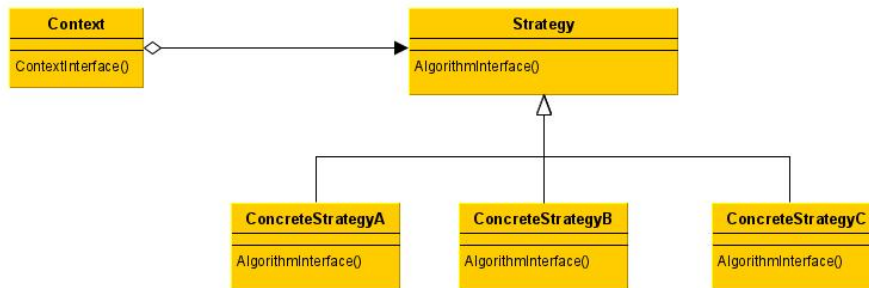


Figure 5: UML diagram for the Strategy pattern.

of the object; in this way, the unnecessary duplication of potentially large data objects is avoided. The shared quantity is referred to as the Flyweight (with mild irony, as these objects may actually be very large).

A specific example in certain types of finite-element code is the storage of the various finite-element matrices needed by each element (and of the latter there may be millions); these matrices involve a table of integrals of basis function bilinears indexed by the row and column of the matrix. The need to store a copy of the matrix for each element can be avoided, in some cases, by storing a global reference matrix and adapting this to each specific element by means of an element-specific mapping (typically using a handful of parameters specific to the geometry of the individual element).

This pattern is ubiquitous in systems containing large numbers of facsimile or closely-related objects. Note there are also considerations relating to how the objects might be created and managed taking into account the need to avoid duplication and these are illustrated in the UML diagram in Fig.6.

2.5.5 Pipeline

The *NekMesh* framework is implemented as a set of modules which operate sequentially on the mesh data in an example of the pipeline architectural pattern. See 7.

2.5.6 Modular design

The modular design pattern organizes source code into components called modules, where each module is a collection of code necessary to achieve a particular task. Variables may be local to a function within a module, or they may be module-level, that is, available for reading and writing to all functions in that module. Further, when given the `public` attribute (in Fortran), module-level variables may be edited by *all* modules. Different modules may interact via public variables, however it is better practice for all interaction to take place using function calls, even if these only trivially return the valuable of one variable from within another module.

Modules must also be self initializing and finalizing, providing routines that are equivalent to con-

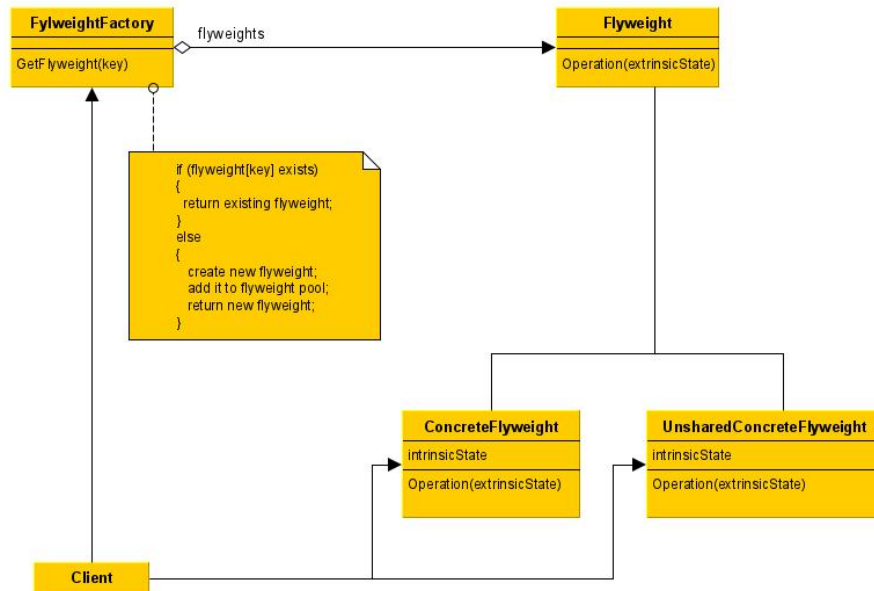


Figure 6: UML diagram for the Flyweight pattern.



Figure 7: Illustrative pipeline of the *NekMesh* process. Reproduced from Figure 4.2 of the User Guide obtainable from [3].

structors and destructors in object-oriented programming. It is the existence of constructors and destructors that distinguishes a module from a namespace.

GS2 makes use of modules, with different physical effects (e.g. collisions, $E \times B$ advection) and infrastructure (e.g. meshing, I/O), each existing within its own module.

2.5.7 Lazy initialization

Lazy initialization is the technique of delaying the call to an object's constructor or a module's initialization function until just before its first use. This allows faster speeds and lower memory consumption during code initialization, though this is a spreading of cost throughout the code's execution rather than a saving.

It also allows for a passive approach to managing dependencies between objects. For example, GS2 does not handle the dependences between modules explicitly in its initialization phase. Instead, each module's initialization subroutine `init` begins by calling the `init` routine of all the module's dependencies, before going on to initialize itself. Repeat initializations are prevented using a module-level `initialized` logical variable in each module. A similar approach is taken to uninitialization. This approach requires a developer to have some sense of the dependencies between modules in order to avoid compilation errors due to circular dependencies. However, dependencies are often straightforward, and this approach is very lightweight and easy-to-use compared to an explicit approach to managing module dependencies.

2.5.8 Dependency injection

Dependency injection is a technique where a function or subroutine receives its dependencies as input arguments. The function receiving the dependency is called a client, and the dependency received is called a service. In this approach, the client is able to use whatever dependency is passed to it, rather than needing to specify or create the dependencies itself. The advantage of dependency injection is that it allows separation of concerns – the client uses the service, but some other part of the code creates the service – increasing readability and code reuse.

Dependency injection also provides a means for older codes like GS2 to incorporate object-oriented patterns without completely rewriting legacy code. In GS2, some objects that were previously module-level variables are instead wrapped into a large `state` object. The `state` is then the input/output of subroutines, allowing the module-level variables to be incrementally removed.

3 Overarching design patterns for NEPTUNE

3.1 Domain-specific patterns for scientific computing

This section reflects the state of NEPTUNE at the time of writing: there are, as yet, no detailed designs for the proxyapps (this is the division between specification and development); and as a result, this report is correctly to be regarded as a living document. That said, some progress can be made by realizing that the forthcoming proxyapps form the prototype components for a modular scientific simulation framework fitting precisely the description of a *System for modeling and simulation* given by Sommerville [13]:

These are systems that are developed by scientists and engineers to model physical processes or situations, which include many separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.

It is for example clear at this point that the lion's share of the NEPTUNE code will be written in object-oriented languages: most of the core proxyapps will be written in C++ in order to leverage existing componentized frameworks (as mentioned above, the *Nektar++* framework makes extensive use of object-oriented methods [14]). Accordingly, much of the tried and tested pattern language developed in [15] and thereafter is relevant (and, further, the proxyapps have the potential to benefit from the advantages of the Object pattern described by Rouson et al in Ch.5 of [16]). As regards the trade-off between object-oriented coding and ultimate performance, it might be noted that these frameworks have proven scaling to at least thousands of processor cores.

Applications of software patterns to scientific computing are yet more directly germane, if sadly rare: the current edition of [16], a textbook aiming to provide a lexicon of domain-specific design patterns aimed at multiphysics applications, was published in 2014 and it explicitly highlights (p.6) the relative paucity of references for design patterns specific to scientific computing as of that date. The book culminates in the Morfeus framework, a proposal that seems to have been aimed at providing an implementation of many of the book's ideas, though it restricts itself to Fortran. Some other salient domain-specific patterns are outlined in the article by Blilie [17], including, for example, how to handle quantities with physical units. Quantities with units raise a number of concerns in scientific programming: a judicious choice can result in simplification of the code, eliminating unnecessary floating-point operations (one straightforward example is removing ϵ_0 and μ_0 when simulating Maxwell's equations, by rescaling the variables representing the magnetic field and the time), as well as improving readability; and a consistent overall strategy for units is obviously required when coupling different proxyapps. This paper highlights also the fact that design patterns can help in the division of labour between domain specialists (i.e. physicists) and experienced software engineers, in that those specialists can use the UML representations to specify the code structure in a language-independent form, to be implemented in detail by software developers with little or no domain expertise. In addition, [17] outlines two general patterns to be used for treating the generic types of problem treated by physics software: particles on one hand and continuous systems (a.k.a fields) on the other, highlighting that useful simplifications can be made in order to handle multiple timescales (for example, updating the fast particle dynamics at a

higher time resolution than that of the fields), though a note of caution is sounded if the physics of the problem is tightly coupled.

3.2 Architectural patterns

Architectural design patterns occupy a level just beneath that of the overall application framework (into which falls the question of division into libraries and appropriate directory structures) and were first defined by Garlan and Shaw in 1996 [18], there called *architectural styles*. They govern overall program control and coordination. Given the incremental development and delivery strategy adopted in NEPTUNE, a detailed discussion of the final architecture (falling under the definition of *architecture in the large* given in ch.6 of Sommerville [13]) is perhaps premature at this point. In future, wisdom in this area could be gleaned by closer study of existing architectures, for example the United States' *One Modeling Framework for Integrated Tasks* (OMFIT) system [19] and the ITER *Integrated Modelling and Analysis Suite* (IMAS) [20], the latter of which uses the Kepler scientific workflow system [21], even for fine-grained component integration. On an individual proxyapp level, called *architecture in the small*, application architectures are again determined to an extent by the re-use of existing frameworks.

One interesting scheme is the layered architectural pattern, which, as well as giving localization of change, may be used to provide performance portability; as a topical example, algorithmic content could be implemented in terms of the API of an abstraction layer such as SYCL [22], Kokkos [23] or RAJA [24]. Although there is a one-off overhead in writing the initial implementation in terms of the chosen API (though some aim to provide a C++-like programming experience), the code is thereafter portable between any of the platforms supported by the abstraction layer, with little additional programmer burden (see Fig.8). These abstraction layers are somewhat new technology at the time of writing and it is certainly possible to envisage cases in which the performance achieved might be bettered by a platform-specific implementation (though see [25] for an analysis of the relative performance of SYCL vs. native code for a standard benchmark based on HPC-style applications). This paradigm offers a potential route to a more sustainable code, since the abstraction layers avoid tying to one platform; moreover, they may also be extensible to future accelerator types. It is also a significant step in the direction of what Rouson et al [16] refer to as the holy grail of parallel scalability, to wit *automatic parallelization of code without direct programmer intervention*.

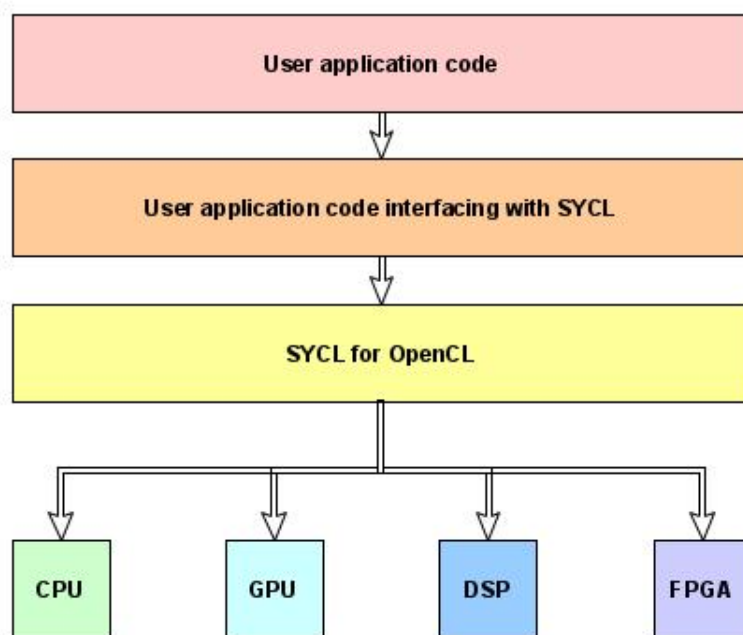


Figure 8: Diagram illustrating layered structure of an application using an abstraction layer (SYCL here). Only the layer directly addressing SYCL would require recompilation to port the code between different accelerator types.

3.3 Concurrency patterns for the exascale

Although at a hardware level, the current HPC scene is something of a menagerie, the schema common to all conventional modern systems is that of a large array of individual compute units (themselves subject to the further comminution of varying degrees of vectorization), a structure that mandates the parallelization of algorithms in order to obtain good performance. Indeed, running on such a machine, any subsection of a code that cannot be efficiently parallelized ultimately becomes a performance bottleneck (Amdahl's law). This leads to a selection of concurrency design patterns aimed at the production of efficient parallel code [26]: the main goal (alongside certain admittedly critical housekeeping duties) is to preserve strong scaling (assuming that good node-level performance has been achieved) thus making sure that the supercomputer is not functioning as less than the sum of its parts.

One key pattern is that of a Compute Kernel: a routine compiled for use on a high-throughput accelerator. This idea has its origins in the pixel shader units of GPUs and typically applies the inner loops of algorithms, being geared toward vectorized processors and SIMD by virtue of its repetitious nature. One concrete example to be incorporated in NEPTUNE spectral / hp element proxyapps is an optimized Compute Kernel for the anisotropic Laplacian operating on fields in a finite-element expansion basis. Following an initial baseline implementation on sequential x86, optimized versions targeting x86 vector intrinsics, NVidia and Arm are anticipated. The efficiencies here are furthered by exploiting the sum-factorization of fairly sparse matrices that is possible for certain classes of finite elements. Thus, the Compute Kernel pattern is expected to play a key part in obtaining good performance for NEPTUNE proxyapps, though there is a little tension between this plan – interchangeable, machine-specific kernels – and the more visionary abstraction layer philosophy described in the preceding section.

Patterns also exist for parallel I/O: the current version of *Nektar++* uses parallel-friendly mesh input files (in order to avoid the bottleneck of an individual processor parsing single-handedly a large mesh) and data checkpoint files. Mesh data is now stored in HDF5 format rather than XML; and in reading a mesh, a domain decomposition is performed, using the PT-Scotch library [14, 27].

Future exascale systems following the current pattern of large numbers of compute units are expected to mandate patterns for fault tolerance and error handling: see, for example, the discussion of resilience engineering in [13]. This will be of particular importance for any code involved in the control of a fusion reactor.

3.4 Coupling patterns

It will be necessary to couple the physics represented by initially separate proxyapps. One goal of Contract Ref. T/NA078/20 is to provide proof-of-principle coupling between the proxyapp and another compatible solver, using the coupling framework that already exists, and has been proven, within the *Nektar++* framework. This works by providing point-located field values and their respective locations via the MPI-based CWIPI framework [28, 29]. There are a number of issues here, including the computational overheads of such a scheme, and the need to convert data formats - even between two solvers using spectral / hp elements methods, there is still a large amount of choice among this family of methods and the optimal choice for one application is unlikely to

match precisely with that of another. It is also critical that spectral accuracy can be preserved during such a conversion (this would boil down to ensuring that the data sampling is of sufficient density to avoid aliasing, cf. the number of sample points needed to perform Gaussian quadrature of polynomials of a given order). More generally, compatibility of the data passed between different proxyapps could be assured by using the Gang of Four Adapter design pattern to provide a translation between any differing data requirements. In the context of the IMAS framework mentioned above, a specific example of an Adapter is provided by OMAS (Ordered Multidimensional Array Structures) [30], a Python library used to interface IMAS to other codes. OMAS implements a replica of the data structures used in IMAS, supports other popular file formats (e.g. NetCDF and HDF5) and also is capable of automatic coordinate convention transformations, grid interpolation, units conversions, and the calculation of physics quantities of interest from the fundamental variables. An additional issue for NEPTUNE is the need to couple outputs to AI / surrogate generators in future and, as these may operate at reduced numerical precision, a coupling framework capable of variable precision would seem to be indicated.

The Puppeteer pattern described in [16] provides a means of coupling multiple abstractions in a way that a) is efficient in terms of the number of couplings and b) uses only the public interfaces of individual modules and is thus a candidate for implementing a coupled multi-physics framework. The existence of this pattern provides a potential strategy for the future coupling of proxyapps into a true multiphysics framework capable of handling a potentially large number of individual physics components, for example fluid and kinetic models, particle dynamics and atomic physics.

4 Summary

This report has presented a subset of known object-oriented software design patterns that are expected to be important in NEPTUNE proxyapps as currently envisaged, using facts derived from existing codebases on which the proxyapps will draw. It is of note at this point that the progenitor frameworks fall into two categories: modern, C++-based approaches, and procedural Fortran codes. For the former, the salient design patterns are principally those that enable the reuse of components in complex, modular codebases: the Template Method, Abstract Factory, Strategy, and Flyweight patterns from the Gang of Four [15]. As for the latter, the potential for extensive code re-use is evidenced in existing codes, stemming primarily from the use of modular design. The SMARDDA framework described in this report is a concrete example of a Fortran code constructed by the aggregation of smaller objects, in a similar manner to that anticipated for NEPTUNE. Here, the use of the layered architectural pattern involves a shared subroutine library for common functionality e.g. error logging, supporting re-use. Higher in the framework, the fact that the modules are agnostic to each others' existence means that the various physics models combine in a manifestation of the puppeteer pattern, though some limitations of the current code (e.g. the existence of a single central control file) will need to be overcome in NEPTUNE, perhaps with the aid of a graph-based dependency mapping approach.

A further section has surveyed overarching patterns, though this section stops short of prescribing an overall integration framework. There is thus scope for exploration of, for example, optimal coupling using prototype versions of the initial proxyapps, with a view to address issues encountered in earlier multiphysics frameworks [1, 31, 32]. Again, certain of the time-honoured Gang of Four design patterns, for example, Adapter, are of use.

Further design inspiration may be gained by study of the game engine structure in the first annex. The second annex has presented a set of guidelines for a series of proxyapps for benchmarking exascale computing platforms, designed by the US Department of Energy's Exascale Computing Project. This provides an evolving set of quality standards and best practices which might usefully be considered during NEPTUNE proxyapp development, as well as possibly providing a more formal basis for future development within ExCALIBUR.

Acknowledgement

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

References

- [1] 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.
- [2] G. Karniadakis and S. Sherwin. *Spectral/hp element methods for computational fluid dynamics 2nd Ed.* Oxford University Press, 2005. <https://doi.org/10.1093/acprof:oso/9780198528692.001.0001>.

- [3] D. Moxey et al. Nektar++ website. <https://www.nektar.info>, 2020. Accessed: June 2020.
- [4] A programmer's guide to nektar++, 2020. Obtainable from the authors of Nektar++.
- [5] B. Dudson, P. Hill, D. Dickinson, J. Parker, A. Allen, G. Breyiannia, J. Brown, L. Easy, S. Farley, B. Friedman, E. Grinaker, O. Izacard, I. Joseph, M. Kim, M. Leconte, J. Leddy, M. Liten, C. Ma, J. Madsen, D. Meyerson, P. Naylor, S. Myers, J. Omotani, T. Rhee, J. Sauppe, K. Savage, H. Seto, D. Schwrer, B. Shanahan, M. Thomas, S. Tiwari, M. Umansky, N. Walkden, L. Wang, Z. Wang, P. Xi, T. Xia, X. Xu, H. Zhang, A. Bokshi, H. Muhammed, M. Estarellas, and F. Riva. BOUT++ v4.3.2, March 2020.
- [6] M. Barnes, D. Dickinson, W. Dorland, P. A. Hill, J. T. Parker, C. M. Roach, S. Biggs-Fox, N. Christen, R. Numata, G. Wilkie, L. Anton, J. Ball, J. Baumgaertel, G. Colyer, M. Hardman, J. Hein, E. Highcock, G. G. Howes, A. Jackson, M. T. Kotschenreuther, J. Lee, H. Leggate, N. Mandell, A. Mauriya, T. Tatsuno, and F. Van Wyk. GS2 v8.0.5, September 2020.
- [7] W. Arter, E. Threlfall, J. Parker, and S. Pamela. Report on user frameworks for tokamak multiphysics. Technical Report CD/EXCALIBUR-FMS/0022-M3.1.2, UKAEA, 2020.
- [8] 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>.
- [9] QPROG, simple demonstration of software design by aggregation. <https://github.com/wayne-arter/smardda-qprog>, 2015. Accessed: December 2020.
- [10] Code generation QPROG style, example of Braginskii plasma transport coefficients. <https://github.com/wayne-arter/smardda-misc>, 2017. Accessed: December 2020.
- [11] reduce-algebra. <https://sourceforge.net/projects/reduce-algebra/support>, 2020. Accessed: December 2020.
- [12] yEd Graph Editor. <https://www.yworks.com/products/yed>, 2020. Accessed: July 2020.
- [13] I. Sommerville. *Software Engineering. 5th Edition (10th Edition, 2017)*. Addison-Wesley, 1997.
- [14] D. Moxey, C.D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kazemi, K. Lackhove, J. Marcon, et al. Nektar++: enhancing the capability and application of high-fidelity spectral/hp element methods. *Computer Physics Communications*, 249:107110, 2020.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [16] D. Rouson, J. Xia, and X. Xu. *Scientific software design: the object-oriented way*. Cambridge University Press, 2011.
- [17] C. Blilie. Patterns in scientific software: An introduction. *Computing in Science & Engineering*, May / June 2002:48–53, 2002.

- [18] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
- [19] OMFIT website. <https://gafusion.github.io/OMFIT-source/>, 2020. Accessed: November 2020.
- [20] F. Imbeaux, S.D. Pinches, J.B. Lister, Y. Buravand, T. Casper, B. Duval, B. Guillerminet, M. Hosokawa, W. Houlberg, P. Huynh, S.H. Kim, G. Manduchi, M. Owsiak, B. Palak, M. Plociennik, G. Rouault, O. Sauter, and P. Strand. Design and first applications of the ITER integrated modelling and analysis suite. *Nuclear Fusion*, 55(12):123006, 2015.
- [21] Kepler website. <https://kepler-project.org/>, 2020. Accessed: November 2020.
- [22] Khronos Group SYCL website. <https://www.khronos.org/sycl/>, 2020. Accessed: November 2020.
- [23] Kokkos website. <https://github.com/kokkos>, 2020. Accessed: November 2020.
- [24] RAJA website. <https://github.com/LLNL/RAJA>, 2020. Accessed: November 2020.
- [25] T. Deakin and McIntosh-Smith S. Evaluating the performance of hpc-style sycl applications. *IWOCL '20: Proceedings of the International Workshop on OpenCL*, April 2020:1–11, 2015.
- [26] Software Design Pattern wiki. https://en.wikipedia.org/wiki/Software_design_pattern, 2020. Accessed: July 2020.
- [27] Scotch website. <https://gitlab.inria.fr/scotch/scotch>, 2020. Accessed: November 2020.
- [28] ONERA. CWIPI Coupling With Interpolation Parallel Interface. <https://w3.onera.fr/cwipi/>, 2020. Accessed: June 2020.
- [29] R. Casta and T. Morel. Test de la fonctionnalité ordre élevé du coupleur de codes CWIPI et Intégration dans OpenPALM. Technical Report TR-CMGC-19-95, CERFACS, 2019.
- [30] OMAS website. <https://github.com/gafusion/omas>, 2020. Accessed: December 2020.
- [31] ComPat Project. Computing Patterns for High Performance Multiscale Computing ComPat website. <https://www.compatproject.eu/>, 2020. Accessed: July 2020.
- [32] VECMA Project. Verified Exascale Computing for Multiscale Applications VECMA website. <http://www.vecma.eu>, 2020. Accessed: July 2020.
- [33] J. Gregory. *Game engine architecture, 2nd Edition*. CRC Press, 2014.
- [34] Exascale Computing Project website. <https://www.exascaleproject.org/>. Accessed: 20th November 2020.
- [35] Exascale Computing Project proxyapp website. <https://proxyapps.exascaleproject.org/>. Accessed: 20th November 2020.
- [36] Exascale Computing Project proxyapp website. <https://proxyapps.exascaleproject.org/standards>. Accessed: 20th November 2020.

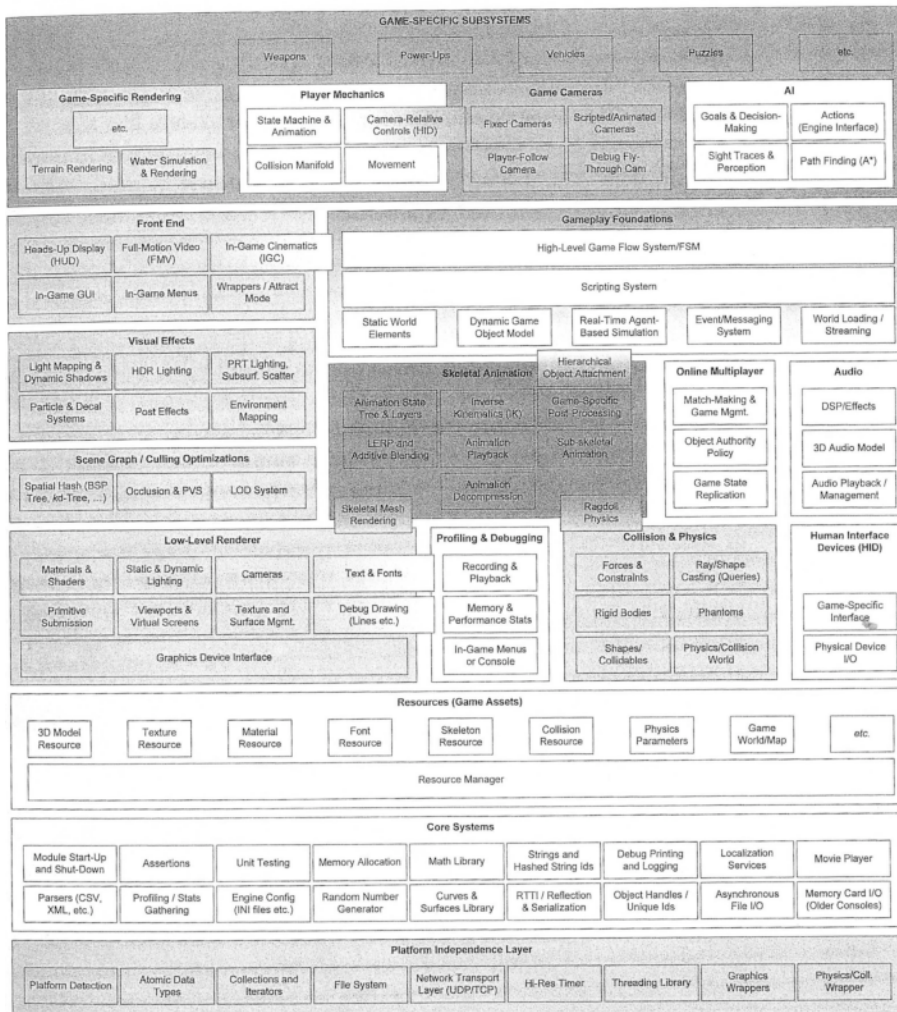


Figure 9: Diagram illustrating complexity of a game engine, reproduced from Gregory [33, § 1.6].

A Structure of a Game Engine

The proposed NEPTUNE development is of sufficient complexity that it is unsurprising that in the scientific literature there is a lack of precedent. However, in the gaming world, it is increasingly common to incorporate physical effects into the software. Indeed, a significant number of features will be shared with a game engine. As illustrated in Figure 9, commonalities include not just “collisions and physics”, but also a “platform independence” layer, likely “Third Party SDKs” (software development kits) and many of the “core systems”. It is notable that there is a scripting system included for developers, a class which might include some of the more technically sophisticated users. Unfortunately, detailed pattern information is harder to extract from Gregory’s textbook [33], so that the value of Figure 9 lies more in a demonstration that complexity can be managed using a design which is not only layered, but where each layer consists of anything up to a dozen different modules. The indicated division of relevant layers into modules is also of interest.

B ECP Proxyapp standards

The US Department of Energy’s Exascale Computing Project (ECP) [34] has curated a suite of proxyapps [35] to benchmark performance for exascale. Along with this, ECP have developed guidance for quality standards and best practices that proxyapps must meet in order to merit inclusion in the suite. Each term in the guidance is designated either required or recommended, with the expectation that the guidance becomes stricter over time, with recommendations becoming requirements.

These standards are available online [36], and are concise and readable. The standards have also been adopted by the ExCALIBUR Benchmarking working group as a basis for the rules to use within ExCALIBUR. We therefore give a condensed description.

The guidelines require that a stable release of the source code must be publicly available. This perhaps presents a problem for nuclear software, as private repositories or any other form of limited access are not deemed acceptable.

Proxyapps should be limited in size to $\lesssim 10,000$ lines of code, aiming to encapsulate repeated code or pass it off to libraries. At the same time, the proxyapp should be self-contained, with minimal dependencies handled through a package manager (Spack is recommended).

The guidelines recommend three running modes. Firstly, a (required) testing mode to verify correct compilation and kernel performance (with execution time recommended to be on the order of seconds). This mode should also include an “automatable” regression test to check the final output against a known reference result. Two further production modes are recommended to allow tracking of weak and strong scaling for (1) current workloads; and (2) anticipated exascale workloads.

The guidelines place a strong emphasis on documentation. There are basic requirements for a general description of the proxyapp, its algorithm, and what aspect of the parent app it imitates, and for documentation of the build and testing process. In addition, it is required that documentation should describe where the proxyapp is – and is not – representative of its parent. It is also recommended that documentation should detail which “simple” optimizations have *not* been applied to the proxyapp, as doing so would not be feasible in the parent.

Finally, the proxyapp must include a “figure of merit” that can be used to compare performance under different conditions. This might be based on the system being modelled (e.g. accuracy of a solution) or on what the proxyapp is stressing (e.g., run time, computational power or memory). In any case, the figure of merit should be representative of the performance of the parent application. If possible, it is recommended that the proxyapp provides a performance model of expected outcomes of the test in different scenarios.