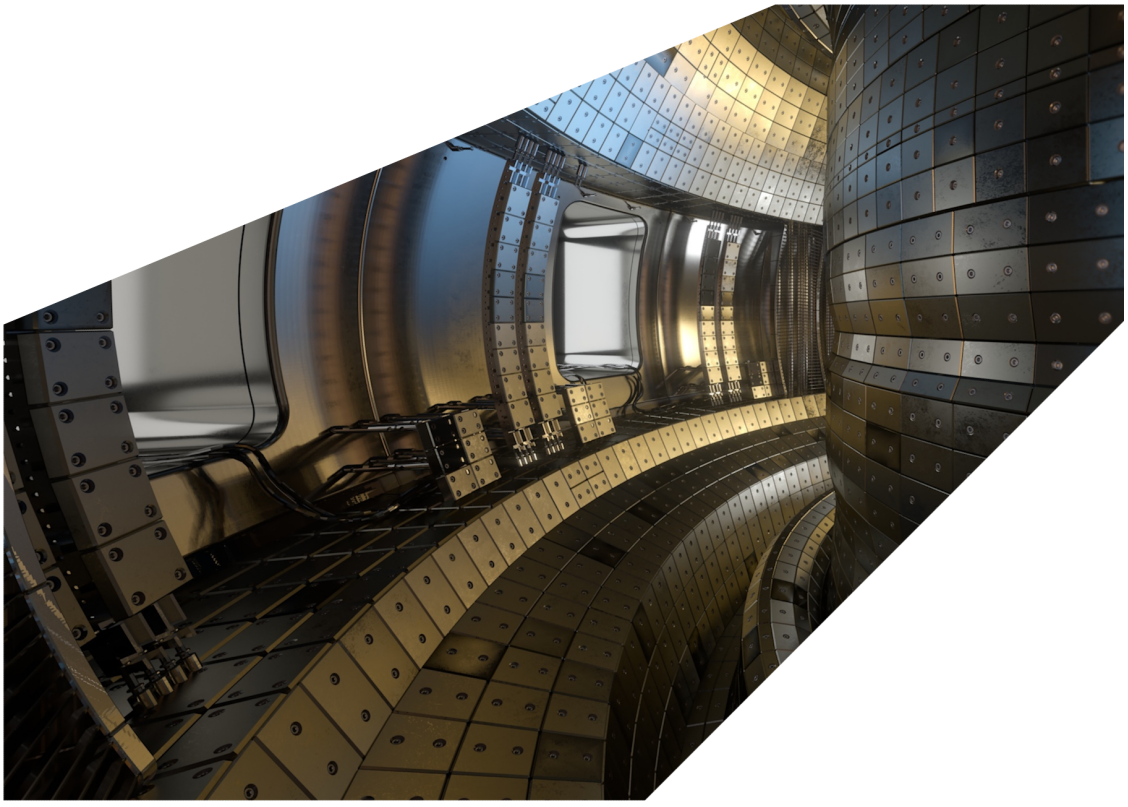
UK Atomic Energy Authority

ExCALIBUR

Report on design patterns specifications and prototypes

M3.3.2

Abstract
The report describes work for ExCALIBUR project NEPTUNE at Milestone 3.3.2. Report on design patterns and prototypes specifications for tokamak edge simulation.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0023
	Issue:	1.00
	Date:	12 August 2020
Project Name: ExCALIBUR Fusion Modelling System		

	Name and Department	Signature	Date
Prepared By:	Wayne Arter Ed Threlfall Joseph Parker Stan Pamela BD	N/A N/A N/A N/A	12 August 2020 12 August 2020 12 August 2020 12 August 2020
Reviewed By:	Rob Akers Advanced Computing Dept. Manager		16 August 2020

1 Introduction

This report has two aims as prefigured in the report [1]. The central Section 2 mainly focusses on design patterns to be used in NEPTUNE, but also pursues the prototyping process. It begins in Section 2.1 by outlining in some detail the design patterns that historically have proven useful in large software engineering projects, starting with general software design before specialising in Section 2.2 to scientific programming and multiscale physics simulation. In Section 2.3, consideration is given to the past and current contexts of software development processes: it is worth noting that, today, the HPC landscape is in something of a state of flux with the rise of heterogeneous architectures and the corresponding coding tools. There is a final summary Section 3.

The original design pattern concept is credited by Sommerville ([2], p.209) to architect and design theorist Christopher Alexander, who in his 1977 book *A Pattern Language: Towns, Buildings, Construction* [3] presented a compendium of *'certain common patterns of building design that are inherently pleasing and effective'*. His text outlines some 253 such patterns, described collectively as a 'design lexicon' or even an entirely new tongue: to quote Alexander himself *'All 253 patterns together form a language.'*.

A similar approach was applied to object-oriented software architecture in the '90s, resulting in the publication of *Design Patterns: Elements of Reuseable Object-Oriented Software* [4], the authors of which have become known as the Gang of Four (Go4, also abbreviated GoF). This book proposed 23 design patterns that are classified according to a trinity of themes: creational patterns, concerning object generation; structural patterns, to do with classes and composition, and behavioural patterns, dealing with interactions between objects. These will be discussed with a view of how germane each is to project NEPTUNE (it is also relevant that, during the time since the publication of [4], some of the original patterns have been elevated to the status of permanent language features in some object-oriented languages). Note that there is now a fourth class of design patterns (as exhibited in the Wikipedia article [5]) concerning concurrency patterns.

Aside from these general concerns, there also exist patterns which apply specifically to *scientific* software. As detailed in Section 2.2.1, the textbook by Rouson, Xia and Xu [6] develops most of a framework explicitly targeted to multiphysics workflows for Exascale HPC, starting from a foundation of object-oriented principles. Their text presents a viewpoint on the most useful Go4 design patterns in the context of scientific programming and also offers some novel patterns tailored specifically to this field.

Consideration is given in Section 2.2.2 to the ComPat project [7], which, being a framework and software suite to study multiscale fusion plasmas on HPC systems, constitutes a further specialisation in the direction of project NEPTUNE. Design patterns here are used to provide separation of concerns, with the full physics being represented by an interchangeable set of submodels in a coupling framework. Attention is given to the preservation of good scaling to Exascale HPC in such a framework. Section 2.2.3 gives more details of the Verified Exascale Computing for Multiscale Applications (VECMA) project [8] which essentially represents a continuation of ComPat, using updated versions of the same components eg. the multiscale coupling framework MUSCLE. The associated VECMA toolkit [9] provides a platform for VVUQ. Although not a pattern in the strict sense, this suite is of interest as it employs several more fundamental patterns (for example the Go4 *Adapter*) in a framework which includes the management of a set of interrelated jobs on

either a cluster or an HPC machine, and the provision of graphical and Python interfaces.

In software engineering, prototyping is normally described as a quick way to produce something that potential customers can explore, primarily with a view to improving the user-interface. Only Booch [10, § 8.1] admits that it may help the developer understand the technical aspects of the problem better, and this work predates design patterns. As indicated in the previous report [11], little formal description has been found of the role of prototyping in scientific computing. However the highlighted 'idea' paper of Dubey and McInnes [12] does include both these applications and is further discussed in Section 2.3.1.

2 Task Work

2.1 General design patterns

2.1.1 Gang of Four

The design patterns introduced by the Gang of Four will be described briefly, bearing in mind their likely relevance to project NEPTUNE. It is notable that, in the light of the decades of software engineering practice that have followed the publication of [4], developments in object-oriented languages themselves have incorporated some of the patterns directly. Note that the Wikipedia articles on these patterns are largely of good quality and provide also UML representations and (polyglot) code examples. Note additionally that the Wikipedia article on *Software design pattern* [5] contains patterns in addition to those in the Go4 text, particularly a fourth category, concerning concurrency patterns, which would seem quintessential to our case.

Creational Design Patterns

These are associated with object creation.

Abstract factory - this provides an interface for creating *families* of related objects without specifying their concrete class. Derived factory classes create, for example, documents with a style (fonts etc.) common to the concrete factory, the documents being represented by derived types eg. fancyLetter, businessLetter, ... , businessReport. The client only knows about the abstract document types (letter, report ...) and the abstract factory, the concrete choice of which determines the style of documents received by the client.

Builder - this separates the construction of a complex object from the representation of that object. Instead of having classes call the object's constructor, they call a separate Builder method that constructs the objects. Altering this Builder means the object can be constructed differently without having to change the object; also, selecting a different Builder means that the type of object created can be changed.

Factory method - this decouples the construction of objects from the objects themselves, allowing the creation of objects whose concrete type is not determined at compile time.

Prototype - this creates objects by cloning an existing object. The point is that a Clone() method may be called on an object of unspecified concrete type - this is similar to the aim of the Factory method above.

Singleton - this restricts a class to having a solitary instance (or none at all). Constructors are hidden by making them private and the sole instance is accessed via a GetInstance() method, which provides global access and can be used to provide lazy initialization - creating the singleton only when it is required.

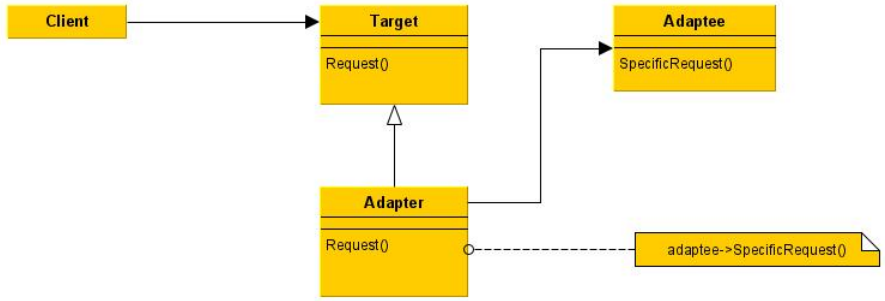


Figure 1: Class diagram for the (object) Adapter pattern (there is also a very similar class Adapter version).

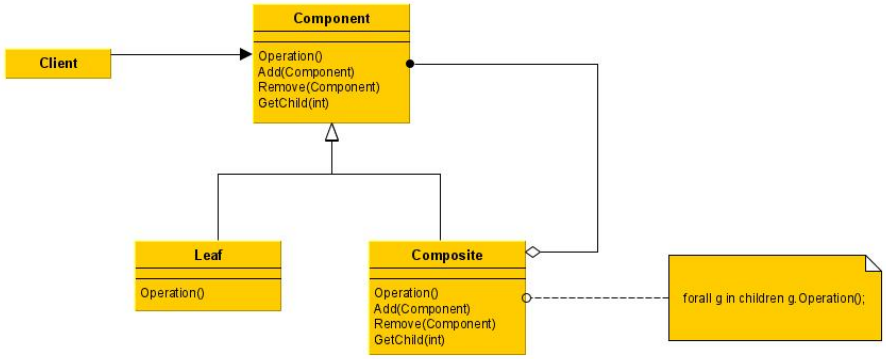


Figure 2: Class diagram for the Composite pattern.

Structural Design Patterns

These are to do with composition, inheritance etc.; some of them seem very straightforward. Others appear directly relevant for NEPTUNE : these examples are presented first and are illustrated with the appropriate UML diagram (all of which were created using the free graphing software *yEd* [13]).

Adapter - sometimes known as a wrapper, an Adapter is simply an object that translates between otherwise-incompatible interfaces, thus allowing objects to co-operate without changes to their interfaces. Arguably one goal of good interface design is to avoid the need for this pattern; however, and with no pejorative intended, since pre-written third-party code may be used in NEPTUNE , it may well be necessary to use Adapters. It is also of note that modern-day unified APIs for heterogeneous computing are, fundamentally, implementations of this pattern (eg. a SYCL plug-in acts basically as an Adapter for CUDA in order to harness NVidia GPUs).

Composite - this means the provision of a unified interface for part and whole; eg. resizing a collection of shapes is simply resizing each shape. Branches forward request to leaves (shapes in this case); another example is printing a collection of graphics objects (which boils down to printing each object). This is certainly relevant to a graph-based approach and indeed it is used in the *Arcos* framework.

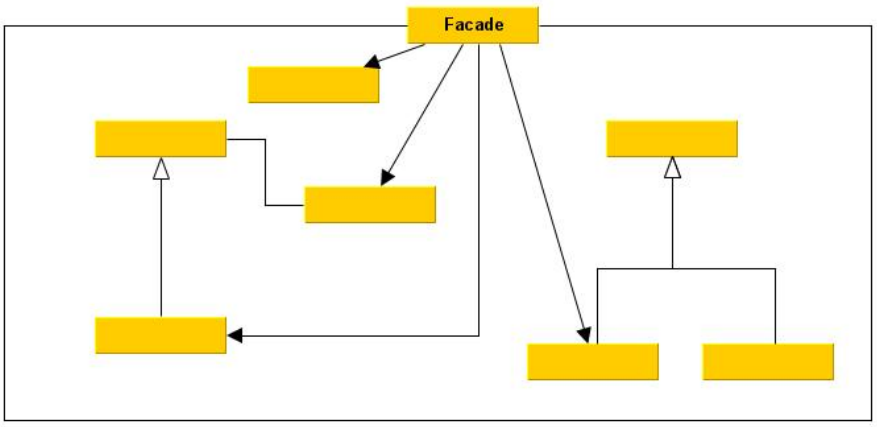


Figure 3: Class diagram for the Facade pattern.

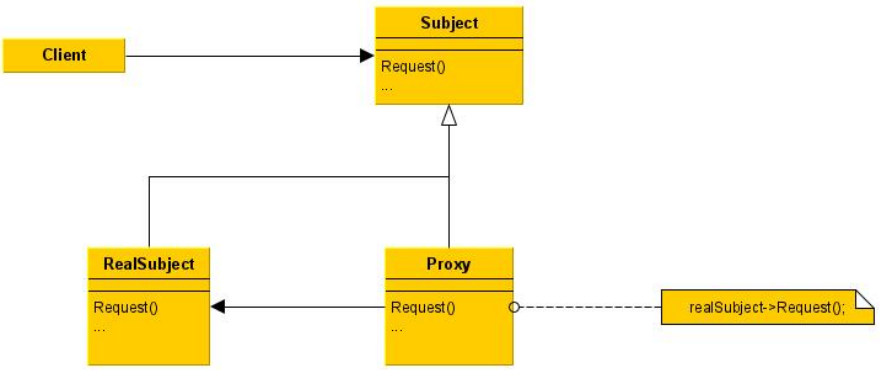


Figure 4: Class diagram for the Proxy pattern.

Facade - this means the provision of a simplified API that delegates to interfaces of subsystems; it can thus be used to provide a greater or lesser degree of automation and it also means that all controls of a set of related subsystems are located conveniently in one place (perhaps a reasonable analogy is having all the controls needed to drive a car in front of the driver). This paradigm seems central to making software easy to use and understand and thus seems to be of great import.

Proxy - this means a class functioning as a stand-in for something else and sharing the same interface. Clients cannot necessarily distinguish between the object and its proxy, but the latter could provide, for example, additional checking for reasonableness of inputs, or security checking, or implement load-on-demand of data. The idea of loading large data objects only when explicitly required is of great utility, particularly in a parallel computing scenario.

The remaining structural patterns were judged to be of lesser interest for project NEPTUNE .

Bridge - this decouples abstraction from interface; the bridge object associates an abstraction (=interface) with its implementation at run-time. This seems very similar to virtual functions (run-time polymorphism). It seems that the goal of this pattern can be accomplished with virtual methods.

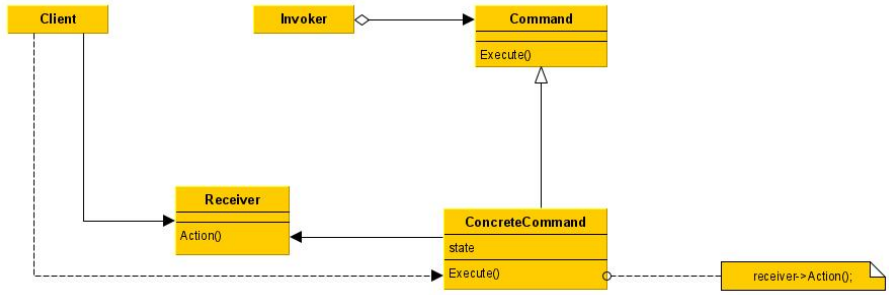


Figure 5: Class diagram for the Command pattern.

Decorator - this means adding functionality to a particular object without affecting the behaviour of other objects of same class. The Decorator object just wraps the class and either overrides or forwards requests. The constructor of the Decorator has a reference to the decorated object as its argument. This seems a rather *ad hoc* way to add functionality *ex post facto*.

Flyweight - this seems to mean the sharing of data between similar objects to save memory, eg. not storing all the letters in a document but rather storing each letter once (ie. an alphabet!) and then the document is represented by a collection of references to these. It is clearly useful when objects occur in large numbers (again, letters in a document). The jargon is that intrinsic data (eg. the glyph for each letter) can be shared; extrinsic data cannot. Note that, perhaps counter-intuitively, Flyweight objects are generally big ('bulky data'); they are addressed by references that store only the extrinsic data (in the document example, this would be the positions of the letters). The reasoning behind the name seems to be that the amount of data stored would be much larger or heavier if the object was fully described at each occurrence. Thus in the example of letters in a document, if each were stored with its case, font type and size, this would use up far more store than stating the font at the beginning.

Behavioural Design Patterns

Three of these patterns were judged to be of somewhat greater interest:

Command - this means having an encapsulated Command object (constituted by all the information needed to perform an action). This can be used in scripting, macro recording, multi-level undo, parallel processing, thread pools ... this is, potentially, a key part of overall program coordination.

Observer - the subject (an object) maintains a list of dependants and notifies them automatically of any state changes. It enables commands to be broadcast to all relevant parties (one might draw the analogy of hormones in animal bodies, or pheromones in eusocial insect colonies). This would appear to be a powerful tool for program coordination. Also, it is very useful in GUIs for making sure all aspects of the interface are updated to reflect changes in user inputs or data.

Strategy - also known as Policy, this pattern enables the selection of an algorithm at run-time, perhaps through the use of a run-time-assigned function pointer. It allows the algorithm to vary independently from the clients that use it. This can be very useful for the dynamical coupling of objects of arbitrary type.

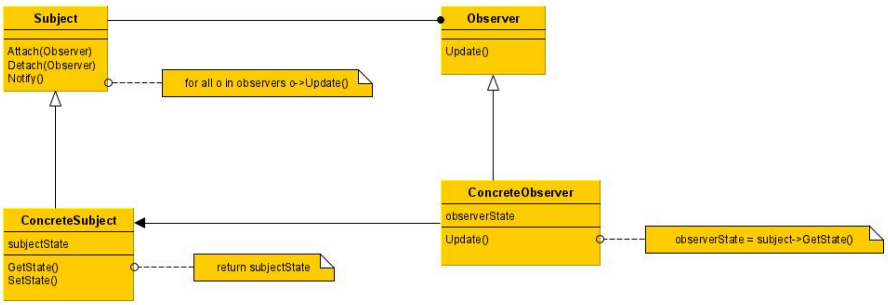


Figure 6: Class diagram for the Observer pattern.

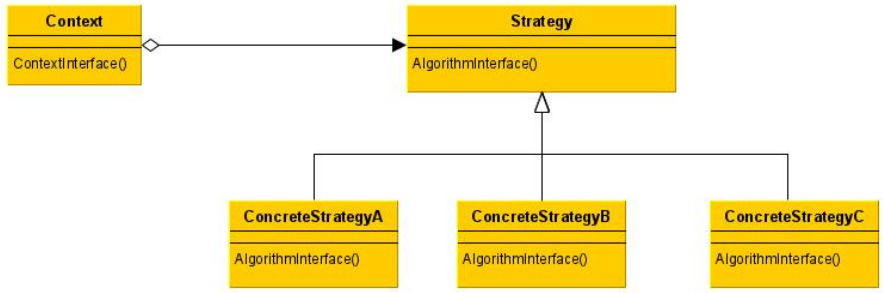


Figure 7: Class diagram for the Strategy pattern.

In the interest of completeness, the remaining behavioural patterns follow.

Chain of Responsibility - in this pattern, a command object emitted by a source is passed along a (possibly branching) chain of potential processing objects, representing a dynamical 'if ... else if ... else if ...'; it seems to require a nerve-like chain to conduct the command. The removal of close coupling appeals, though the Observer pattern appears more flexible as it removes the need for the chain / tree of command.

Interpreter - this means basically providing a domain-specific language with its own grammar, rather than hard-coding each individual command. Example: instead of writing DO_A_WITH_B() the grammar is Execute(DO A WITH B). This is useful because grammar can make the driver syntax much cleaner.

Iterator - this means the ability to access elements of an aggregate object without exposing the underlying representation. Note that one criticism of Go4 is that it listed features that ought to have been part of C++ and now it seems that some of the patterns have been incorporated in the language - I believe the STL Iterator implements this pattern. This pattern would appear to be already present in C++, in a sense; otherwise, it seems simply to mean that the API of an aggregate object should hide the implementation, which is something of a given.

Mediator - this is intended to reduce close coupling between objects, as they communicate via the mediator, rather than directly. Again it is nice that close coupling is reduced but this scheme does introduce the central control object (mediator), which may not always be desirable. A single mediator object may have unwanted consequences in a massively-parallel environment (though

one can imagine hierarchies of mediators).

Memento - this involves the provision of the ability to restore an object to a past state (eg. for rollback undo). It seems that an object must store a copy of its past self and some additional information about how to get from that copy to the present version (mathematically, this seems like the Markov property).

State - means allowing an object to alter its behaviour when its internal state changes. One defines State objects (and derives special cases of States), then classes delegate state-specific behaviour to their State object. The advantages of this pattern are presently unclear; it seems there are also cases when one wants objects *not* to have internal state data at all ...

Template method - here, the skeleton of a method is located in a superclass; the 'template' that is never overridden, this contains invariant parts of the algorithm. Its components *are* overridden; for example, one might have a computer game with invariant structure Initialize(), Startplay(), Endplay(), all of which are overrides. This might be quite useful for different physics solvers; it is obviously logical to have only one copy of common code.

Visitor - this means adding new virtual functions to a family of classes without modifying the classes or defining a new operation for a class without changing the class: one calls a Visitor method on an object and the Visitor does something to the object. It is useful for performing an operation on a wide variety of classes without having to put a new method in each class. It is obviously good 'economics' to have only one copy of common code.

2.1.2 Post-Go4 Patterns

The Wikipedia page on *software design pattern* [5] outlines a further 16 *concurrency patterns*, defining them as *'those types of design patterns that deal with the multi-threaded programming paradigm'*. Five of these are to be found in the text *Pattern-Oriented Software Architecture Vol.2: Patterns for Concurrent and Networked Objects* by Schmidt, Stal, Rohnert and Buschmann (another gang of four) [14]. Given the lie of the current high-performance computing landscape, these would seem of particular relevance to NEPTUNE and so, a brief outline of each is given here.

Active object - method execution is here decoupled from method invocation in order to avoid race conditions on an object's data (or indeed the need to write additional code to synchronize such cases). Instead of object methods acting directly on the object, the work is delegated to a task scheduler: the upshot is that only one thread may ever modify the internal state of an object. The pattern uses *asynchronous method invocation*, itself a further pattern.

Balking - this means that an action is performed on an object solely if that object is in a particular state. There is some debate as to the validity of this as a pattern and one might argue that an API should not support requests that are 'invalid' in the sense of being inconsistent with the internal state of the object. Regardless, the pattern can be used to ensure that a worker object only accepts new jobs if it is not in a 'busy' state, for example.

Binding properties - though the Wikipedia article is somewhat unclear, it seems that this means having multiple observers enforce synchrony (or some other consistency) of the properties of some other object; it would give a way of automatically enforcing mutual consistency of properties

of different objects.

Compute kernel - this is a routine compiled for execution on an accelerator, for example a shader routine of a GPU that is called by the main program. The kernel corresponds roughly to the 'inner loop' ie. the actual work. SIMD intrinsics would seem an example of this paradigm also. More succinctly: the same computation, many times in parallel.

Double-checked locking - this seems to mean doing a check on a lock using a criterion (known as the 'lock hint') before explicitly acquiring the lock. Locking only proceeds in the lock hint indicates that locking is required. It is a method of reducing overhead by doing a quick (and possibly dirty - the Wikipedia article cautions that the pattern can be unsafe on certain combinations of hardware / software / language) check before proceeding with an operation.

Event-based asynchronous - this seems intended to address problems with the *asynchronous method invocation* (referred to above, in *Active object*), also known as simply the *asynchronous* pattern, which means that a long-running method returns immediately a 'working ...' response and notifies the caller on later completion of the work. One says the caller is not 'blocked' (that is the meaning of asynchronous; the Wikipedia article [5] does not detail the precise meaning of *event-based asynchronous*).

Guarded suspension - seemingly an alternative to the *balking* pattern, this pattern prevents acquisition of a lock until a precondition is satisfied (it is a way of making a program wait before doing something, for example waiting for an queue to contain an object before attempting to operate on an object in the queue).

Join - *join-patterns* refer to the high-level (as opposes the nitty-gritty of threads and locks) practice of making programs work in parallel by *message passing*, enabling scalability. The article linked from [5] contains much detail on *join-calculus*.

Lock - this is a foundation of multi-threaded programming: one thread puts a 'lock' on a resource, preventing other threads from accessing or modifying that until the lock is released by the subject thread.

Messaging design pattern (MDP) - this describes how a communications protocol works, for example request-response (like HTTP) or one-way (like UDP), or request with optional response.

Monitor object - this seems to refer to using mutexes (which can be implemented simply as the acquisition of a lock) in order to permit access to the methods of an object on a serial basis only. Succinctly: an object whose methods are subject to mutual exclusion.

Reactor - this is *'an object providing an asynchronous interface to resources that must be handled synchronously'* (Wikipedia). The object (or *service handler*) is said to *demultiplex* incoming requests and pass them on in a synchronous manner.

Read-write lock - this means the provision of concurrent read but serial write access.

Scheduler - this means the means by which work is doled out to threads; clearly the issue of load-balancing and overall rate-limiting steps raise their heads here. Aside - this concept also allowed multi-tasking back in the days of single-processor architectures.

Thread pool - this refers to a set of threads that are available to be assigned work; the threads may be organized in some way eg. as a queue.

Thread-specific storage - 'Static or 'global' memory local to a thread.' ([5]).

2.2 Scientific patterns

2.2.1 Rouson's patterns

The book by Rouson, Xia and Xu [6] could be seen as template for designing and writing the software to be written for NEPTUNE, since it starts with an introduction to object-oriented programming and finishes with a description of the Morfeus multiphysics framework aimed ultimately at Exascale HPC. Since most code examples are presented in both languages, the book may also be regarded as a 'Rosetta stone' for scientific programming in C++ and Fortran 2003, see Table 1, and including a chapter (§ 11) devoted to discussion of the interoperability of the two languages. Notably too, Rouson et al [6, §,4.2.3] describe how one of the first examples of object-oriented scientific programming, described in the book by by Gardner and Manduchi [15], is an application in nuclear fusion, namely a data acquisition code written in Java.

Table 1: Rouson, Xia and Xu as a 'Rosetta Stone' – Table 2.1 from [6] on Object-Oriented Programming nomenclature.

Fortran 2003	C++	General
Derived type	Class	Abstract data type (ADT)
Component	Data member	Attribute
Class	Dynamic Polymorphism	
select type	(emulated via dynamic_cast)	
Type-bound procedure	Virtual Member function	Method, operation
Parent type	Base class	Parent class
Extended type	Subclass	Child class
Module	Namespace	Package
Generic interface	Function overloading	Static polymorphism
Final procedure	Destructor	
Defined operator	Overloaded operator	
Defined assignment	Overloaded assignment operator	
Deferred procedure binding	Pure virtual member function	Abstract method
Procedure interface	Function prototype	Procedure signature
Intrinsic type/procedure	Primitive type/procedure	Built-in type/procedure

In this report, the focus is on design patterns, to which concept Rouson et al [6] contribute the idea of "domain-specific design patterns", where the domain is the scientific one encompassing NEPTUNE. Rouson and coworkers also describe language-specific design patterns, namely *Surrogate* [6, § 7] for Fortran 2003 and *Compute Globally, Return Locally* (CGRL) [16] for Co-array Fortran (CAF). These are necessary because of deficiencies in the programming language(s) and will not be further discussed.

In Rouson et al [6, § 5], the concept of object (which they pedantically describe as a language-specific pattern because it is not fundamental to either the Fortran 2003 or C++ languages) underlies their discussion of design patterns. The book also emphasises the importance of UML in describing patterns [6, § B]. Table 2 which is reproduced from [6], summarises the generic design

patterns found useful for scientific work in the book, either directly or because they inspired related scientific design patterns.

Table 2: Useful Go4 design patterns and summary descriptions – Table 4.1 from [6], augmented with pointers to sections in the book

Pattern	Allow Varying of
Factory Method	Subclass of instantiated object § 9
Proxy	Location or method of accessing an object
Composite	Object structure and composition (§ 9.2)
Facade	Interface to a subsystem § 6
Iterator	How an aggregate's elements are accessed § 5.4
Mediator	How and which objects interact with each other § 8
Strategy	An algorithm § 7
Template Method	Step(s) of an algorithm § 9.4
Singleton	Global variable(s) § 9.3
Abstract Factory	Factory method § 9

The *Strategy* pattern is shown to be directly useful in scientific programming, with the example of enabling an easy switch between different methods for integrating a PDE forward in time, such as forward Euler or 2^{nd} order Runge-Kutta. The next novelty arises from the perceived need for a multiphysics program with 'separation of concerns' to be adaptable at the level of mathematical abstraction of the vector calculus, or more precisely the tensor calculus. This leads to the introduction in ref [6, § 6] of the Abstract Data Type or ADT calculus for PDEs, referred to as 'abstract calculus' where say partial derivative with respect to time $(\partial/\partial t)$ and Laplacian operator ∇^2 have special representations in software, cf. overloading operators. Externally, the *Abstract Calculus* pattern exhibits many aspects of the *Facade* design pattern, whereas internally the separate operators may be regarded as *Template Methods*.

There is criticism of the *Mediator* pattern see Figure 8, because of the linear growth in its complexity with the number of objects it has to connect, hence the introduction of the *Puppeteer* pattern, see Figure 9. The key distinction is that the connected objects need know nothing about the *Puppeteer*.

The application of *Abstract Factory* pattern to *Abstract Calculus* is to define discrete properties to be applied to a field in order to solve a PDE. Rouson et al give the example of the flow $u(x,t)$ as a solution to Burgers' equation, as the 'Field' returned by reference from the *Factory Method* create() in Figure 10, where the *Abstract Factory* **FieldFactory** uses the concrete Periodic6thFactory which aggregates (discrete) periodic boundary conditions with a 6^{th}-order Padé approximation. The point is that other boundary conditions and approximations may be easily substituted for periodic and 6^{th}-order Padé respectively, but the create() interface remains the same. Indeed strictly speaking, these other substitutions must be available to satisfy the definition that an *Abstract Factory* is capable of creating a family of related objects.

Software described in the Rouson et al book apparently culminates with the Morfeus framework for multiphysics. The reason this otherwise seemingly very relevant framework is not described in the report [17] is that on close inspection, what ref [6] contains is more a proposal to implement the design patterns in the book than a finished project. The only related public repository found

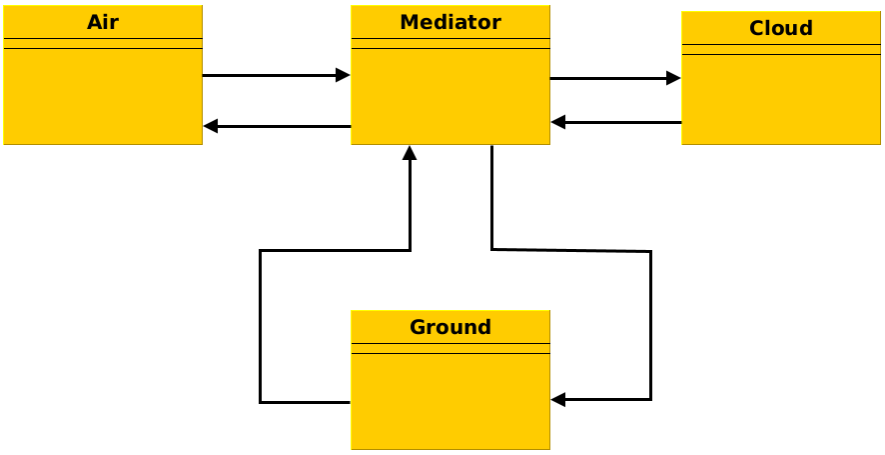
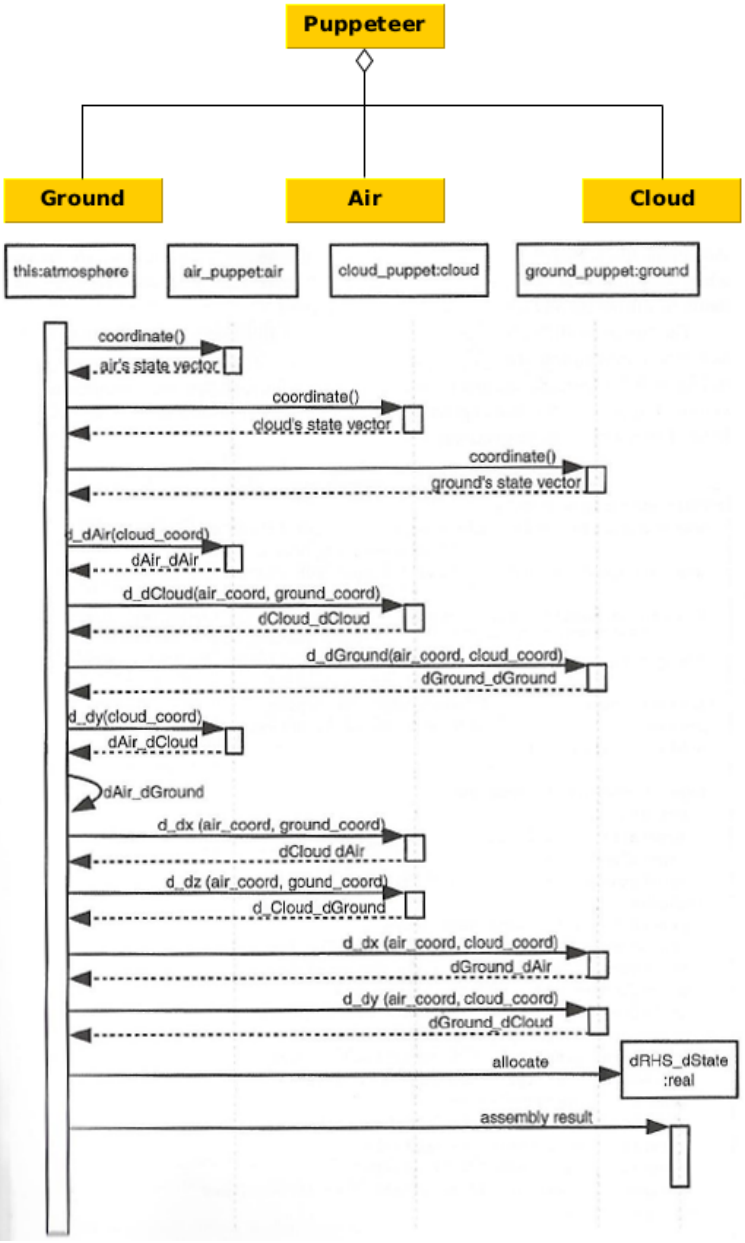


Figure 8: UML class diagram for the *Mediator* pattern showing application to the problem of accessing an atmospheric state defined by separate data describing air, cloud and ground properties – After Fig. 8.1 from [6].

was actually set up very recently (2020) by Rouson, and contains Fortran 2003 code only [18].

(a)



(b)

Figure 9: At top (a), UML class diagram for the *Puppeteer* pattern – after Fig. 8.2 from [6]. Below (b), UML sequence diagram for the *Puppeteer* pattern – Fig. 8.3 from [6], when specifically derivative information is required so the puppeteer can form a Jacobian matrix.

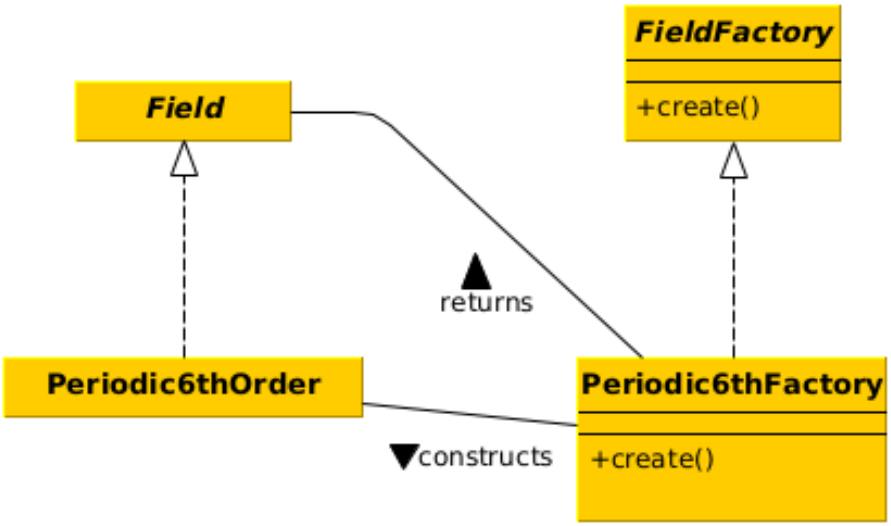


Figure 10: UML class diagram for the *Abstract Factory* pattern showing application to solving a PDE – after Fig. 9.1 from [6].

2.2.2 ComPat

The ComPat project [7, 19] is an extreme scaling design pattern targeting the multiscale modelling of magnetic confinement fusion (MCF) plasmas. Models of MCF plasmas span lengthscales from the electron gyroradius $\sim 10^{-6}$ m to the device size ~ 10 m, and timescales from the electron gyroperiod $\sim 10^{-10}$ s to the confinement time ~ 10 s. Including even smaller length- and timescales from atomic and sheath physics would be desirable. The different physical effects within an MCF plasma are also characterized by models of different dimensionality, with five-dimensional gyrokinetic turbulence, two- or three-dimensional edge transport, and one-dimensional core transport models.

The ComPat project adopts a submodel approach to modelling MCF plasmas, using single-scale submodels provided by different codes, and coupling these together by passing relevant information in an agreed format through middleware. This approach has a number of benefits. It allows the separation of concerns: using existing codes as submodels means that the components are developed by domain specialists, and are therefore better tested and validated. Discrete submodels are also easier to maintain and optimize. Further, it allows users to change which code is used for each submodel, so that a user could, for example, choose between continuum and particle-based solvers for the gyrokinetic turbulence, or indeed, between gyrokinetic and gyrofluid models. The drawback of separating codes in this manner is that the process of coupling codes introduces complexity and runtime overheads.

ComPat is a framework and software suite designed to facilitate studying multiscale fusion plasmas on HPC systems. It has two major objectives [20]: (1) to provide a collection of methods and software to aid the development of component-based multiscale simulations; and (2) to create generic transformation and optimization methods to ameliorate coupling overheads and improve the simulation's runtime (and/or other performance metric) on a targeted set of execution platforms. Different workflow configurations may be used depending on what the user wishes to optimize, for example, minimizing wall clock time, minimizing total energy consumption, or maximizing scaling efficiency.

Implementation The ComPat software stack has several components. Firstly, it uses a domain-specific language, the Multiscale Modelling Language (MML) and its representation in XML (xMML) to allow developers to give a high-level description of submodels and their interactions. It uses the tool jMML to generate a topology, task-graph and skeleton configuration file for the coupling framework. The coupling framework used is MUSCLE2 (Multiscale Coupling Library and Environment). This itself has two parts, a library for data exchange between submodels, and a runtime environment to manage each submodel's execution on (possibly) distributed resources. The library has APIs in C, C++, Fortran, Python and Java, with functions allowing users to query parameters, send and receive data, and log and stage files. This component-based approach means that the framework is agnostic towards which submodels are actually used. Submodels (called *kernels* in MUSCLE2 nomenclature) therefore may be swapped in and out, and different multiscale models treated within the same framework.

ComPat uses QCG middleware [21], as a resource broker to set-up and execute jobs. The QCG client provides a simple interface to machines through batch scripts or XML files. This allows cross-

cluster jobs to be configured with a single script and to be monitored from a single workstation.

Performance Alowayyed *et al.* [22] identify three Multiscale Computing Patterns (MCPs): *Heterogeneous Multiscale Computing* (HMC), *Replica Computing* (RC) And *Extreme Scaling* (ES). ComPat follows the *Extreme Scaling Pattern*, as the computing cost of one submodel (called the *primary*) dominates that of all other submodels (called the *auxiliaries*). In ComPat, the primary is the model for microscale turbulence. Through its use of multiple computing resources, the extreme pattern introduces an extra layer of parallelism at the code coupling level, meaning that coupling overheads may be hidden, and the coupled code may even outperform a monolithic implementation.

In Luk *et al.* [20], a further optimization is found by solving the submodels in an “asynchronous” fashion. A simple first analysis of the data dependencies in ComPat showed that the different models needed to be executed sequentially, and that consequently the primary microturbulence submodel was often idle, waiting for other submodels to finish. However, by considering the timescales of the auxiliary submodels, Luk *et al.* argue that the longer-timescale equilibrium and transport solvers do not change their input to the turbulence solver significantly from one time step to the next. Therefore, they introduce an ”asynchronous” mode, where the microturbulence model takes equilibrium and transport data from the *previous* timestep. This removes the data dependency, and allows the submodels to run in parallel. This asynchronous workflow decreases the total wall clock time by around 6% without appreciably changing the simulation results.

Despite this improvement, the runtime is still limited by the scaling performance of microturbulence submodel, which only scales well to 16 cores per flux tube. This emphasises that in the *Extreme Scaling Pattern* it is necessary to optimize both the workflow and the raw scaling performance of the submodels (that are on the critical path) in order to obtain good scaling performance at Exascale.

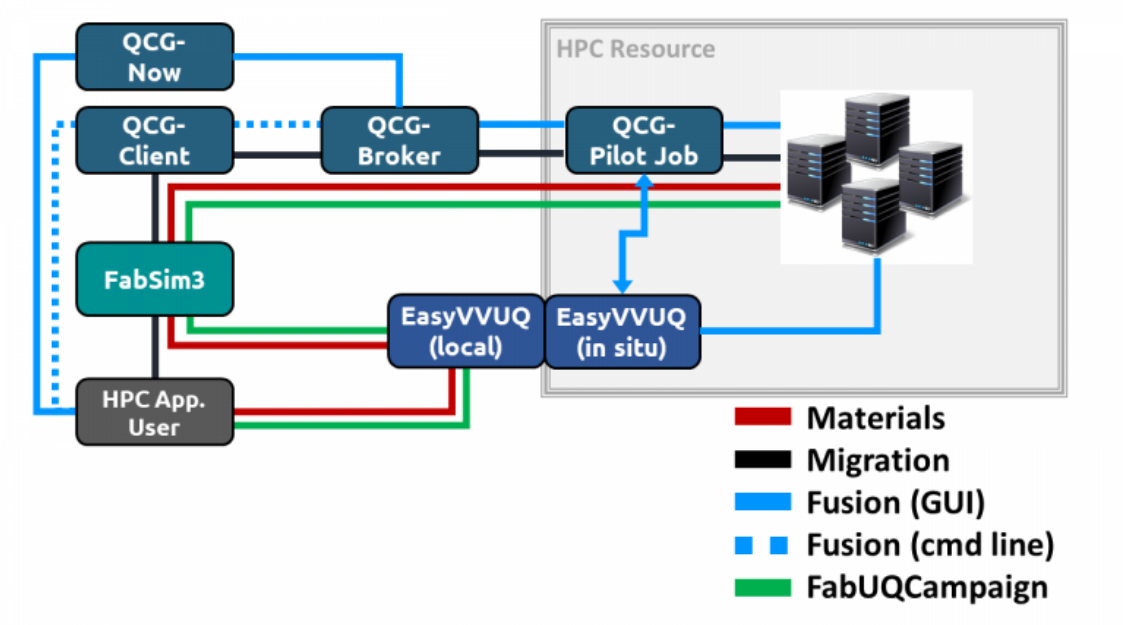


Figure 11: Usage of VECMAtk, taken from the website [9].

2.2.3 VECMA toolkit

The Verified Exascale Computing for Multiscale Applications (VECMA) project [8] in many important respects follows-on from the Computing Patterns for High Performance Multiscale Computing (ComPat) project [7]. Thus the ComPat website [19] lists the following software

- FabSim
- QCG – Quality in Cloud and Grid
- MUSCLE 2 – The Multiscale Coupling Library and Environment

whereas the VECMA toolkit (VECMAtk) website [9] lists (and provides links to):

- FabSim3
- QCG Pilot Job (QCGPJ), QCG-Now, QCG-Broker and QCG-Client
- MUSCLE 3
- EasyVVUQ – Facilitate verification, validation and uncertainty quantification (VVUQ)
- EasyVVUQ-QCGPJ

where the EasyVVUQ software represents the VVUQ capability developed by the VECMA project.

The roles played by FabSim, QCG and MUSCLE 2 appear to have been taken over by FabSim3, QCGPJ and MUSCLE 3 respectively in the more recent project, and as they are described in Section 2.2.2 will not be described further.

MUSCLE 3, updating MUSCLE 2, is effectively a separate development from VECMAtk, that provides a coupling capability between different codes. It is of additional interest because it has the capability not only to implement but also to create couplings using the Multiscale Modeling and Simulation Language (MMSL) which was developed in tandem with the original MUSCLE. Unfortunately, as of mid-2020, there is no further funding for developing or supporting MUSCLE 3, but it will continue to be supported by Lourens Veen at the Netherlands eScience center in his "spare time".

Figure 11 indicates how the components of VECMAtk, apart from MUSCLE 3 (which would be confined to the HPC Resource), interact. VECMA toolkit is not a software pattern in the strict sense in that it typically spans a network. The QCG software devloped in Poland at the Poznan Supercomputing and Networking Center (PSNC) provides the user with a system-independent approach to the problem of executing a set of interrelated jobs on a cluster of machines or as indicated in the figure, a HPC machine in the background. QCG-Now is a desktop GUI interface to the other components of the QCG-Client, of which the most important for VECMAtk is QCGPJ. EasyVVUQ [23] is a Python library to help produce computational "campaigns" to enable UQ of simulations on HPC machines in combination with QCGPJ, with which it is specially packaged as EasyVVUQ-QCGPJ. The intent appears to be that EasyVVUQ-QCGPJ will replace FabSim3 which provides similar functionality also using Python.

The basic pattern in EasyVVUQ-QCGPJ is in fact the straightforward [24]

1. Produce ensemble of calculations ("campaign"),
2. Execute unmodified simulation code for each set of parameters
3. Correlate results of ensemble

so that EasyVVUQ-QCGPJ is non-intrusive in the sense that like most UQ packages it simply wrappers existing code. Indeed it exploits Feinberg's chaospy library [25] to help generate the ensembles via sampling from distributions provided by the library [26].

2.3 Prototype Specification

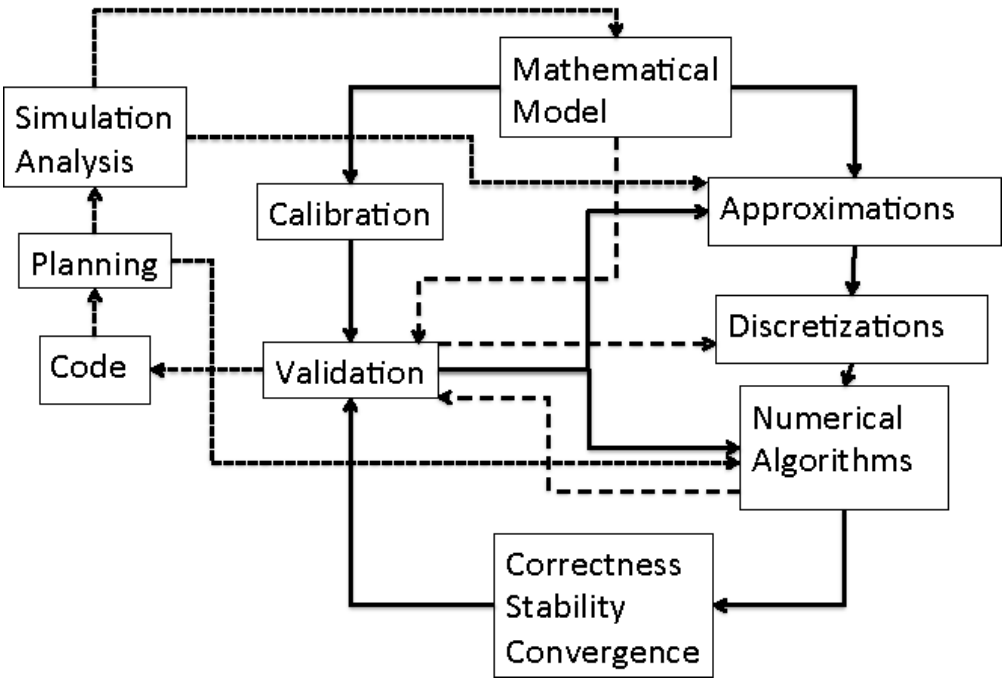
2.3.1 Dubey's patterns

There are many well-documented software development processes, such as waterfall, spiral, rapid application development (RAD) and agile. The purpose of these processes is to decompose software development into its constituent components, so that developers can focus on the quality of each component separately, raising the quality of the software overall. Dubey and McInnes [12] discuss software lifecycle processes in the context of scientific software development. They note that, in contrast to most non-scientific software development where the goal is software creation, in scientific software development, software is the *means* of conducting research, not the *product* of it. Therefore they argue that conventional processes of software development are not well-suited to scientific software development.

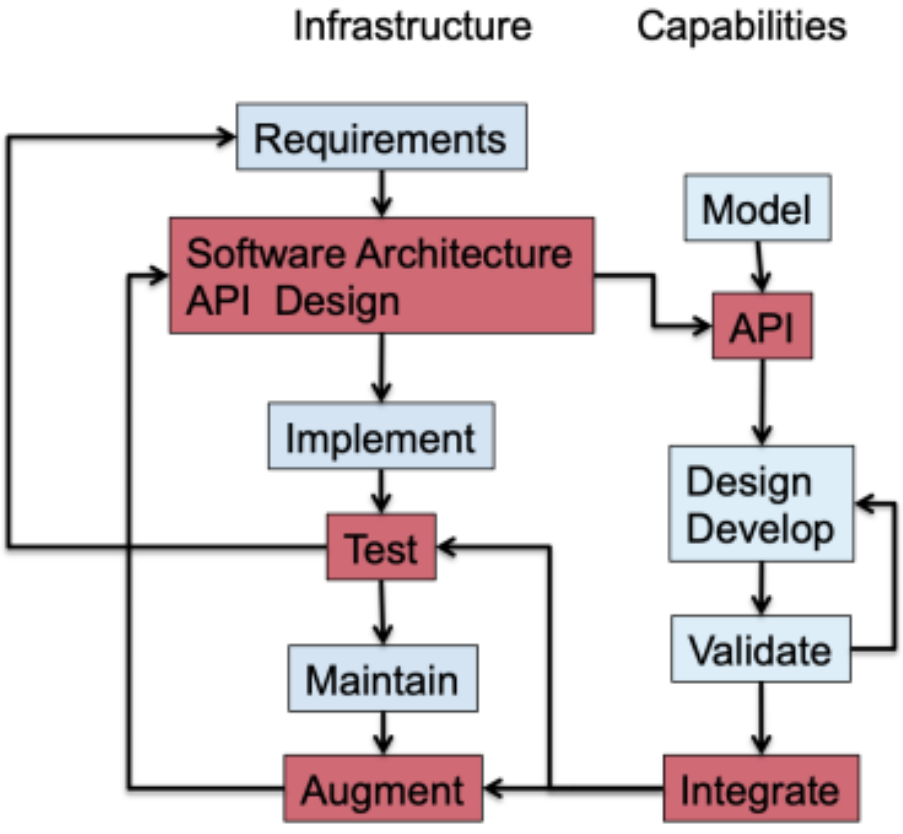
In [12], Dubey and McInnes describe a scientific software lifecycle, formalizing the approaches already used in various projects, such as SAMRAI [27], FLASH [28], Enzo [29] and Amanzi [30]. The key point is that scientific software can be thought of as consisting of two components: the *infrastructure* code and the *scientific capability* code. The infrastructure code handles tasks like discretizations and I/O. It is relatively stable, and it may be developed using any standard software lifecycle pattern. The scientific capability code describes the scientific model itself. Its requirements are research-driven, and the code itself is constantly evolving. Dubey and McInnes introduce a software development lifecycle for the scientific capability code, and describe how this should interface with the development process of the infrastructure code. Isolating the scientific code from the infrastructure code and limiting the points of contact between them allows rapid prototyping and development of scientific code, without undermining the stability or functionality of infrastructure code.

Scientific capability development process The development lifecycle proposed by Dubey and McInnes is shown schematically in Figure 12(a). The essential components are connected by solid lines. The process is this: first, requirements for the software are gathered, and the target phenomena are described using a mathematical model. Next, approximations are introduced as necessary to simplify the model or make it tractable. Then the equations are discretized, and appropriate numerical algorithms implemented to solve the model. The resulting software is then tested for correctness – *i.e.* that is solves the correct equations – and its stability and convergence properties are studied. This produces software of the required standard, but it does not ensure that the underlying mathematical model adequately describes the target phenomena. The code is therefore then validated against calibrated observations, with the mathematical model feeding into the calibration by defining regimes in which the model is valid. The validation phase then feeds back into the approximation, algorithm and (occasionally) the discretization phases, if it is necessary to make changes to ensure that the model describes the target phenomena. This feedback from simulations into planning is an integral part of the evolution of scientific code requirements capture.

The wide-dashed arrows from mathematical models and numerical algorithms indicate phases that may be skipped in certain cases. For example, calibration may not be needed in simple cases where a mathematical model is universally applicable. Similarly, some correctness, stability and



(a)



(b)

Figure 12: Lifecycle patterns proposed by Dubey and McInnes [12]: (a) the scientific software lifecycle pattern; and (b) the interaction of the infrastructure and scientific capability patterns. These graphics are respectively Figures 3 and 4 of ref [12], the latter also corresponds to Figure 1 of ref [1].

convergence checks may not be required if the software uses third-party libraries or externally-verified software suites.

Finally, the narrow-dashed lines indicate how the software development cycle interfaces with other concerns like simulation planning and scientific developments in the field. Planning, such as determining available compute resources can necessitate changes to the numerical implementation to make a set of simulations feasible. Analysis of simulation results can lead to an improved understanding of the target phenomena, which can result in modifications to the mathematical model and approximations used.

The above description treats a single mathematical model. In more complex multiscale or multi-physics systems, there will be multiple mathematical models, but the same software lifecycle may be used for each model independently.

Interfacing development processes Now we must consider how to connect the development process for scientific capability code described above with a standard development process used for infrastructure code. Ideally, there will be few points of contact between the two processes, as this will allow separation of concerns, and for the two software lifecycles to proceed at their differing rates largely independent of one another. Dubey and McInnes [12] propose the approach shown in Figure 12(b), where high-level descriptions of the infrastructure and scientific capability codes are given in the left- and right-hand columns respectively. There are only three coupling points between the two lifecycles. The primary coupling point is through the design of the API. A well-designed module structure will ensure that discrete modules communicate their data through a fairly stable API. However, improvements in scientific understanding might reveal necessary changes in the data dependencies between modules, that would be reflected in changes to the API. The other coupling points are where the scientific capability code interfaces with the infrastructure's testing framework, and where improving the scientific capability of the code (by perhaps adding a feature) means that the functionality of infrastructure code must also be extended. These coupling points are however very limited, and therefore the infrastructure and scientific capability code can be developed with largely independent lifecycles.

3 Summary

The role of software design patterns is probably best summed up by Barr [31, § 7], who argues that even the simplest ones such as *Singleton* are a useful start to creating a common language for documenting and communicating code details. The Go4/GoF *Strategy* pattern is singled out as making it much easier to extend software capabilities. The same pattern is also mentioned by Hewitt [32, § 7] as a way of producing more flexible and reusable software, and Rouson et al [6] show a scientific application to the choice of discrete time integration scheme. Hewitt [32, § 8] also describes use of the *Proxy* (particularly) and the *Facade* patterns as a very useful way of saving time when developing software, for much the same reasons given in Section 2.1.1, and uses *Builder* to help describe another (specification) concept. When it comes to scientific application however, Rouson et al take most Go4 design patterns only as an inspiration for developing their own *Abstract Calculus* and *Puppeteer* patterns.

It becomes clearer how design patterns fit into the landscape of software reuse [2, § 15], although it is perhaps more appropriate to think of a hierarchy rather than a landscape. At the top level of a given piece of software, there might be a framework which calls other code that in turns uses lower-level libraries which may be written specially for the project, from open-source repositories or provided by the machine vendor. The middle-level code is where design patterns come into play, and perhaps more as abstractions, ie. 'concept reuse' [2, 7.3.1]. This would seem to sum up best the use of the design pattern concept by Rouson et al [6].

In the context of reuse, Sommerville also mention components and component frameworks, ie. 'collections of objects and object classes that operate together to provide related functions and services', but these do not seem to have found much application in scientific programming [6, § 1.4]. ('Component' is also used to mean a stand-alone piece of software such as might form an element of a workflow, as in VECMAtk, and further as an addition to the C++ Standard Template Library.) It would of course be helpful to be able to reuse the work of the ComPat and VECMAtk projects, but the results obtained so far in application to fusion (Section 2.2.2 andSection 2.2.3) have been disappointing because of load-balancing issues, and it is not immediately evident how to remedy this situation within their general framework. The preferred way to proceed is perhaps to examine more problem-specific formulations using the NEPTUNE proxyapps.

Given the lack of design patterns for scientific work, arguably the most important practical aspect of patterns as far as NEPTUNE is concerned, is the concept reuse aspect when it extends to concepts like Resource acquisition is initialization (RAII). Such concepts are invoked as a means to avoid deficiencies in C++, which in the case of RAII, is a lack of ability to monitor memory leakage. Thread-safety is another issue, and as prefigured in Section 1 associated patterns have become part of languages like Rust [33], but which are sadly not in common use on HPC. However these 'safety-first' concepts need to feed into instructions as to how the main NEPTUNE software is coded, particularly when C++ is used.

Regarding prototyping, as might be expected on the basis of previous comments, little use has been made of design patterns in scientific applications and more generally, little further literature was found on how best to deal with the specific needs of prototyping scientific software beyond the work of Dubey and McInnes [12]. It is worth remarking that as Dubey and McInnes themselves explain, their document [12] is not meant to be definitive for scientific prototyping work, instead

more as an approach to be tested for its suitability by different groups for different projects. In the context of the NEPTUNE project, their approach could be used separately to develop each of the proxyapps. There is a higher level to NEPTUNE, however, whereby the experience gained by producing each proxyapp feeds into the design and development of subsequent ones. Thus there might be added a second loop on the right of Figure 12(b) indicating changes to the way the software is designed, developed, validated and integrated in the light of experience, and feeding into aspects of the model definition.

Acknowledgement

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

References

[1] L. Anton, W. Arter, and D. Samaddar. NEPTUNE: Report on system requirements. Technical Report CD/EXCALIBUR-FMS/0014-1.00-M3.1.1, UKAEA, 2020.

[2] I. Sommerville. *Software Engineering. 5th Edition (10th Edition, 2017)*. Addison-Wesley, 1997.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] Software Design Pattern wiki. `https://en.wikipedia.org/wiki/Software_design_pattern`, 2020. Accessed: July 2020.

[6] D. Rouson, J. Xia, and X. Xu. *Scientific software design: the object-oriented way*. Cambridge University Press, 2011.

[7] ComPat Project. Computing Patterns for High Performance Multiscale Computing ComPat website. `https://www.compatproject.eu/`, 2020. Accessed: July 2020.

[8] VECMA Project. Verified Exascale Computing for Multiscale Applications VECMA website. `http://www.vecma.eu`, 2020. Accessed: July 2020.

[9] VECMA Project. VECMA VVUQ Toolkit VECMAtk website. `http://www.vecma-toolkit.eu`, 2020. Accessed: July 2020.

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

[11] 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.

[12] 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.

[13] yEd Graph Editor. https://www.yworks.com/products/yed, 2020. Accessed: July 2020.

[14] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[15] H. Gardner and G. Manduchi. *Design Patterns for e-Science*. Springer, 2007.

[16] M. Haveraaen, K. Morris, D. Rouson, H. Radhakrishnan, and C. Carson. High-performance design patterns for modern Fortran. *Scientific Programming*, Article ID 942059, 14 pages, 2015.

[17] 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.

[18] Morfeus - Multi-physics Object-oriented Reconfigurable Fluid Environment for Unified Simulations. https://github.com/sourceryinstitute/MORFEUS-Source, 2020. Accessed: August 2020.

[19] ComPat Project. Software resources for ComPat website. https://www.compatproject.eu/software-resources/, 2020. Accessed: July 2020.

[20] O.O. Luk, O. Hoenen, A. Bottino, B.D. Scott, and D.P. Coster. Compat framework for multiscale simulations applied to fusion plasmas. *Computer Physics Communications*, 239:126–133, 2019.

[21] Poznan Supercomputing and Networking Center. QCG website. http://www.qoscosgrid.org/trac/qcg, 2020. Accessed: June 2020.

[22] S. Alowayyed, D. Groen, P.V. Coveney, and A.G. Hoekstra. Multiscale computing in the exascale era. *Journal of Computational Science*, 22:15 – 25, 2017.

[23] R.A. Richardson, D.W. Wright, W. Edeling, V. Jancauskas, J. Lakhlili, and P.V. Coveney. Easyvvuq: A library for verification, validation and uncertainty quantification in high performance computing. *Journal of Open Research Software*, 8(1), 2020.

[24] D.W. Wright, R.A. Richardson, W. Edeling, J. Lakhlili, R.C. Sinclair, V. Jancauskas, D. Suleimenova, B. Bosak, M. Kulczewski, T. Piontek, P. Kopta, I. Chirca, H. Arabnejad, O.O. Luk, O. Hoenen, J. Weglarz, D. Crommelin, D. Groen, and P.V. Coveney. Building confidence in simulation: Applications of EasyVVUQ. *Advanced Theory and Simulations*, page 1900246, 2020.

[25] J. Feinberg and H.P. Langtangen. Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal of Computational Science*, 11:46–57, 2015.

[26] Jonathan Feinberg. ChaosPy – Uncertainty Quantification Library. https://chaospy.readthedocs.io/en/master/index.html, 2020. Accessed: July 2020.

[27] Lawrence Livermore National Laboratory Center for Applied Scientific Computing. SAMRAI Structured Adaptive Mesh Refinement Application Infrastructure. https://computation.llnl.gov/casc/SAMRAI/. August 2020.

[28] A. Dubey, K. Antypas, M.K. Ganapathy, L.B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[29] G.L. Bryan, M.L. Norman, B.O'Shea, T. Abel, J.H. Wise, M.J. Turk, D.R. Reynolds, D.C. Collins, P. Wang, S.W. Skillman, B. Smith, R.P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N.J. Goldbaum, C.B. Hummels, A.G. Kritsuk, E.J. Tasker, S. Skory, C.M. Simpson, O. Hahn, J.S. Oishi, G. So, F. Zhao, R. Cen and Y. Li. ENZO: an adaptive mesh refinement code for astrophysics. *Astrophysical Journal Supplement Series*, 211(2):19, 2014.

[30] J.D. Moulton, S. Molins, J.N. Johnson, E. Coon, K. Lipnikov, M. Day, and E. Barker. Amanzi: An open-source multi-process simulator for environmental applications. In *AGU Fall Meeting Abstracts*, 2014.

[31] A. Barr. *The problem with software: Why smart engineers write bad code*. MIT Press, 2018.

[32] E. Hewitt. *Semantic Software Design: A New Theory and Practical Guide for Modern Architects*. O'Reilly Media, 2019.

[33] Rust programming language. https://www.rust-lang.org/, 2020. Accessed: August 2020.