Middle-out Design: A Best Practice for GEOSS Design

Jerry Overton[1]
[1] Computer Sciences Corporation, joverton@csc.com

Abstract

Because of its scale, the GEOSS design has the challenges that come with desigining Ultra-Large-Scale (ULS) systems. Traditional methods (top down and bottom up) are not adequate to handle these challenges. We describe the method of middle-out design: a suggested best practice for designing the GEOSS.

Keywords: best practice, ultra-large-scale, systems design, GEOSS, design pattern, problem view

1 Introduction

The proposed scale of the Global Earth Observation Systems of Systems (GEOSS) [1] suggests that its design will have all the challenges of designing Ultra-Large-Scale (ULS) systems: competing requirements, continuous evolution, heterogeneous parts, and normal failures [2]. Traditional design methods are not adequate to handle these challenges. Top-down design is unlikely to converge on a viable solution and bottom-up design is unlikely to result in a useful system. To successfully design at the scale proposed for the GEOSS, we need a best practice that will allow us to converge onto a viable solution to our problem while at the same time address our ULS-specific design issues.

We describe the method of middle-out design as a best practice for GEOSS design. We start by using the patterns to define a new view (the problem view) of software systems design that breaks complex GEOSS problems into simpler ones with known stable solutions. We then show how to solve GEOSS design problems by working from the middle of the problem out: from patterns to problems to solutions.

2 An Integrated GEOSS Architecture: An Ultra-Large-Scale Design Problem

Because of the GEOSS' proposed scale, the architecture task of the GEO 2009-2011 Work Plan fits the profile of a ULS design problem [2]. The GEOSS has the potential to be vastly greater in size than many software-intensive systems in several dimensions: lines of code; number of system users; amount of data stored, accessed, and processed; number of software component interdependencies. In the architecture, we see the characteristic challenges of designing ULS systems: resolving competing requirements, allowing for continuous evolution, accommodating heterogeneous parts, and protecting against normal failures [2]. Each of the following problems is a consequence of the scale of the GEOSS, and each of these problems contributes to the complexity of designing the GEOSS.

2.1 Resolving Competing Requirements

Satisfying GEOSS design means balancing several competing requirements. For example, the AR-09-02: Interoperable Systems for GEOSS task [1] requires an integrated network of services; while the GEOSS Common Infrastructure (task AR-09-01 [1]) requires those services to be composed of diverse component systems contributed by countries and organizations from around the world. The AR-09-01: GEOSS Common Infrastructure (GCI) task [1] requires that existing global telecommunications systems be extensible to new weather, water and climate data exchange services; but the AR-09-03: Advocating for Sustained Observing Systems task [1] requires that this

network of services remain reliable and stable. As more complexity is added to the GEOSS the more likely it is that competing requirements will be discovered.

2.2 Allowing for Continuous Evolution

The GEOSS will be in service for a long time and its eventual size will make it impractical to replace or retire the entire system. The system has to progress by a process of evolution where the system changes continuously, not just in discrete phases. The system needs a process of change that allows the local needs of the subsystem designers to be satisfied without destroying the value of the overall system. With this kind of evolution, different subgroups of stakeholders should be free to choose local solutions that fit their own needs, but be constrained so that they contribute to (or at least not interfere with) the value of the overall system.

2.3 Accommodating Heterogeneous Parts

The component systems of the GEOSS will come from the contributions of various GEOSS members and participating organizations [3]. These systems will have their own architecture, hardware platform, software platform, etc. Already, there exists a list of diverse systems proposed as components for the GEOSS (see Annex 1 of [3]). The GEOSS design will have to account for the integration of heterogeneous parts contributed by various organizations.

2.4 Protecting Against Normal Failures

If the transfer protocol of any of the GEOSS dissemination and distribution networks (GEONET, GEONETcast, etc [1]) fails only once in a million uses, but is used millions of times each day (which seems likely because of its proposed scale), a transfer failure will occur, on average, once a day. The same estimations hold true for the application services running within these networks. We can expect that in the GEOSS, even with high-quality components and infrastructure, something will always be failing. We need a design method capable of designing a sytem that stays running even when its components are constantly failing.

3 Problems with Traditional Design Methods

The 4+1 model view of software is an industry standard method for designing software-intensive systems [4]. The method breaks the system into multiple, concurrent views (logical view, development view, process view, physical view, and scenarios). Each view describes the system from the viewpoint of stakeholders (like end-users, developers, and project managers) that share a common set of concerns. In particular, the logical view describes the component details needed to achieve the functionality of the system; while scenarios describe, in story form, the problems that the system is designed to solve. The details of the other views are not important to this paper.

The 4+1 model gives us the options of designing systems by working from either the top down – starting from scenarios and ending with a detailed logical view – or by working from the bottom up – starting from a logical view and ending with detailed scenarios. But neither of these methods of systems design is adequate to handle the ULS design problems described in the section 2 (see Figure 1).

Take, for example, the problem of designing the GEOSS Common Infrastructure (GCI) [1]. Working the problem from the top down, we can start with the problem of integrating contributed components into a functioning, interoperable network. From here, we could easily decompose the problem into subproblems with no real viable solution. Say we decompose the GCI problem into, among others, the subtask of building a core component registry [1]. The registry's structure has to be stable enough to support the continuous operation of the GEOSS and flexible enough to accept components designed for things we can not anticipate now. But our scenarios do not tell us whether or not current registry technology can actually do that. The registry has to support some kind of failover for when a component fails, but our scenarios do not tell us how much fault-tolerance support we can expect from current registry technology and whether or not that will be adequate to solve our problem.

We will likely not fare any better working the problem from the bottom up. Say we start solving the problem of building an architecture pilot [1] by starting with specific mechanisms like Web service-based components registered in a service-oriented architecture (SOA) registry-repository. Are SOA registry-repositories capable of integrating

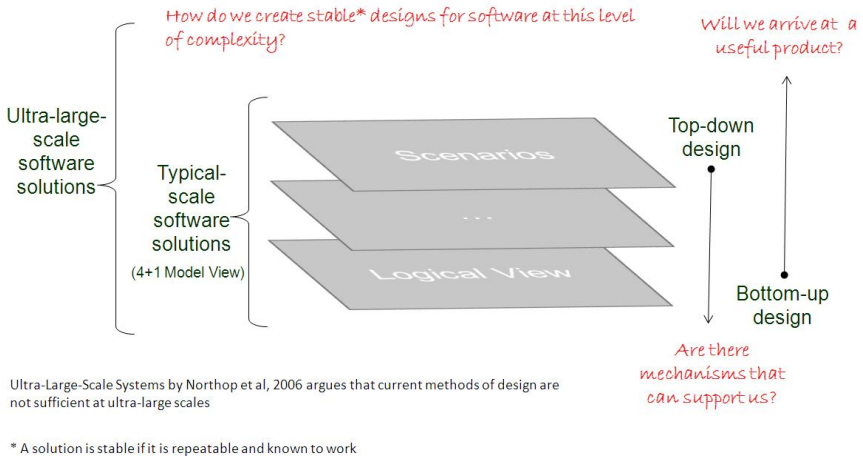

Figure 1. The Problem with Traditional Software Design Methods

with heterogeneous system development environments well enough to provide useful technical component information on the scale proposed for the GEOSS? We will not know for sure until much later in the life of the GEOSS project if making a design correction (if necessary) will be very expensive, or simply not feasible.

To successfully design at the scale proposed for the GEOSS, we need a new method for system design. We need a best practice that will allow us to converge onto a viable solution to our problem while at the same time address our ULS-specific design issues: resolving competing requirements, allowing for continuous evolution, accommodating heterogeneous parts, and protecting against normal failures.

4 The Problem View: A New View of Software Systems Design

We can solve the problem of converging onto a viable solution by adding a new view, the problem view, to the standard 4+1 model. Figure 2 shows a problem view of the design for the GEOSS common infrastructure problem. The notation of the problem view comes from [5].

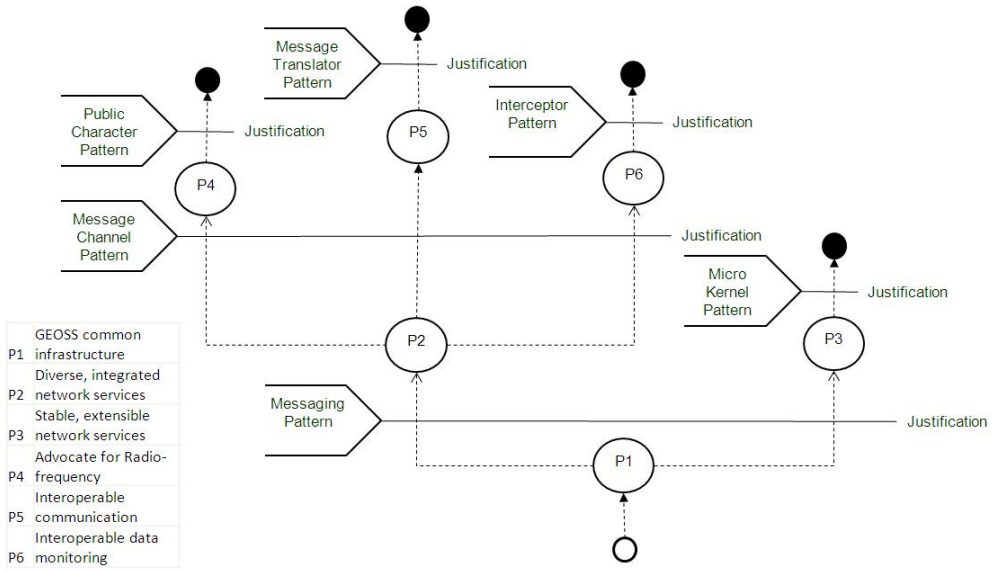

Figure 2. Problem View for the GCI

The open, labeled circles represent problems. The table describes the problems represented by each lablel: P1 represents the GEOSS common infrastructure problem; P2 represents the diverse, integrated network services problem; and so on. The arrows represent a transition from one problem to another. Every problem transition is guarded by a gate that describes the design pattern used to guide the transition from one problem to the next. Using the messaging pattern, the GEOSS common infrastructure problem (P1) is transformed into the problems of building diverse, integrated network services (P2) and the problems of building stable, extensible network services (P3). A problem is solved when all of its subproblems are solved. P1 is solved when P2 and P3 are solved. P2 is solved when P4, P5, and P6 are solved. The transition into a black circle indicates that a problem has been solved. In our problem view, P3, P4, P5, and P6 are all solved problems.

Figure 3 shows the relationship between the problem view and the other views of the 4+1 model.

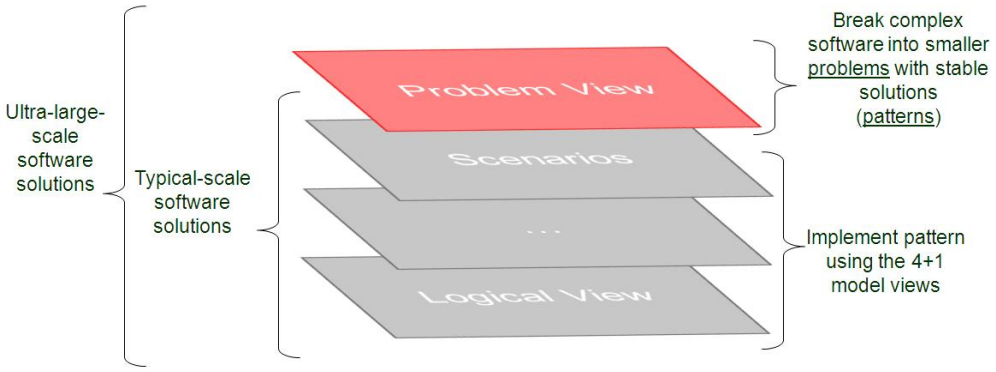

Figure 3. Adding a Problem View to the 4+1 Model

The problem view is concerned with reducing complex problems to simpler ones or ones that we have seen before. It allows a designer to take complex problems and, guided by design patterns, break them into smaller problems with known stable solutions. A design pattern is a solution to a common problem in software systems design and, because the breakdown of problems is guided by the application of design patterns, using the problem view makes us reasonably certain that the resulting problems will have viable solutions.

5 Working Middle Out: A Best Practice for GEOSS Design

We can solve our remaining design problems (resolving competing requirements, allowing for continuous evolution, accommodating heterogeneous parts, and protecting against normal failures) by working from the middle out. Figure 4 shows the relationship between top-down, bottom-up, and middle-out design (in the figure, examples are listed next to each concept). Working middle out, we start with an appropriate design pattern, describe our problem in terms of that pattern, and then craft a solution according to the requirements of the pattern.

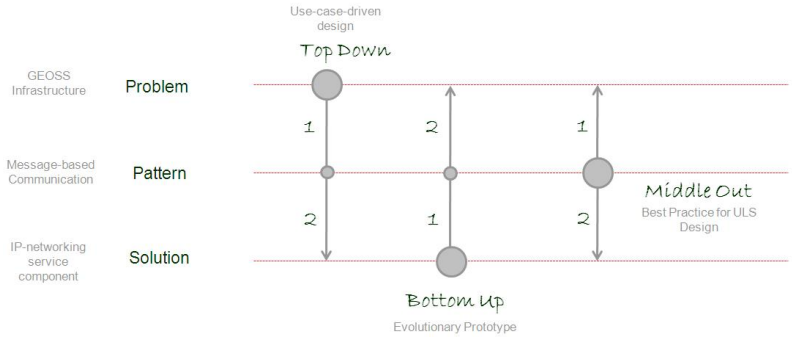

Figure 4. Top-down, Bottom-up, and Middle-out design

Figure 5 shows an example application of middle out to the GEOSS design. Working out from the root node of the problem view (see Figure 2), our knowledge of design patterns allows us to describe the GCI problem in terms of the messaging pattern [6]. That is, the GCI problem is described as the problem of creating a message bus that will allow GEOSS services to communicate without being tightly coupled. Solving this problem forms the first development line of our solution.

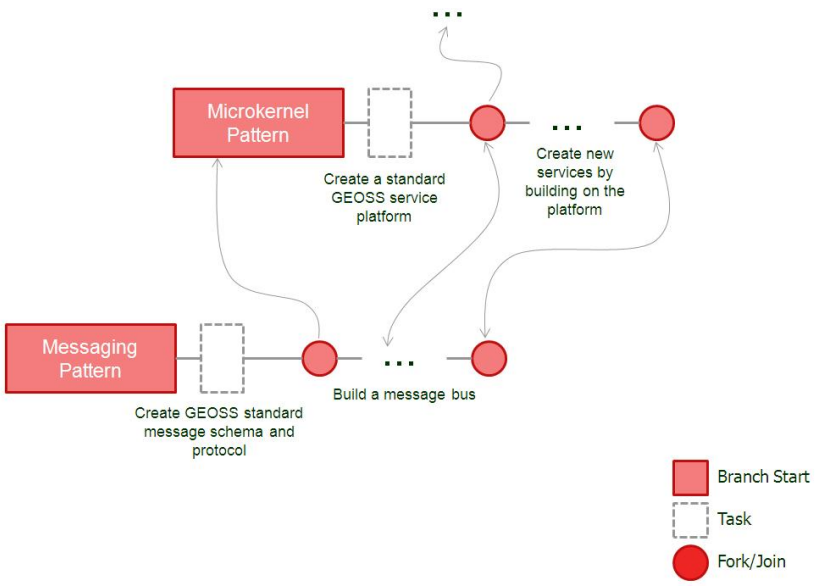

Figure 5. An Example Application of Middle-out Design

With our first task within the Messaging Pattern line, we create a standard messaging schema and protocol that all GEOSS services will follow. Next, we branch off a new line (Microkernel Pattern [6]) dedicated to defining a standard kernel for all GEOSS Services. The kernel will be built around the standard message schema and protocol defined in the Messaging Pattern line. Back in the Message Pattern line, the next task is to define the message bus. As indicated by the synchronization tasks from the Microkernel Pattern line to the Messaging Pattern line, the details of the system's message bus will evolve as the details of the microkernel defined.

From the Messaging Pattern line, refinement of the solution continues recursively up the tree defined in the problem view until all the problems are solved. The structure of the system's branch lines mirrors the structure of the problem view. The results from every child branch line are continuously integrated into the results of its parent line.

Working from the middle out solves our remaining ULS design problems. For the problem of competing requirements, the Messaging Pattern line balances the requirements of a diverse, yet integrated, network; the Microkernel Pattern line balances the requirements of stable, yet extensible network services. For the problem of continuous integration, the Messaging Pattern line is structured to allow continuous adaptation to the results of the Microkernel Pattern line. For the problem of heterogeneous parts, the standard message schema from the Messaging Pattern line allows integration from heterogeneous services. Finally, for the problem of normal failures, the message protocol from the Messaging Pattern line allows for network operation in the face of service failures.

6 Benefits of Working from the Middle Out

As long as we can find appropriate design patterns, we can work on our problem from the middle out. We start by using the patterns to define a problem view that breaks our complex problem into simpler problems with known stable solutions. We work middle out by defining a development line for every pattern in the problem view. Within each line, we first establish the framework of the pattern using an infrastructure task. After the infrastructure of a line is established, the details are handled by recursive branching. The results of branched lines are continuously integrated into the branches' base lines.

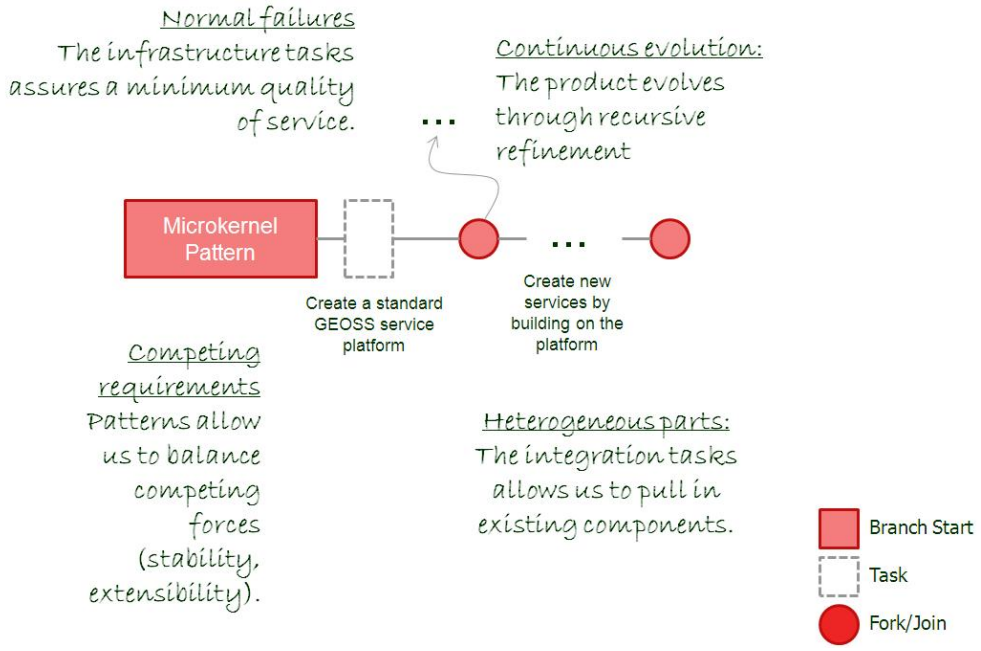

Figure 6. Benefits of Working Middle-out

Using patterns (whose primary purpose is to balance opposing forces [6]) to define the structure of the problem view and development lines allows us to define a design that balances competing requirements. The integration from branch development lines to base development lines allows for a continuous system evolution. The infrastructure task establishes a framework for integrating heterogeneous parts from branch lines. The infrastructure task also establishes standards for handling the failures of parts integrated from branch lines.

7 Acknowledgements

We would like to thank our colleagues in the Computing Department at The Open University for their continuing support; especially Jon G. Hall, Lucia Rapanotti and Yijun Yu, for their insightful reading and comments on this paper.

References

[1] GEO 2009-2011 work plan, December 2009.

[2] L. Northrop, P. H. Feiler, B. Pollak, and D. Pipitone. *Ultra-large-scale Systems: The Software Challenge of the Future.* Software Engineering Institute, Carnegie Mellon University, 2006.

[3] The global earth observation system of systems (GEOSS) 10-Year implementation plan reference document, February 2005.

[4] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[5] J. Hall and L. Rapanotti. Assurance-Driven design in problem oriented engineering. *International Journal on Advances in Systems and Measurements*, 2(1):119–130, 2009.

[6] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*, volume 4. John Wiley & Sons, 2007.