

Layered Architecture Revised

Asher Sterkin

NDS Technologies Israel Ltd.,

P.O. Box 23012, Har Hotzvim, Jerusalem, Israel

Abstract - A systematic approach to building layered software architecture is proposed. Separate layering models required for a proper abstracting of hardware, persistent data, and communication protocols are described. Implications on various types of testing are discussed.

Keywords: embedded systems, architecture, layers, design, UML

1 Introduction

The idea of a layered architecture was originally formulated by E.W. Dijkstra in his seminal “The Structure of the ‘THE’-multiprogramming system” paper [1]. This approach was later generalized and codified in a form of “Layers” architectural pattern [2].

After almost 40 years since its initial inception many developers, especially in the embedded software applications area, still tend to compromise or avoid it completely due to a perceived potential overhead incurred. In fact the opposite is true, for embedded systems in general, and for consumer electronic devices in particular, due to limited resources and extremely high demand for reliability and usability incorrect layering can be a fatal mistake.

In this paper a systematic approach to building layered software architecture, which does not incur any uncontrollable overhead, is proposed. Separate layering models required for a proper abstracting of hardware, persistent data, and communication protocols are described. Implications on various types of testing are discussed.

2 “4+1” View of Architecture

In [3] Philippe Krutchen introduced a concept of “describing the architecture of software-intensive systems, based on the use of multiple, concurrent views. According to the “4+1” View approach we distinguish between: logical view (object model of the design), process view (concurrency and synchronization), physical or deployment view (mapping software on hardware) and development view (organization in development environment). Every view can have its own layering, which is quite different from those of other views. In this chapter we will briefly review possible layering for deployment, process and development views. We will deal with logical view layers in more detail in the next chapter.

2.1 Deployment View Layers (Tiers)

There is a lot of confusion about in what if any tiers are different from layers. For example, the “Layering Strategies” [9] white paper claims that “the fact is that a tier *is* a layer, albeit a layer based upon particular strategy – one of responsibility.” Still, more traditional definitions associate the concept of tiers with client-server distribution pattern. In this context by tier we understand a group of computers with the same workload, scalability and redundancy patterns, and security profile. Also there two possible sub-versions of this architecture: thin- and rich- (or thick, fat) client.

2.2 Development View Layers

This type of layering is dealing primarily with ownership and business issues rather than with the system functionality reflected in logical view layers. Business considerations and contractual relationships between multiple vendors and final customers are playing a crucial role in establishing coarse grain development layers. System build and integration process could introduce secondary layers.

2.3 Process View Layers

This is probably the least investigated area, which to the best knowledge of this author has a very little coverage in literature. Layers in a process view could be identified and properly modeled. For instance in a modern computer, one could distinguish between hyper threads running within a scope of the same core, threads running on different cores of the same

CPU, threads or tasks running on different CPUs of the same computer, tasks running on different nodes of the same High Performance Computing (HPC) cluster, tasks running on different computers (clustered or not) within the same local network, tasks running on different remote computers (e.g. Intranet or Internet). Dealing with this kind of distribution, or as it's sometimes called application partitioning, models is obviously related to the deployment patterns discussed above, but are still quite different. Here we are mostly concerned with a question of which level of parallelism and by which means could be achieved in the system. Together with deployment patterns the process view layering has a strong business impact in terms of purchased equipment, operational cost and a guaranteed level of quality of service.

3 Logical Layers

Within the logical view we can provide a concept of layers with a more precise meaning and to utilize its full power of abstraction for dealing with the software system complexity.

Within the logical view scope by a *layer* we will understand a UML class package [5] stereotyped as <<layer>>. The most important part in this definition is a restriction on relationships between layers: it could be only "depends on" relationship between a higher layer and a lower layer. When a strict layering model is applied any layer can be dependent only on a layer, which is right underneath. When we say "depends on" we actually mean interface dependency, namely when a layer "A" depends on a layer "B" it means that "A" either *uses* or *implements* one or more *interfaces*, defined by "B". There is no any assumption about control or data flow between the two layers, which will typically flow in the both directions.

In the case of embedded applications including consumer electronic devices developers many times need to deal with system, communication and application software simultaneously. Each one of these "dimensions" can be modeled using its own layering structure. When dealing with the hardware *dimension* the following layers are typically used: hardware, drivers, operating system, middleware, framework, application. When dealing with the communication *dimension* the following layers are typically used: physical, data link, network, transport, session, presentation, application.

When dealing with persistent data *dimension* the following layers are typically used: data, data access, domain logic, application logic, presentation. To ensure testability the presentation layer has to be as *thin* as possible and not to contain any logic at all.

To reduce the overall complexity the application logic layer is further subdivided into three loosely coupled packages: model, view, controller. To reduce the overall complexity even further the whole application can be modeled as a set of loosely coupled subsystems, each having its own presentation and application logic layers, and being built on the top of the same domain logic layer. The loosely coupled subsystems can be also used for the domain logic layer itself in order to reduce its complexity and improve modularity and testability.

The main benefit for the testability is that each subsystem and each layer can be tested completely independently.

4 Summary

When applied properly the "Layers" architectural pattern is a powerful modeling mechanism, which could significantly increase architectural control over even very sophisticated software system. One still has to pay a caution and not to overcomplicate the model unjustifiably. The basic rule of thumb is: "do not model those system aspects, which are given for free." For instance, if you are lucky to develop your software using a well-defined framework on the top of a feature-rich middleware with standard and well-specified communication protocols, you probably do not need to bother about modeling these aspects of your software, and to devote the whole energy to building good Domain Model and Application Logic layers. Ideally this should be the case, and hopefully it sometime will be. Unfortunately, nowadays, especially in the embedded software world, system and communication aspects could not be completely ignored and thus need to be modeled accurately. Separation of concerns should be the main guiding rule here: when you model a system or communication, ignore application; when you model domain and/or application, ignore the system and communication. We also need constantly keep in the mind that whatever models we use, they are just models intended to help us with dealing with a system, which complexity is beyond our mental capabilities.

5 References

- [1] Dijkstra, E.W.: "The structure of the 'THE'-multiprogramming system", Commun. ACM 11 (1968)
- [2] Buschman, F., et al: "Pattern-Oriented Software Architecture: A System of Patterns," John Wiley & Sons, 1996
- [3] Krutchen, P.: "Architectural Blueprints: The "4+1" View Model of Software Architecture", IEEE Software 12(6), 1995
- [4] P. Eeles, "Layering Strategies", Rational Software White Paper, TP 199
- [5] M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language", Addison-Wesley, 2003