Logical Layering Heuristic

WIEBE WIERSEMA, HU University of Applied Sciences, Utrecht, the Netherlands
LEO PRUIJT, HU University of Applied Sciences, Utrecht, the Netherlands

There have been a large amount of publications on the subject of logical layering in software architecture. What is rarely discussed is the manner how a software architect can design a layering model that satisfies the requirements of the system.

In this work we present a heuristic that defines the steps that need to be taken by a software architect to design a software architecture layering model that satisfies the given requirements: LOGICAL LAYERING HEURISTIC.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*(Design) Patterns*

General Terms: Software Architecture, Layers, Layering

Additional Key Words and Phrases: Software Architecture, Layers, Logical Layers, Layering

ACM Reference Format:

Wiersema, W., Pruijt, L. 2015. Logical Layering Heuristic HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. (October 2015), 11 pages.

Logical Layering Heuristic

The Layers pattern, or Layered style, is one of the most common patterns used in software architecture [Clements et al. 2002] [Harrison and Avgeriou 2008]. The concepts of layering can be traced back to works of [Dijkstra 1968] and [Parnas 1972]. [Buschmann et al. 1996] described the Layers pattern extensively and [Avgeriou and Zdun 2005] show that layers are also described as patterns or styles by many other authors.

A layered architecture offers significant benefits. [Evans 2004] clearly explains in the "layered architecture" pattern the reasons why a layered architecture is important: Using layers allows for a much cleaner design. Software within that layer will address fewer concern compared to a non-layered architecture. Layering lowers maintenance costs as each layer can evolve as demanded by the needs of that layer. Layers can be positioned on different hardware tiers allowing more options to optimize for performance, bandwidth and resource usage. Additionally having good layers improves test-ability and enables lower regression testing costs for future updates.

As shown above a lot of literature mentions and describes layers, but there is actually very little material that comprehensively explains the steps an architect needs to take to design a layering model that meets the requirements of a system. Notable exceptions are [Buschmann et al. 1996], [Larman 2012] and [Hill 2009]. These publications describe a part of the steps and design considerations that are needed to reach a logical layering model.

Authors' addresses: wiebe.wiersema@hu.nl; leo.pruijt@hu.nl

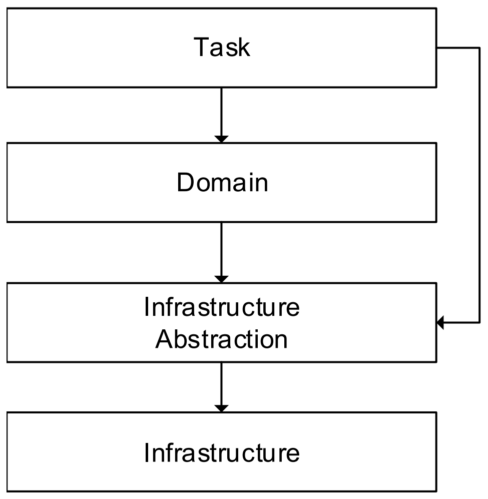


Fig. 1. Example of a layering model, using the terms from [Pruijt et al. 2013].

In this paper we will combine and enhance the knowledge of the aforementioned publications and other sources in order to provide a comprehensive heuristic to design a logical layering model. Logical means here "as free of implementation choices as possible". The layered style is a modular style [Clements et al. 2002] and does not focus on the runtime behavior, or the allocation of software components.

This paper is aimed at students, designers and software architects who struggle in finding the right layering for the software they design. This pattern will especially help to consistently and methodically design a layering model. Use LOGICAL LAYERING HEURISTIC when you are starting to learn how to design layering models and you want them the transferable to the team that will design, build and test the application. It often happens that software engineering projects do not pay enough attention to Non Functional Requirements (NFR), this pattern helps to clarify which NFR's are in scope and what measures need to be taken to be comply with the NFR's. Also the result of this heuristic will help testers to specify a testplan for Non Functional Testing.

Problems and forces around layering

Finding a suitable layering for an application or a component is hard. It is a serious challenge for the less experienced software architects to correctly design a logical layering for an application. These architects will skips steps and not cover the requirements in the design, leading to an an application that does not have a correct logical layering. The application will most likely not meet non-functional requirements such as usability, maintainability and scalability. Not meeting these requirements will lead to dissatisfaction from both end-users and business owners and might jeopardize acceptance of the application.

—Learning curve of Software Architecture.
Software engineers who want to learn software architecture can be overwhelmed by the number of subjects they need to take into consideration to design a layering model and will skip important steps.

—Time pressure.
Application development often happens under a lot of time pressure so the layering model needs to be correctly designed in a short time. Inexperienced architects may skip the necessary steps to meet their deadline.

—Choosing the right number of layers
Some architects tend to have too many layers in their design which might lead to performance problems. Other architects use not enough layers and deliver applications that may have maintainability and portability issues.

—Just enough architecting.
Agile software engineering still needs software architecture, but it needs it fast. How to get a good design that is good enough to start working on the software?

—Not something you do every day.
Architects do not design layering models very often. They only need to do this with new applications or when heavily refactoring an application.

—Systems that disappoint when delivered.
It often happens that systems do not have enough performance, can't be adapted quickly enough or are very expensive to maintain.

All these forces make life hard for software architects that are learning the trade and also threaten the success of the project if the application delivered falls short in the eye of the customer.

Heuristic

To help with the aforementioned problems this paper describes a repeatable heuristic to get the appropriate layering for the architecture of an application. Use the following steps to determine the layering. These steps are described in a sequential manner but also can be used in an iterative manner. The steps support answering the central question when applying the layer pattern: "How to divide the functional and non-functional requirements of the system across the layers so the system will satisfy these requirements?". This heuristic will not guarantee that your layering is the best, but it will take you through the correct steps to design a layering model that fits the customer needs and is transferable to the team that needs to implement the software according to the designed layering. It covers subjects as layer responsibilities, traceability with requirements and documenting the layering model.

Step 1. **Determine the quality goals**
The main question for this step is: "why do we need a layering and when are we done layering?"
The answer to this question exists out a set of quality goals, that should be satisfied by the layering. Traditionally design and analysis of an application focuses on the functional aspects. Functional requirements in general will not help to derive the right layering. To get right level of layering non-functional requirements (NFR) are the main driver. Use an industry NFR model such as the ISO25010 to have a checklist for the NFR you need to consider. NFR of specific interest are: efficiency, operability, reliability, security and maintainability [ISO 2011].

Step 2. **Determine the logical layers**
The main question for this step is: "What functionality needs to be divided across separate layers?"

Use the TYPOLOGY OF SOFTWARE LAYER RESPONSIBILITY pattern [Pruijt et al. 2013] (TSLR) to get a comprehensive overview of the functionalities and responsabilities that you need to decide about. The TSLR pattern distinguishes and describes different types of logic and provides concrete examples of functionality. See Appendix A for a summary of the TSLR.

With the separation of responsibilities from the TSLR the functionalities can be divided in various ways across the layers based upon the NFR. Use the following substeps to do this:

This step is divided in two substeps: 2.a to determine the quality goals, 2.b to define the layers and their ordening.

Substep 2.a. **Determine which layers are needed to satisfy the quality goals**
This substep uses the selected quality goals from Step 1 as input. Determine for each quality whether a separation in layers contributes to the realization of this goal. Also consider which functionalities and responsibilities need to be separated.

Examples of quality goals and how they can be realized by separating logic:

Quality Goal	**Which functionalities to separate**
'reusability' Enable the reuse of tasks logic so different presentation technologies can use the same logic 'maintainability - analysability and modifiability' is improved by separating task specific logic from presentation logic	Separate presentation logic from task logic.
'security' is improved by re-using the same security layer for all security services	Separate domain specific logic from infrastructure abstraction logic.
'portability - replaceability' is improved by separating DBMS, application server and peripherals from each other	Allow only the use of infrastructure abstraction logic by higher layers to hide specific infrastructure implementation details.
'maintainability' is improved by separating layers that have a different pace of change over time. This called shearing layers in [Foote and Yoder 1997]	Separating process related logic (that can quickly change over time) from more stable business logic that implements parts of a Federal law will improve maintainability.

Substep 2.b. **Define the layers and order the layers**
use the results from substep 2.a to define the layers and decide on their hierarchical level. Also define per layer what functionality is part of the layer and what not. Use the following design rules:

—The less layers the better!
Besided their benefits layers also have distinct disadvantages: Designers and developers need to invest time to understand and use each extra layer. Also extra layers may add performance problems [Larman 2012].

—Every layer has to respect the standard communication rules
A lower layer is not allowed to call an higher layer. If two elements can communicate freely, these parts are not layers but most likely two elements within the same layer.

—A layering model is a hierarchical vertical ordering.
Within a layer it is allowed to place several functionalities. But structures that are placed horizontally besides the vertical layering dont belong in this model. Often these structures are different subsystems, components or aspects.

—Separate subsystems can have different layering.
A backend server offering webservices will have different layering compared to the client app that uses the web services.

Step 3. **Determine the communication rules between the layers**
The main question for this step is: "What communication rules are applicable and what exceptions to the rules are allowed?"

This step is divided in two substeps: 3.a to determine which rules apply and why, 3.b to determine exceptions to the rules and why.

The common rules for layering as described in [Buschmann et al. 1996] are:

—Calls are only allowed from a higher layer to a lower layer. Communication the other way is not allowed.

—Layers are not allowed to skip a layer while communicating (strict layered)

We strongly advise to take these rules very seriously! Not adhering to these will allow free (and unstructured) communication between layers and will make it impossible to reach a number of quality goals.

A nice industry example where this went wrong was in the early versions of the .Net MVC Framework from Microsoft [Microsoft 2015]. The Model View Controller pattern [Buschmann et al. 1996] clearly separates presentation logic from domain logic. In the examples provided by Microsoft some views contained SQL statements, clearly breaking the rules where the view should be provided the database data by the controller through a model.

Substep 3.a. **Determine which rules apply and their rationale**

For each of the quality goals from Step 1 and the layers from Step 2 decide which communication rules should be followed. Also create a clear rationale why a rule is necessary.

Substep 3.b. **Determine which exceptions to the rules apply and their rationale**

Architecture is an exercise in compromise. Certain quality goals will be in conflict with other quality goals. If certain rules have a strong negative impact on other quality goals it may be necessary to provide an exception for that in specific cases. If exceptions are needed it should be very clear when and why these exceptions apply and what mitigations are needed to minimize the negative consequences of applying the exception to the rule.

Examples of quality goals and how they can be realized by communication rules:

Quality Goal	**Communication rule**
'reusability' Enable the reuse of tasks logic so different presentation technologies can use the same logic	Only allow layers to call an adjacent lower layer, skipping layers is not allowed. (Strict Layered Model).
'portability - replaceability' is improved by hiding implementation behind a facade at a lower level layer	Do not skip layers.
'efficiency - resource behavior' of searching in the database is improved for an object oriented application by allowing the serach function to bypass the domain model and to call optimized stored procedures that quickly deliver results.	Skipping the domain model layer is allowed when performing searches that can be best be handled by the database and that only retrieve read-only data.

Step 4. **Verify if the layering model is fit for use**

The main questions for this step are:

Q1. "Can the software really work using this layering model?"

Q2. "Are the quality goals adequately covered by this layering model?"

Q1 can be answered carefully selecting (parts of) the requirements that are architecturally significant. i.e. the requirements that either have led a architectural decision or can be used to mitigate a risk of not meeting key requirements. This can be done by making a design, as done by Larman [Larman 2012]. Larman assign the classes involved in a use case to a layer and verifies of it all fits or if there are places where there is friction. Additionally he uses a sequence diagram to visualize how the use case is realized and if any communication rules are broken. Another approach to answer Q1 is by building a prototype to verify to operation of the model in practice.

Answering Q2 is harder to do. Along the lines of the ATAM method [Clements et al. 2002] we need to take the quality goals from Step 1 and for each of these goals we need to create and simulate a scenario. These simulations need to show that the layering model (or the architecture) satisfies the quality goals. Examples of such scenarios are:

—Quality goals around reusability and maintainability could be tested by adding an extra use-case and assessing the impact of such an addition.

—Change an existing functionality that occurs in several use-cases. Are the changes isolated in one layer? Do we have the desired level of maintainability or adaptability?

—Change the database vendor, the operating system or the chosen security solution. Is it portable enough?

Step 5. **Document the layering model**

The main question for this step is: "How to capture the decisions, rationale and layering so the design can be understood now and in the future?".

The layering needs to be documented in such a way that it can be discussed, understood and verified by the current and future development and maintenance teams. This helps to implement and maintain the software in accordance with the layering model.

—Document the layers and the rationale why you think these layers are necessary and assign descriptive names to the layer

—Document which communication rules are obligatory and why. If necessary describe the rules per layer. Also document which exceptions there are and when it is valid to use these exceptions, and what mitigation's are necessary to minimize the negative impact of using the exception.

—Visualize the main overview of the layering model.

—Document which quality goals are realized by the layering and how the goals need to be satisfied.

While documenting keep the seven fundamental principles from [Clements et al. 2002] in mind that apply to all technical documentation, but especially to architecture documentation: (1) Write from the point of view of the reader; (2) Avoid unnecessary repetition; (3) Avoid ambiguity; (4) Use a standard organization; (5) Record rationale; (6) Keep documentation current but not too current; and (7) Review documentation for fitness of purpose.

Now you have reached the end of the steps and should have a good overview of the layers and the reasons they are in place.

Discussion

This pattern has been applied as part of the software architecture education at the HU university of Applied Sciences in Utrecht. We have also applied this pattern within a government agency where software architecture was still in its early stages of adoption. We have found it gives good guidance both students and junior software architects to learn the ropes on layering.

Industry example of this pattern are the Rational Unified Process Software Architecture Document [IBM 2015] and the OpenUP Architecture Notebook [OpenUP 2015]. Both these documents have requirements and constraints as an early chapter and have a separate chapter on layering. The steps in this pattern completely align with the tasks that you need to do to be able to provide content for the layering paragraphs of these documents.

This pattern is less applicable when you have to implement software within packages such as SAP and Oracle Siebel which prescribe a layering architecture that has to be followed in order to get working software.

Please note that while using this heuristic the following mistakes are often made.

—Following the layering model that was given by the supplier or an author and not starting with the quality goals that you need to satisfy.

—Only defining layers in the model and not describing any communication rules. Another variant is striking a lot of the rules. This will strongly reduce the advantages of the layering model.

—Putting subsystems or cross-cutting functionality as layers in the model. Examples of these are: security, workflow or reporting as top-layers above the User Interface layer.

—Positioning functionality that can be called from several layers i a column adjacent to the layering model. A layering model is a hierarchical model, this sort of functionality should be placed in a lower layer with an exception that it is allowed to skip layers to access this functionality.

—Not documenting the layering model and failing to effectively communicate this with the team. This will lead to interpretation errors or even people being unaware from the design constraints and making designs and code that ere detrimental to the quality goals.

We have found this pattern to be useful to the intended target audience both in education and industry.

Future research

One very interesting venue of research is on the area of defining a pattern language on software layering. This pattern language could be useful to provide an overview on the patterns that can be used for layering the software. Candidates for such patterns are structural patterns such as MVC, but also patterns that describe a specific layer such as a Services layer.

Acknowledgements

We wish to express many thanks to our shepherd Philipp Bachmann for all his highly useful comments and feedback. Furthermore the feedback given by the participants of the writers' workshop at the PLoP'15 conference helped a lot to improve this work.

REFERENCES

AVGERIOU, P. AND ZDUN, U. 2005. Architectural patterns revisited - a pattern language.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester.

CLEMENTS, P., GARLAN, D., BASS, L., STAFFORD, J., NORD, R., IVERS, J., AND LITTLE, R. 2002. *Documenting software architectures: views and beyond*. Pearson Education.

CLEMENTS, P., KAZMAN, R., AND KLEIN, M. 2002. Evaluating software architectures: methods and case studies. *Addison-Wesley Professional*.

DIJKSTRA, E. W. 1968. The structure of the "the"-multiprogramming system. *Communications of the ACM 11,* 5, 341–346.

EVANS, E. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

FOOTE, B. AND YODER, J. 1997. Big ball of mud. *Pattern languages of program design 4*, 654–692.

HARRISON, N. AND AVGERIOU, P. 2008. Analysis of architecture pattern usage in legacy system architecture documentation. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*. IEEE, 147–156.

HILL, D. 2009. Microsoft application architecture guide, patterns and practices. `http://msdn.microsoft.com/en-us/library/ff650706.aspx`.

IBM. 2015. Rational unified process. `http://www.ibm.com/software/rational`.

ISO. 2011. Iec 25010: 2011: Systems and software engineeringâĂŞsystems and software quality requirements and evaluation (square) system and software quality models. *International Organization for Standardization*.

LARMAN, C. 2012. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e*. Pearson Education India.

MICROSOFT. 2015. `http://www.asp.net/mvc`.

OPENUP. 2015. Openup the eclipse foundation. `http://www.eclipse.org/epf/`.

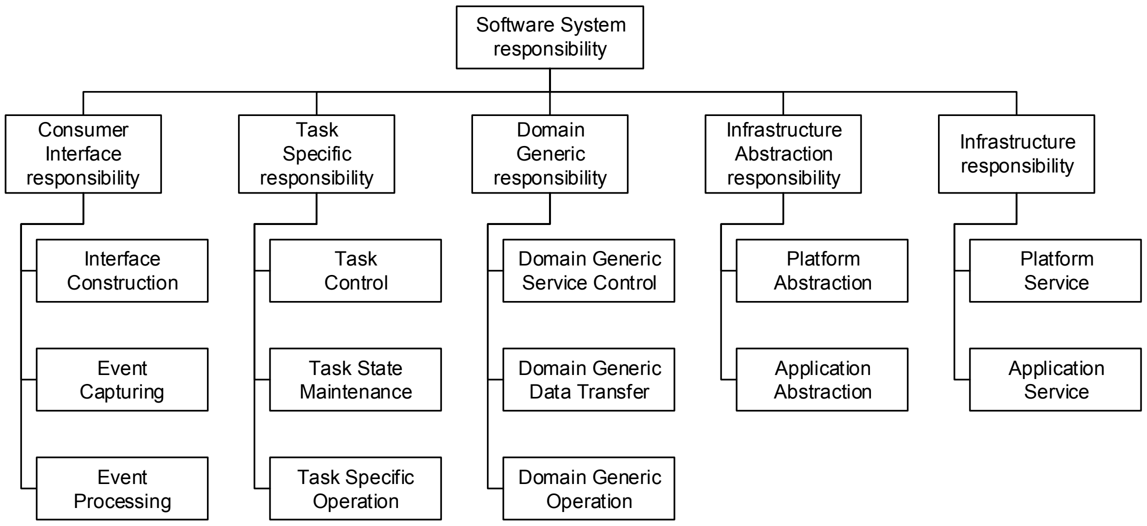
PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15,* 12, 1053–1058.

PRUIJT, L., WIERSEMA, W., AND BRINKKEMPER, S. 2013. A typology based approach to assign responsibilities to software layers.

Appendix

A. TSLR

TSLR describes the following toplevel responsibilities:



Software System responsibility
Consumer Interface responsibility
Task Specific responsibility
Domain Generic responsibility
Infrastructure Abstraction responsibility
Infrastructure responsibility
Interface Construction
Event Capturing
Event Processing
Task Control
Task State Maintenance
Task Specific Operation
Domain Generic Service Control
Domain Generic Data Transfer
Domain Generic Operation
Platform Abstraction
Application Abstraction
Platform Service
Application Service

Fig. 2. Overview of the Typology of Software Layer Responsibility from [Pruijt et al. 2013].

—Consumer Interface responsibility
Takes care of establishing and maintaining communication with a consumer of a system service in a manner appropriate to the task of the consumer. The consumer can be an end user communicating via a user interface, but also an automated client system communicating via a service interface. Consequently, interface and events may have different forms.

Design criteria: Responsibility is...
—Included, when it is specific to the interface of a task.
—Excluded, when it is reusable across different interfaces, which support the same task.

—Task Specific responsibility
Takes care of the coordination of the task, the maintenance of the task state and the execution of functionality specific to the task. A task is a unit of work, to be performed as a whole, which provides the consumer with a result of value. A task is generally, in terms of Cockburn (1997), at the user-goal level. Task Specific responsibility is potentially reusable across different interfaces (e.g. on different platforms) of the same task.

Design criteria: Responsibility is...
—Excluded, when it is specific to a task interface.
—Included, when it is specific to a task.

—Excluded when it is potentially broadly reusable.

—Domain Generic responsibility
is responsible for the coordination and the execution of functionality dealing with concepts, information and rules of the business. Domain Generic responsibility has to do purely with the problem domain and is potentially reusable across different tasks.

Design criteria: Responsibility is...
—Excluded, when it is specific to a task.
—Included, when it is specific to the business.
—Excluded when it has knowledge of infrastructure that has to be abstracted.
—Excluded, when it is reusable across different business applications.

—Infrastructure Abstraction responsibility
is responsible for the translation of infrastructure independent requests into requests dependent on the infrastructure. Infrastructure Abstraction responsibility is separated from other responsibility types, when needed to meet quality requirements like portability, analyzability, and re-usability.

Design criteria: Responsibility is...
—Excluded, when it is specific to a domain or task.
—Included, when it has knowledge of infrastructure that has to be abstracted.
—Excluded, when it is part of an infrastructure platform or infrastructure application.

—Infrastructure responsibility
is responsible for broadly reusable functionality, non-specific to the business. It may be bought, but also self-built, e.g. utilities. Since there are a huge number of infrastructural services, the TSLR connects here with the TOGAF Technical Reference Model (TRM) (The Open Group, 2009). The TRM defines and exemplifies the concepts Infrastructure Application and Application Platform. Furthermore, it provides a typology of the services of the Application Platform.

Design criteria: Responsibility is...
—Excluded, when it is specific to a business application.
—Included, when it is reusable across different applications and/or businesses.