**Logical Architecture and UML Package Diagram**

**What is the Logical Architecture? And Layers?**

The **logical architecture**is the large-scale organization of the software classes into packages (or

namespaces), subsystems, and layers. It's called the *logical*architecture because there's no

decision about how these elements are deployed across different operating system processes or

across physical computers in a network (these latter decisions are part of the **deployment architecture**).

A **layer**is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive

responsibility for a major aspect of the system. Also, layers are organized such that "higher"

layers (such as the UI layer) call upon services of "lower" layers, but not normally vice versa.

Typically layers in an OO system include:

* ·         **User Interface** or UI
* ·         **Application Logic and Domain Objects/Business Layer or Business Logic Layer** software objects representing domain concepts(For example, a software class *Sale*) that fulfill application requirements, such as calculating a sale total.
* ·         **Technical Services/Data Access Layer or Data Layer**general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.



·         **Difference between the Tier and Layer**

1.    Tier is a *physical unit*, where the code / process runs. E.g.: client, application server, database server;

Layer is a *logical unit*, how to organize the code. E.g.: presentation (view), controller, models, repository, data access.

2.    Tiers represent the physical separation of the presentation, business, services, and data functionality of your design across separate computers and systems.

Layers are the logical groupings of the software components that make up the application or service. They help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sublayers, with each sublayer performing a specific type of task.

|  |  |
| --- | --- |
| **strict layered architecture** | **relaxed layered architecture** |
| 1.    This layer only calls upon the services of the layer directly below it. | 1.In which a higher layer calls upon several lower layers. |
| 2.    This design is common in network protocol stacks, | 2.This Design used in information systems, |
| 3.      | 3.For example, the UI layer may call upon its directly subordinate application logic layer, andalso upon elements of a lower technical service layer, for logging and so forth. |

**What is Software Architecture?**

·         Architecture serves as a **blueprint for a system**.

·         It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

·         It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

·         It involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product.

·         The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



[**Software Design vs. Software Architecture**](http://stackoverflow.com/questions/704855/software-design-vs-software-architecture)

·         **Design:** UML diagram/flow chart/simple wireframes (for UI) for a specific module/part of the system

Software design is about designing the individual modules / components. What are the responsibilities, functions, of module x? Of class Y? What can it do, and what not? What design patterns can be used?

·         **Architecture:** component diagram (showing how the different modules of the system communicates with each other and other systems), what language is to be used, patterns...?

The architecture of a system is its 'skeleton'. It's the highest level of abstraction of a system. What kind of data storage is present, how do modules interact with each other, what recovery systems are in place. Just like design patterns, there are architectural patterns: MVC, 3-tier layered design, etc.

**Applying UML: Package Diagrams**

·         UML package diagrams are often used to illustrate the logical architecture of a system layers, subsystems, packages (in the Java sense), etc.

·         A layer can be modeled as a UML package; for example, the UI layer modeled as a package named UI.

·         A UML package diagram provides a way to group elements. A UML package can group anything: classes, other packages, use cases, and so on. Nesting packages is very common. A UML package is a more general concept than simply a Java package or .NET namespace, though a UML package can represent those and more.

·         The package name may be placed on the tab if the package shows inner members, or on the Main folder, if not.

**Construction**

·         A package diagram shows the dependencies between different packages in a system.

·         It is common to want to show dependency (a coupling) between packages so that developers can

see the large-scale coupling in the system. The UML **dependency line**is used for this, a dashed

arrowed line with the arrow pointing towards the depended-on package





**Design with Layers**

**Why we used layers in designing:**

·         Organize the large-scale logical structure of a system into discrete layers of distinct, related

responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers

are low-level and general services, and the higher layers are more application specific.

·         Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is

avoided.

**Software Engineering Layers**



**Using layers helps address several problems:**

·         Source code changes are rippling throughout the system many parts of the systems are

Highly coupled.

·         Application logic is intertwined with the user interface, so it cannot be reused with a different

Interface or distributed to another processing node.

·         Potentially general technical services or business logic is intertwined with more application specific

Logic, so it cannot be reused, distributed to another node, or easily replaced with a

Different implementation.

·         There is high coupling across different areas of concern. It is thus difficult to divide the work

Along clear boundaries for different developers.

**Benefits of using Layers**

1.    Separation of Concerns - at a contract/object level

2.    Allows design of each layer to be individual - you can focus on the task of each - extension of 1

3.    Allows multiple developers to work on each layer at the same time

4.    Allows you to swap out layers / share functionality (thanks for the comment =>)

5.    Each layer can be/is an abstraction of the operational view of your software

6.    Allows for independent testing and verification of layers

7.    Allows your brain to think in that layer not the whole while coding

8.    I have found it easier for junior programmers to on-board to a project faster as they can be assigned "easier" layers

9.    Lower layers contain reusable functions.

10.  Some layers (primarily the Domain and Technical Services) can be distributed.

Guideline: Model-View Separation Principle

This principle has at least two parts:

q  Do not connect or couple non-UI objects **directly** to UI objects.

q  Do not put application logic in the UI object methods.  UI objects should only initialize UI elements, receive UI events, and **delegate requests for application logic to non-UI objects**.

q  Information and behavior related to application logic should be encapsulated within the domain classes.

The **Model-View Separation**principle states that model (domain) objects should not have

*direct*knowledge of view (UI) objects, at least as view objects. So, for example, a *Register*or

*Sale*object should not directly send a message to a GUI window object *ProcessSaleFrame*, asking

it to display something, change color, close, and so forth.

The motivation for Model-View Separation includes:

·         To support cohesive model definitions that focus on the domain processes, rather than on

user interfaces.

·         To allow separate development of the model and user interface layers.

·         To minimize the impact of requirements changes in the interface upon the domain layer.

·         To allow new views to be easily connected to an existing domain layer, without affecting the

domain layer.

·         To allow multiple simultaneous views on the same model object, such as both a tabular and

business chart view of sales information.

·         To allow execution of the model layer independent of the user interface layer, such as in a

message-processing or batch-mode system.

·         To allow easy porting of the model layer to another user interface framework.

**What's the Connection Between SSDs, System Operations, and Layers?**

n  An SSD shows the events generated by the actor/s on the system.  It shows the system operations but it does not show the specific UI objects involved.

n  System operations involve application logic so they should not be operations of the UI layer.  Thus, system operation requests should be forwarded by the UI layer to the domain layer for handling.

