1. **Name examples of the layered architecture. Do they differ or just extend each other?**

The examples of layered architectures as Software Architecture patterns are:

* *Pattern description:* Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database.

Most of the implementations of the layered architectures just differ one from another.

Software operates in layers that allow each component to be independent of the rest.

The main advantages are Encapsulation of hardware, software, and functionality. If a layer is changed, the rest of the layers stay the same.

The main disadvantages are: For small applications, many layers create a performance problem and are very difficult to maintain.

1. **Is the below layered architecture correct and why? Is it possible from C to use B? from A to use C?**

**Chart, box and whisker chart

Description automatically generated**

In this case is not a layered architecture since the main principle in layered architectures is to make one layer be able to use the one next to it in only one direction (example Presentation layer will reference Business Rules layer, but shouldn’t access directly to Data / persistence layer, and also Data layer should not know what is happening on the Business layer). Therefore A should not be able to use C, and C should not be able to use B.

1. **Is DDD a type of layered architecture? What is Anemic model? Is it really an antipattern?**

Domain-Driven Design, initially written in 2003 by Eric Evans, introduced new approaches towards designing software by using a layered architecture with a rich domain model in the center.

Uncle Bob wrote Clean Architecture in 2017 and summarized his research on what constitutes a clean architecture, also using a layered architecture with a domain layer in the center.

The anemic domain model is described as a programming anti-pattern where the domain objects contain little or no business logic like validations, calculations, rules, and so forth. The business logic is thus baked into the architecture of the program itself, making refactoring and maintenance more difficult and time-consuming.

Some authors differ on the idea of considering Anemic Model an anti-pattern since many see also benefits in it, for example:

* Clear separation between logic and data.
* Works well for simple applications.
* Results in stateless logic, which facilitates scaling out.
* Avoids the need for a complex OO-Database mapping layer.
* More compatibility with mapping and injection frameworks expecting dumb properties rather than a specific constructor or property population order.

A common criticism is the idea that anemic domain model makes it easier to follow the SOLID principles:

Single Responsibility Principle, which suggests that a class should do one thing, and do it well. But, according to Robert C. Martin, this is a misunderstanding of that principle:

"Of all the SOLID principles, the Single Responsibility Principle (SRP) might be the least well understood. That’s likely because it has a particularly inappropriate name. It is too easy for programmers to hear the name and then assume that it means that every module should do just one thing. Make no mistake, there is a principle like that. A function should do one, and only one, thing. We use that principle when we are refactoring large functions into smaller functions; we use it at the lowest levels. But it is not one of the SOLID principles—it is not the SRP. (...) the final version of the SRP is: A module should be responsible to one, and only one, actor.”

1. **What are architectural anti-patterns? Discuss at least three, think of any on your current or previous projects.**

Architecture Antipatterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development:

* Good architecture is a critical factor in the success of the system development.
* Architecture-driven software development is the most effective approach to building systems. Architecture-driven approaches are superior to requirements-driven, document-driven, and methodology-driven approaches. Projects often succeed in spite of methodology, not because of it.

Software architecture is a subset of the overall system architecture, which includes all design and implementation aspects, including hardware and technology selection. Important principles of architecture include the following:

* Architecture provides a view of the whole system. This distinguishes architecture from other analysis and design models that focus on parts of a system.
* An effective way to model whole systems is through multiple viewpoints. The viewpoints correlate to various stakeholders and technical experts in the system-development process.

The following Antipatterns focus on some common problems and mistakes in the creation, implementation, and management of architecture.

***Autogenerated Stovepipe***

This Antipattern occurs when migrating an existing software system to a distributed infrastructure. An Autogenerated Stovepipe arises when converting the existing software interfaces to distributed interfaces. If the same design is used for distributed computing, several problems emerge.

***Stovepipe Enterprise***

A Stovepipe System is characterized by a software structure that inhibits change. The refactored solution describes how to abstract subsystem and components to achieve an improved system structure. The Stovepipe Enterprise Antipattern is characterized by a lack of coordination and planning across a set of systems.

***Jumble***

When horizontal and vertical design elements are intermixed, an unstable architecture results. The intermingling of horizontal and vertical design elements limits the reusability and robustness of the architecture and the system software components.

***Stovepipe System***

Subsystems are integrated in an ad hoc manner using multiple integration strategies and mechanisms, and all are integrated point to point. The integration approach for each pair of subsystems is not easily leveraged toward that of other subsystems. The Stovepipe System Antipattern is the single-system analogy of Stovepipe Enterprise, and is concerned with how the subsystems are coordinated within a single system.

***Cover Your Assets***

Document-driven software processes often produce less-than-useful requirements and specifications because the authors evade making important decisions. In order to avoid making a mistake, the authors take a safer course and elaborate upon alternatives.

***Vendor Lock-In***

Vendor Lock-In occurs in systems that are highly dependent upon proprietary architectures. The use of architectural isolation layers can provide independence from vendor-specific solutions.

***Wolf Ticket***

A Wolf Ticket is a product that claims openness and conformance to standards that have no enforceable meaning. The products are delivered with proprietary interfaces that may vary significantly from the published standard.

***Architecture by Implication***

Management of risk in follow-on system development is often overlooked due to overconfidence and recent system successes. A general architecture approach that is tailored to each application system can help identify unique requirements and risk areas.

***Warm Bodies***

Software projects are often staffed with programmers with widely varying skills and productivity levels. Many of these people may be assigned to meet staff size objectives (so-called “warm bodies”). Skilled programmers are essential to the success of a software project. So-called heroic programmers are exceptionally productive, but as few as 1 in 20 have this talent. They produce an order of magnitude more working software than an average programmer.

***Design by Committee***

The classic Antipattern from standards bodies, Design by Committee creates overly complex architectures that lack coherence. Clarification of architectural roles and improved process facilitation can refactor bad meeting processes into highly productive events.

***Swiss Army Knife***

A Swiss Army Knife is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds many interface signatures in a futile attempt to meet all possible needs.

***Reinvent the Wheel***

The pervasive lack of technology transfer between software projects leads to substantial reinvention. Design knowledge buried in legacy assets can be leveraged to reduce time-to-market, cost, and risk.

***The Grand Old Duke of York***

Egalitarian software processes often ignore people’s talents to the detriment of the project. Programming skill does not equate to skill in defining abstractions. There appear to be two distinct groups involved in software development: abstractionists and their counterparts the implementations.

1. **What do Testability, Extensibility and Scalability NFRs mean. How would you ensure you reached them? Does Clean Architecture cover these NFRs?**

**Testability:** Covering the levels of test coverage in place, the effectiveness and efficiency of tests, and the quality of testing reporting, the testability quality attribute requirement relates to how confident teams can be that the system will function as intended. In these cases, it’s important that the software testing/Quality Assurance (QA) team are closely involved with the project from an early stage, so that they can advise on the most suitable testing approaches for each feature, as well as for other QARs you may have in place. These approaches can include automated functional tests, integration and contract tests, manual tests, and third-party or bespoke testing tools.

Clean Architecture also makes it easier to test and debug the code. Because the inner circle is independent of the outer layers, it's easier to write unit tests that focus specifically on the business rules. This can help to catch errors early in the development process and reduce the overall testing effort.

**Extensibility:** Extensibility means that the code can easily be extended its functionality based on the existing one without breaking changes. Is a manner on how the software has been built to be open for “extension” and close for modifications, satisfying SOLID principles.

The main principle of Clean Architecture is a clear separation of concerns, and its main goal is to help to create a clear separation of concerns within the codebase. Each layer has a specific purpose and is decoupled from the others, making it easier to understand and modify individual components without affecting the rest of the system. This modularity also makes it easier to reuse components in other projects.

**Scalability:** Scalability means that the system must be able to accommodate larger volumes (whether of users, throughput, data) over time, and includes NFRs such as elasticity, which is the ability to scale up and down quickly, as needed. Today, scalability can be achieved more easily than in the past thanks to modern cloud-based solutions, which have the infrastructure needed to auto-scale according to requirements.

Clean Architecture can easily be broken down into multiple modules or services, if performance is suffering and there is a need to scale out then the individual components can be scaled out without major impact on other components that aren’t on an high load