Sarada Sastri · Follow
Apr 17, 2021 · 9 min read · Listen

Software Architecture — Principles, Practices & Styles

Designing the right architecture for a problem statement is more of an art than a science because it depends a lot on the understanding of the problem statement, the context, and where we think it will expand next. The most important thing about any architecture is how adaptable it is in the face of changing requirements of business and scale. Below are my experiences of how different architecture styles, principles, and methodology come together to form an architecture that is evolution-ready.

What is bad architecture and how to recognize it?

For the sake of velocity of development, developers often put in bad code, which ultimately leads to what we traditionally call as the spaghetti code. This leads to functional paralysis at some point in time, where the cost of building afresh is cheaper than fixing the existing code. Some of the characteristics are

1. **Unnecessarily Complex** — Ironically it is easy to write complex code, anyone can do it, but it is hard to write simple code.
2. **Rigid/Brittle** — Since it is unnecessarily complex, it is not easy to understand and therefore making it non-maintainable, easy to break for even a small code change.
3. **Untestable** — Such code will be tightly coupled, will typically not follow the single responsibility principle, will be difficult to test.
4. **Unmaintainable** — Brittle code with less test coverage evolves to becomes a maintenance nightmare

What is good architecture and what properties do they exhibit?

1. **Simple** — Easy to understand.
2. **Modularity/Layering/Clarity —** This is important so that one layer is able to change independently of the others with minimum coupling between the layers
3. **Flexible/Extendable**— Can be easily adapted to new evolving requirements
4. **Testable/Maintainable** — Easy to test, add automated tests, and encourage the culture of TDD and therefore maintainable

Why bother about architecture, principles, practices?

Cost reduction — Though initially, the velocity of development will be less, but eventually, the overall cost of building and maintenance will be less

Build what is essential — It allows us to build the most essential and necessary parts. it is important to build what is necessary when it is necessary not before that. This approach helps clear the clutter by building only what is essential thus reducing the overhead in code maintenance.

Optimization — Optimize for better maintainability. It is the developers, the users for whom the optimization should be done upfront

Performance Optimization — While planning and designing a system that can evolve for performance, keep in mind that the code level optimization for performance that compromises the optimization done for maintainability should be deferred till the LRT.

Last Responsible Time — LRT is a concept borrowed from lean principles where decisions/changes are deferred till a point in time beyond which the cost of not making the decision will become costlier than making the decision. When requirements are grey, design decisions should be put off till the LRT so that we get enough knowledge by then to make sound design decisions.

Adaptability/Evolution — Following the above, software always follows an evolutionary pattern when it keeps adapting itself to new demands of business and scale

2. **Agile methodology**— Build the right way. Build software in a way that is agile, adaptable, fast to respond to changing market requirements
3. **Test-driven development practice & Automated Tests** — Test drive implementation ensuring testable software design. This supports the Shift Left methodology, "Test early, Test often" resulting in maintainable code as it eliminates the fear of breaking existing functionality un-intentionally.

What architecture styles are followed?

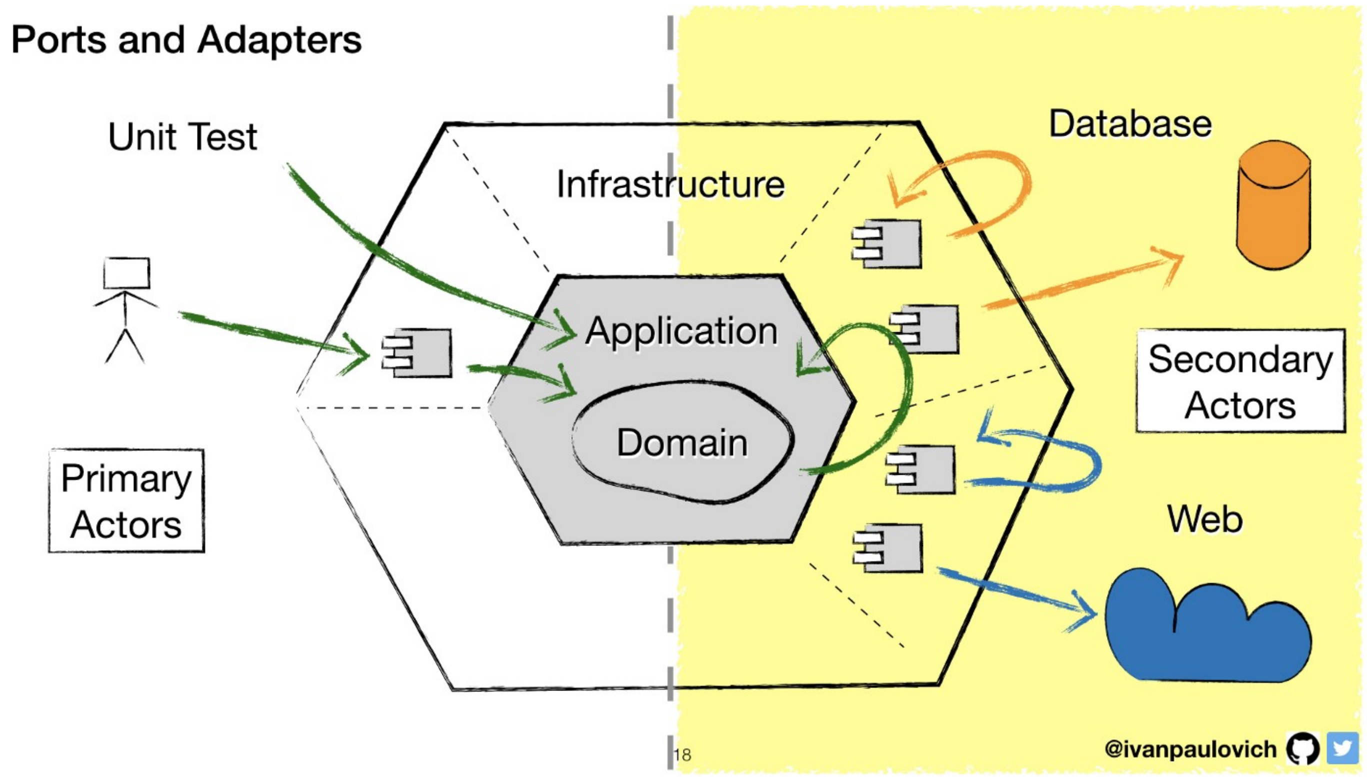
Typically there is never one size fits all. Design decisions for a problem statement depend on the context and every design has its trade-off. Below are some of the most commonly used architecture styles that typically come together. What combination fits best for your application is best known to you.

1. Domain Centric Architecture
2. Application Centric Architecture
3. Screaming Architecture
4. Microservices architecture
5. Event-Driven Architecture — EDA
6. Command Query Responsibility Segregation — CQRS

Domain Centric Architecture

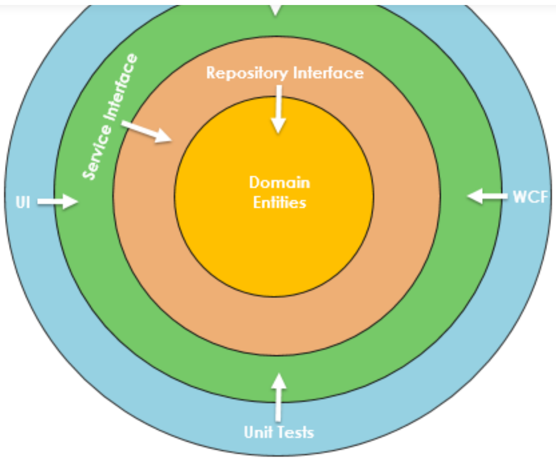
The domain is at the center of the model and everything else is built around it, the application layer, the presentation layer, the persistence layer, notification service, web services, etc. i.e. Domain is essential and everything else is just an implementation detail that is replaceable.

The domain here represents the mental model of the users of the system. This is the most stable portion of the architecture that rarely changes. This is followed by the application layer which embeds the use-cases. These use-cases define everything else.



Original source: https://images.app.goo.gl/cW5QmNMxn912DM4D6

There are 2 such domain models defined, the hexagonal and the onion model. The essence is, each of the outer layers depends on the



Original Source: https://images.app.goo.gl/dRgpzwPR9w8u5u4t7

Pros

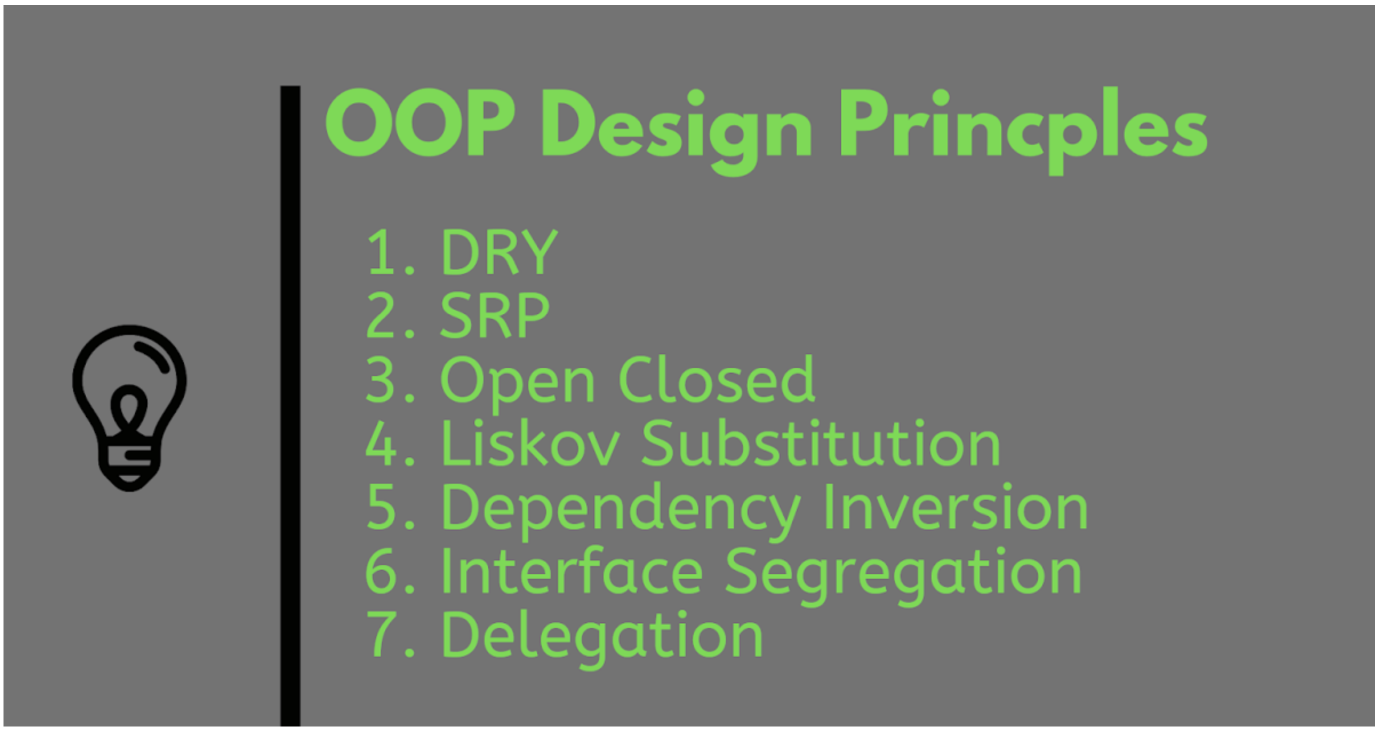
1. This allows for Domain-Driven Design (DDD) thinking. The focus is on a domain, users, and use-cases.
2. Reduced coupling between domain (stable and changes less) and the implementation detail (which changes faster like presentation layer, database)

Cons

1. The initial cost is higher as more time/thought/discussion has to be put into separate models needed for the domain versus application layer rather than all models just put together.
2. Developers tend to avoid it, as it requires more thought. They stick to the old 3 layered Database Centric Architecture

Application Centric (Layered Architecture)

Once the domain boundary is defined, the application layer comes next. The application layer is made robust by applying the below SOLID principles.



OOP Design Princples

1. DRY
2. SRP
3. Open Closed
4. Liskov Substitution
5. Dependency Inversion
6. Interface Segregation
7. Delegation

Original source: https://images.app.goo.gl/UwBEyStMHaVVJt7u5

Segregation(the HOW?) — it is done, the implementation details aspect is pluggable using dependency injection**(DI)**. DI not only applies for injecting business logic using various design patterns but it specifically holds true when injecting infrastructure elements like databases, caches, notification servers, external web services, etc.

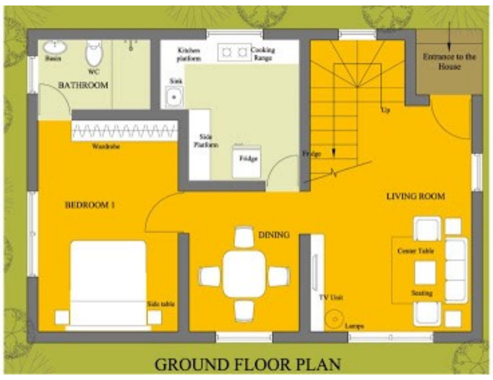
Interfaces/Contracts (Handshakes)— Such an approach automatically builds a layered architecture with clear interfaces for each external element. The segregation of responsibility coupled with each layer owning one single responsibility reduces coupling. This in turn helps in easy-to-test code which can be unit tested too using mocks.

Again the application layer does not depend on any of the other implementation details and has only the knowledge of the domain layer on which it depends.

Functional Organization of code (Screaming Architecture)

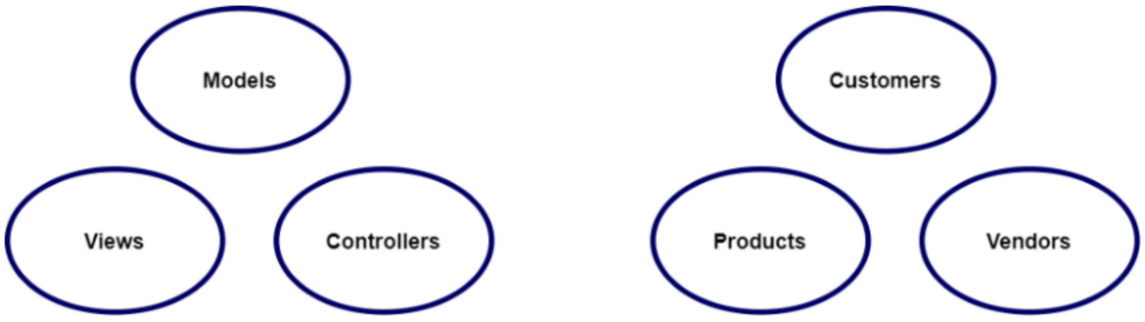
The architecture should scream the intent of the system — Uncle Bob

This is best explained with a blueprint of a residential building, as to how each room clearly spells the intent and the use.



Original Source: https://images.app.goo.gl/3am8cnt6BrzFzh3JA

For the backend layer — we can modularize the code in functional segments by folder structure. Code with functional cohesion is kept together. Each module can have an aggregate root as the single point of entry to the module, and thus just looking at the aggregate root we should be able to spell out all the use-case of the module. Thus simplifying the functional intent of the module.

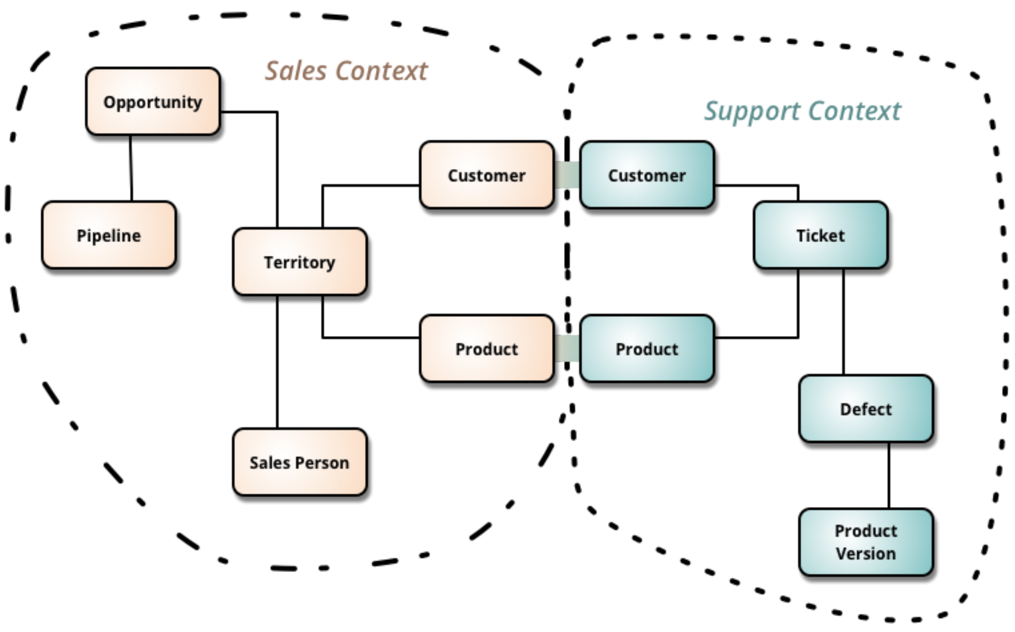


Functional Organization and Categorical Organization

Original source: https://levelup.gitconnected.com/let-me-hear-you-screaming-architecture-3adcc02f2ca3

As for the presentation layer — This may still want to follow the old categorical approach of models/views/controllers. The presentation layer should be kept lightweight with no business logic. This helps in 2 ways. First, we eliminate duplication of logic. Second, such an organization helps the UI-junior developers to concentrate on just making the UI rich.

Microservices Architecture



https://images.app.goo.gl/HHfv3ojn17B5L1dU7

Bounded Context —It is the recognition of a specific contextual scope within which a specific terminology of the domain model is valid and makes sense.

In the new model, you don't have to fit a “Contact” into a “Customer” in the support domain. But rather use the right terminology “Contact” in the support domain. When you want to talk to the Sales domain you convert the “Contact” into the “Customer” object using well-defined interfaces. This also leads to high functional cohesion and reduces coupling across different domains.

Microservices defined — They divide a single monolith into sub-systems that take up single responsibility as a service. They communicate with each other with clearly defined interfaces. They have autonomous deployment, have their own independent database and backing services. Each microservice is independent to choose the technology stack, tools, practices that best suit them. They scale independently. The teams are relatively small. It is possible one team takes care of one or more microservice. Each team just needs to know the domain knowledge of the microservice they are accountable for and not everything in the monolith.

Cons

1. The initial cost is higher
2. DevOps automation is necessary as automated deployment is now a need.
3. Extra time and cost is incurred to deal with such distributed computing in terms of latency, load balancing, logging, monitoring, dealing with eventual consistency, etc

Event-Driven Architecture (EDA)

Microservices can now communicate with each other using a request/response mechanism like JSON over REST calls OR using an event-driven architecture with a message-broker. Modern architecture prefers EDA as it allows the services to more responsive, reduced latency, robust, fault-tolerant, guaranteed service, and allows to scale better.

In EDA there are 3 participants namely a producer that creates the triggering event, a message broker that carries the message in a robust manner, and a consumer which can subscribe for select/all events. This leads to “Reactive Programing” which reacts to the event(trigger) coming from the data stream resulting in faster response time and therefore low latency.

For microservices that need the ACID properties of transaction across microservices using eventual consistency, instead of EDA, the SAGA pattern is used where there is an explicit rollback mechanism to handle error conditions to roll back changes. This however complicates the design, so should be used with prudence.

optimization is achieved in Event-Sourcing by creating snapshots of the current state at fixed intervals.

CQRS pattern-Command Query Responsibility Segregation

The advent of microservice and EDA also has given birth to the CQRS pattern. The command is something that modifies the state of the underlying object & the query does not modify the object but just returns the requested subset of objects.

How is this useful? Few examples

1. You can improve your read scalability without impacting the write. For example, by adding more secondary nodes in MongoDB to serve the read requirements, you selectively scale the read capability.
2. The command has to send the updates to the database. You may choose to have a caching layer to provide for faster reads.
3. Sometimes the object may belong to another microservice and querying another microservice every time may be costly, so you can use the caching layer for your query needs. Data is duplicated but as long as it is maintained, and is not changing very fast, you are able to reduce the latency to a good extent. This sometimes offers resilience too. Even if the other microservice is not available your microservice can continue to work normally. ex. Caching the product catalog in the Order-microservice.

The above are a few high-level design choices and practices that are often used. Again, these are used in conjunction with several other low-level design choices which is a combination of different design patterns, principles, tools, etc. All this stitched together in a meaningful way goes on to define a solution that is agile, adaptable, extendable, maintainable, testable, and most importantly simple.