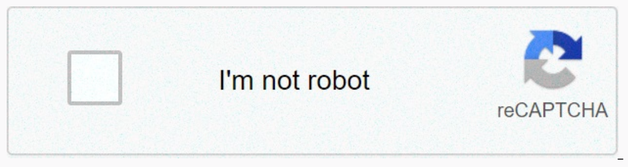

Continue

Design patterns explained pdf github

Instantly share code, notes, and snippets. You can't do that action right now. You sign in with another tab or window. Reload to refresh your session. you exit in another tab or window. Reload to refresh your session. A curated list of software and architectures related to design patterns. Software design patterns – Common reusable solutions to common problems occurring in certain contexts in software design. This is a description or template for how to solve problems that can be used in a variety of situations. Contents Programming Language Design Patterns General Architecture Cloud Architecture Microservices Serverless Architecture & Distributed System Internet of things Big Data Databases Docker and DevOps Mobile Front-End Development Security Books Other Awesome Lists Other amazing lists can be found in awesome lists. Granting the License As far as possible under the law, Dov Amir has waived all copyrights and related rights or neighbors for this work. Page 2 Watch 475 Star 11.5k Fork 914 You cannot perform that action at this time. You sign in with another tab or window. Reload to refresh your session. you exit in another tab or window. Reload to refresh your session. You can pull the entire project to take each sample pattern, or just check the pattern folder. Each key class of pattern (containing the main method) uses this format: #name#Pattern (e.g. factoryPattern) Other pattern examples are still in development :) What are design patterns? In software engineering, software design patterns are a common reusable solution to common problems occurring in certain contexts in software design. This is not a finished design that can be converted directly into a source or machine code. This is a description or template for how to solve problems that can be used in a variety of situations. Design patterns are formalized best practices that programmers can use to solve common problems when designing applications or systems. Design pattern in The Creation Factory design pattern link project to the Create object package without specifying the right class to create illustration by RTS unit creation system Singleton design pattern link to package Restricting object creation to classes using only 3 instances Using 3 different approaches (not safe for threading, safe for threading and safe for threading using synchronized) Structural Decorator link pattern for dynamically adding/overriding package behavior in existing methods of objects illustrated by dessert + material name and price printing Behavorial Observer link pattern for package Object, called subject, maintaining its dependent list, called observer, and know them automatically from any change of status. Used primarily for the handling of events illustrated by the display of GPS notifications on changes in position / precision What ? A summary of design patterns is described in the book Design Patterns in Ruby, in which Russ Olsen explains and adapts to Ruby 14 of the original 23 GoF design patterns. Design Patterns GoF Patterns Adapter: helps two incompatible interfaces to work together Builder: create complex objects that are difficult to configure Command: perform certain tasks without having information about composite request recipients: build a hierarchy of tree objects and interact with them all in the same way Decorators: vary the responsibility of adding objects some features of the Factory: creating objects without having to specify the exact class of objects that the Interpreter will create: provides a specific language for solving well-defined problems of knowing the Domain Iterator: provides a way to access the collection of sub-objects without exposing the Observer's underlaying representation: helps build a highly integrated system, can be maintained and avoid connectors between Proxy classes : allows us to have more control over how and when we access certain Singleton objects : has one particular class example throughout the application strategy: varied parts of the algorithm on the Runtime Method Template: redefining certain steps of the algorithm without changing the algorithm structure of the Non-GoF Pattern: Patterns For Ruby Contribution Contributions are welcome! What can you do?: Find typos and grammar Errors Propose a better way to explain patterns Add clearer examples of using patterns Add other GoF patterns that aren't included in the Sample PR refactoring code book won't be considered. The examples provided by Russ Olsen in his book are meant to be simple and clear, not the best performing or most elegant, their only educational goal. This project is a study of object-oriented and functional design patterns, applied at Scala. It's for my own studies and future references but if you find the subject matter interesting and if the information here helps you in your studies then I'll be happy! :) I sure have a good understanding of design patterns, and sufficient insight/experience when implementing them is key to creating agile software and therefore supports agile software development goals such as Agile Discipline Delivery. Patterns always have two parts: how and when. Not only do you need to know how to implement it, you also need to know when to use it and when to leave it alone. Core design principles Martin Fowler Core Design Principles for Software Developers by Venkat Subramaniam There are various categories of software patterns: Architectural patterns: large broad things that can describe the entire system They are agnostic programming languages, but talk about how all the different subsystems and components fit together, Design patterns: patterns that describe how things happen in components, how a program is structured, how data flows, how execution flows, etc.. Idioms: (pattern-oriented software); specific to one programming language; examples of design patterns applied in programming languages, or smaller local patterns seen in a particular programming language What is anti-pattern? Anti-pattern is the opposite of pattern; while it also describes recurring solutions to common problems encountered, the solutions are usually dysfunctional or ineffective, and have a negative impact on the health of the software (in terms of sustainability, extensibility, resilience, etc.). Anti-patterns serve the same purpose for patterns; Anti-pattern descriptions may describe typical anti-implementation patterns, explain the context that generally occurs, and show how implementations generate problems for software. The potential problem with the concept of anti-pattern design is that it might prevent critical thinking about the application of patterns. A design that may be inappropriate in some contexts may be a sensible decision in others; the solution may be discarded once it is recognized as anti-pattern, although it will be suitable for the problem at hand. Why do we need patterns? We need to know the design patterns to find solutions to frequent problems. And we want to re-use this solution whenever we face a similar situation in the future. This is one type of template to overcome solutions in many different situations. In other words, it's a description of how different objects and their classes each solve design problems in a particular context. The following patterns (23) describe creative, structural, and behavioral patterns fully described in the Gang of Four book 'Design Patterns: Elements of Reusable Object-Oriented Software' Creational (5) Creation patterns are patterns that create objects for you; therefore creation. The goal here is, rather than having an instant object of your code directly and therefore having all the logic on how to create (sometimes complex initialization logic around the object), the creation pattern will do that for you. This has a second benefit, since it has all the logic of creation in one place, a change in that logic will spread to where a new object is needed. Creation patterns provide guidance on creating objects. They help hide instantiation details of objects from code that uses those objects. That is they make an independent application of how the object is created, composed and represented. This leads to high cohesion between objects and their users, but a low pairing between the user and the way the object is created. For example, if I have a Java interface implemented by three different classes, then use Factory I I instantiate one of the three classes depending on the current situation. All users of the returned object need to know is what interface they all apply. Actual implementations are subject to change each time the plant is used, but these details are hidden. There are 5 different patterns in the creation pattern category. They are Factory Method, Abstract Factory, Builder, Prototype and Singleton. These patterns are presented below: Factory method patterns: Factory method patterns provide patterns that explain the use of factory classes (or methods) to build objects. Method on factory return object that implements certain interface. Factory users only know about the interface. Thus different objects can be created depending on the current situation (as long as they implement the interface). create objects without specifying the right class to create. Abstract factory patterns: Abstract factory patterns describe patterns for creating families of related or dependent objects. Builder pattern: The builder pattern separates the construction of complex objects from their use. Thus the client can determine what type of object is required and its content possible, but it is not necessary to know about how the object was built and initialized. Prototype pattern: The prototype pattern allows the user object to create a customized object, based on what prototype is required. This means that the pattern explains how new objects can be created based on existing object adjustments. Singleton pattern: The singleton pattern describes a class that can only have one object built for it. That is, unlike other objects it should not be possible to get more than one instance in the same virtual machine. Thus the Singleton pattern ensures that only one instance of the class is created. All objects that use instances of that class use the same instance. The motivation behind this pattern is that some classes, usually classes involving resource center management, must have one proper example. For example, an object attributed to the re-use of a database connection (i.e. a set of connections) could be a singleton. Structural patterns (7) Structural organization describe objects. That's how classes and objects are arranged to form larger structures. For example a large department store near where I live, appears to form the outside into a single entity with a very magnificent frontage. However, behind this frontage is a completely new store containing a variety of independent stores. This means that as a customer I see forming an outer one and quite magnificently intact. But the inner form there are some smaller stores/brands all working together. This is the essence of the Facade pattern. Structural patterns concern the composition of classes and objects; therefore Structural. They use inheritance to interface and define ways to build objects to get new functionality. Proxy pattern: provides placeholders for other objects to control access, reduce costs, and reduce complexity. Flyweight pattern: reduces the cost of creating and manipulating a large number of similar objects. Bridge pattern: separate abstraction from its implementation so that both can vary independently. Façade pattern: provides a simplified interface to the large content of the code. Decorator pattern: dynamically adds/overrides behavior in existing object methods. Adapter pattern: allows classes with incompatible interfaces to work together by wrapping their own interface around an existing class. Composite patterns: arrange objects zero or more similar so they can be manipulated as a single object. Behavior patterns (11) Behavior patterns are specifically related to communication between objects, hence behavior. Here our concentration is on algorithms and the assignment of critical responsibilities among objects. We also need to focus on the communication between them. We need to take a closer look at how they relate. Behavior patterns relate to organizing, managing, and assigning responsibility to objects during execution. That is, the focus of behavior patterns is on communication between objects during the implementation of several tasks. Typically, these patterns characterize complex control flows that are difficult to follow at process time. They therefore help shift the emphasis from low-level control flows to higher-level object interactions. Pattern of chain of responsibility: the delegate instructs to chain the processing object. Command pattern: creates an object that summarizes actions and parameters. Translator pattern: implement a specific language. Iterator pattern: access object elements sequentially without exposing the underlying representation. Mediator pattern: allows loose connectors between classes by being the only class that has detailed knowledge of their methods. Keepsak pattern: gives the ability to restore the object to its previous state (undo). Observer pattern: is a publication/subscription pattern that allows a number of observer objects to see an event. Status pattern: allows an object to change its behavior when its internal state changes. Strategy pattern: allows one of the algorithm families to be selected while running. Template method pattern: defines the algorithm framework as an abstract class, allowing its subclasses to provide concrete behavior. Visitor pattern: separates the algorithm from the structure of the object by moving the hierarchy of the method into one object. Replacing Object-Oriented Patterns with Functional Patterns Replaces Functional Interfaces Replacing functional interfaces that carry countries Replace Replace command for irreversible Objects Replacing Iterator Changing Template Method Replacing Strategy Replace Dekorator Replcing Null Object Replacing Visitor Replace Monoid Pattern Injection Dependence Pattern Functor Pattern Functor Applicative Functor Type Pattern Tail Class Recursion Mutual Recursion Filter-Map-Reduce Chain of operations Builder Memoization Lazy Sequence Focused Mutability Customized Control Flow Domain-Specific Language A registry is a list of items with such an index in the database table or card catalog for the library. If you lose the registry, the item still exists; You just might need to rediscover them. The Repository repository is an abstraction of a collection of objects. The repository stores actual items (objects), such as the database table itself or the library bookshelves. If you lose the repository, the item is lost. The repository is closer to the domain. It acts on aggregate roots (domain objects/entities) and will use multiple DAOs to build a single entity. Spring repository Documentation CQRS Domain Repository Repository CQRS domain repository gets the aggregate root by its id, and loads/stores the resulting event. DAO Pattern DAO Pattern Data Access Object Patterns are used to separate low-level data that access API operations from high-level business services. The following are participants in the Data Access Object Pattern. Data Access Object Interface: the interface determines the standard operation to be performed on the model object. Concrete Class Data Access Object: concrete class that implements the above interface. This class is responsible for obtaining data from data sources that can be databases/xml or other storage mechanisms. Model Object or Value Object: an object managed by DAO. Usually objects are of modest value as for example. case class. DAO ensures that specific implementations are abstracted far behind the interface so that implementations can be switched. It also ensures modularization and cohesion of correct functionality. DAO returns data in the broadest sense of the word and is a very definition (data access object). How DAO accesses data, be it by accessing queues, XML files, or by requesting one or more un specifyed tables. Thus, DAO handles the problem of persistence and is an abstraction of data persistence. It's also closer to database/persistence than the repository would be. The repository relates only to domain objects In trenches, DAO is sometimes called a provider and Martin Fowler defines it as the Data Gateway Table pattern. Facade pattern The pattern of the facade pattern is used to hide the complexity of the system call. It provides a simple abstraction, one method, that abstracts all the complexities of the call to multiple subsystems. For example, clients do not need to know that, to return results, a certain amount of (local or remote) involved. So, the facade deals with controls and workflows. Services From: StackExchange It is not easy to determine what the responsibility of the service is. The service is not a canonical or generic software term. In fact, the sying-along service in the class name is very similar to the much vilified Manager: It tells you there is almost nothing about what the object actually does. In fact, what the service should do is very specific architecture, so first you have to determine the architecture and then the responsibility of the service will become clear. Traditional Layered Architecture In traditional layered architecture, service is literally synonymous with layers of business logic. This is the layer between UI and Data. Therefore, all business rules go into service. Data layers should only understand basic CRUD operations, and UI layers only have to deal with mapping Data Transfer Object (DNO) presentations to and from business objects. RPC style In rpc style distributed architecture (SOAP, UDDI, BPEL, etc.), this service is the logical version of the physical endpoint. This is basically a collection of operations that the maintainer wants to perform as a public API. Various best practice guidelines explain that service operations should actually be business-level (capability) operations and not CRUD, and I tend to agree. However, since merute everything through a remote service that can actually seriously hurt performance, usually it is best not to have this service actually apply the logic of the business itself; instead, they must wrap up a set of internal business objects. One service may involve one or more business objects. Model-View-Controller (MVC) In MVP/MVC/MVVM/MV* architecture, service does not exist at all. Or if they do, the term is used to refer to any generic object that can be injected into the controller or display model. Business logic is in your model. If you want to create a service object to set up complex operations, it is seen as an implementation detail. Many people, unfortunately, apply MVC like this, but it is considered an anti-pattern (Anemic Domain Model) because the model itself does nothing, it's just a bunch of properties for the UI. Some people mistakenly think that taking a 100-line controller method and pushing it all into service somehow makes for better architecture. It really isn't; all it does is add another indirect layer that may not be necessary. Practically speaking, the controller is still doing the job, it's just doing it through a badly named helper object. I highly recommend Jimmy Bogard's Evil Domain Model presentation for a clear example of how to turn an anemia domain model into a Useful. This involves careful examination of the model you are exposing and which operations are completely valid in a business context. For example, if your database contains and you have a column for Total Amount, your application may not be allowed to actually convert that field into an arbitrary value, because (a) it's history and (b) it should be determined by what's in the order as well as perhaps some other time sensitive data/rules. Creating a service to manage Orders doesn't always solve this problem, as the user code can still retrieve the actual Order object and change the amount on it. Instead, the order itself must be responsible for ensuring that it can only be changed in a safe and consistent manner. Domain Driven Design (DDD) In DDD, the service is intended specifically for situations when you have incorrect operations belonging to any aggregate root. You have to be careful here, because often the need for service can imply that you are not using the correct root. But assuming you do, the service is used to coordinate operations at various roots, or sometimes to address issues that do not involve a domain model at all (such as, perhaps, writing information to a BI/OLAP database). One important aspect of the DDD service is that it is allowed to use transaction scripts. When working on large applications, you are very likely to end up experiencing instances where it's just an easier way to achieve something with a T-SQL or PL/SQL procedure than to bother with a domain model. It's OK, and it belongs in service. This is a radical departure from the definition of layered architectural services. The service layer summarizes domain objects; DDD services summarize anything that is not in the domain object and does not make sense to be. SOA In Service Oriented Architecture, services are considered a technical authority for business capabilities. That means that it is the exclusive owner of a certain subset of business data and no one else is allowed to touch that data – not even just read it. By necessity, service is actually an end-to-end proposition in SOA. That is, the service is not so many specific components as the whole stack, and your entire application (or your entire business) is a set of these services running side by side without intersections except in the message layer and UI. Each service has its own data, its own business rules, and its own UI. They don't need to ororize with each other because they are supposed to be aligned with the business – and, like the business itself, each service has its own responsibilities and operates more or less

independently of the other. So, by the definition of SOA, every piece of business logic anywhere is contained in the service, but then again, so is the whole system. Services in the SOA can have components, and they can have end, but dangerous enough to call a piece of service code because it contradicts what the original S should mean. Because SOA is generally pretty pretty on the message, operations that you may have packaged in the previous service are generally encapsulated in the handler, but remain different. Each handler handles one type of message, one operation. This is a strict interpretation of the Principle of Sole Responsibility, but makes great care because of every operation that may exist in its own class. So you don't really need centralized business logic, because orders represent business operations rather than technical one. In conclusion in the end, in whatever architecture you choose, there will be some components or layers that have most of the business logic. After all, if business logic is scattered all over the place then you only have spaghetti codes. But whether you refer to that component as a service or not, and how that component is designed in terms of things like the number or size of operations, depends on your architectural goals. There is no right or wrong answer, just what applies to your situation. To add to the discussion, the service may be: Stateless, Stateful or , Be local or remote Will be used by the client (local or remote client) through the Service Contract i.e. Interface Depending on technology (stack), can be provided with container services such as security and transactions. Where to put your business logic in JavaEE is where to put business logic in the business logic layer or just logic. These are objects that communicate with each other, optionally accepting services such as security and transactions. As we use Actors, a place to put business logic in traits. When using functional programming styles, Actors can manage countries and business logic can be mixed and thus reused where appropriate. Anemic Domain Model Anemic Domain Model is the use of a software domain model in which domain objects contain little or no business logic (validation, calculations, business rules, etc.). This pattern was first described by Martin Fowler, who considered the practice to be anti-pattern, which does not mean that it is anti-pattern, it is the opinion of Martin Fowler. In anemic domain design, business logic is usually applied in separate classes that change the status of domain objects. Fowler called the script an external class transaction. This pattern is a common approach in Java applications, perhaps driven by technologies such as early versions of EJB Entity Nuts, as well as in .NET applications following the Three Layer Service Application architecture where the object belongs to the Category of Business Entity (although the Business Entity may also contain behavior). The reason why the model of the absence of business logic is anemic Almost total, as in applications that are mainly the assembly of CRUD screens? Service-oriented architecture where the 'domain object' is actually data transfer object? Political or pragmatic considerations such as code ownership or backward compatibility that excessively inhibits refactoring? Apply procedural/relational design in an object-oriented language? In any case, if I choose a simple rule of thumb for the boundary between domain model logic and service logic, it is that interacting with related objects is fine within the domain, while accessing the outside world (user interface, web services, etc.) may not be included in the domain model. Another point of view is the anemia domain model = database table mapped to an object (field values only, no real behavior) A rich domain model = set of objects that exposes behavior If you want to create a simple CRUD application, perhaps anemic model with the classic MVC framework is enough. But if you want to apply some kind of logic, the anemic model means you won't do object-oriented programming. Note: Object behavior has nothing to do with persistence. Different layers (Gateways/DAO's, Data Mappers, Repositories, etc.) are responsible for maintaining domain objects. The main point is that the domain model implements logic and operates in the state. Tiers vs Layers A tier is the limit of existing modules to provide insulation between the main components of the system. The presentation level visible to users is separate from the application level which includes business logic. In turn the data level is another separate system that can manage, survive, and provide access to data. Components grouped in tiers can be physically located on different infrastructures. Layers are logical slices that carry our special responsibilities in the application. Each level can have multiple layers in it responsible for various elements of functionality such as domain services. DTO'S, DDD & The Anemic Domain Model I agrees that the domain model is bad anemia, if there is no behavior then what is the point right? DTO: For me, DTO moves data between 'levels'. They are packaging data that is ready to be transported. The JSON object is also a DTO. Read the model: This will be a different model from your actual Domain model. Read models are very light, thin and anemic. The goal is to present aggregated data to a specific screen or message. DTO, for me, can be a reading model, can also be a Display Model. Domain model: rich and full of behavior. This model is most valuable when performing complex business rules during storing and updating data in certain transactions. It can also be used to read data as well. Splitting models allows &reading; the writing fluctuates independently, thus leading to higher maintenance. These models can also run at different levels/nodes to increase scalability (readability rate/ writing level), all of which are options. At some point, either from the view or the incoming DTO, there will be a mapping back to the domain model. It's 'friction' or fairly easy to manage using assemblers/translators, or tools like AutoMapper. Greg Young & Udi Dahan took this concept further and applied a programming principle called Command-Query Responsibility Separation/Seggregation with distributed programming and SOA. Model Domain Anemia (ADM) is not Anti-Pattern I would argue that such an anti-pattern is the Domain Model Anemia (ADM). ADM is regarded by this author as a failure to model solutions in an Object Oriented way, instead relying on procedural design to express business logic (which is the perfect definition of ADM). This approach contrasts with the Rich Domain Model (RDM), where classes representing domain entities summarize all business logic and data. While ADM may of course be a poor design choice in some systems, it is not clear that this is the case for all systems. In some scenarios, ADM seems to be a sensible design choice, in terms of compliance with [SOLID principles of Object Oriented design](object-oriented_design). SOLID principles are guidelines that seek to balance simplicity of implementation, scalability, and durability. In particular, by contrast adm design with RDM design for hypothetical problems. I will try to show that ADM is more suited to SOLID principles than RDM solutions. ADM consists of a series of behavior-free classes containing the business data needed to model a domain. These classes usually contain little or no business logic (validation, calculations, business rules, etc..), instead, business logic is applied by the domain service layer/ logic. The domain/logic service layer consists of a set of types and functions that process the domain model as determined by business rules. The argument against this approach is that the data and methods are divorced, violating the basic principles of Object Oriented design by removing the ability of the domain model to enforce its own invariants. Conversely, while RDM consists of the same set of types containing the necessary business data, domain logic is also fully populated on this domain entity, expressed as a method. RDM then aligns with the concept of encapsulation and concealing information; as Michael L. Scott points out, the Encapsulation mechanism allows programmers to group data and subroutines that operate together in one place, and to hide irrelevant details from users of abstraction. In RDM, the domain service/logic layer is very thin or none, and all domain rules are applied through the domain model. The content is that the domain entities in the RDM are then fully capable of enforcing their invariants, and therefore This system reads from an Object Oriented design perspective. However, the ability of domain entities to constraints of only one property in a set of desired qualities in a system; while ADM sacrifices this capability on the granularity of individual domain entities, it does so in exchange for greater potential flexibility and overall implementation sustainability by allowing domain logic to be implemented in a specific class (and exposed through the interface). My personal opinion distributes logic all over the place, not making a product that can be maintained properly. However, this is very specific implementation. Using Akka with Persistence, the responsibility of persistence is delegated to the Akka runtime and therefore transparent for implementation. The actor handles the message, which makes the test ability better, and encapsulates the best/loose connector I've seen in process time. Handling these messages with event handlers can still create software that can be properly tested by modulateing e.g. code. use facades or services. Akka makes the Rich Data Model feasible and usable anywhere. Benefits of Removing the separation between logic and data; (Procedural programming). Every procedure operates on data. Works well for simple applications. Generates stateless logic, which facilitates Scaling Out Avoiding the need for complex OO-Database mapping layers. It follows the principle of Sole Responsibility giving the class no more than one reason to change (change data). Liability logic cannot be implemented in a truly object-oriented way, because logic and data are separated. Violation of the principle of encapsulation and concealing information (objects operate based on their circumstances through methods/messages) Requires a separate business layer to contain the logic located in the domain model. It also means that the domain model object cannot guarantee its correctness at any time, since validation and mutation logic are placed somewhere outside (most likely in some places). Requires a service layer when sharing domain logic in consumers that is different from the object model (domain client, on-premises or remote, communicating with the domain through the service) Makes the model less expressive. SOLID Object-Oriented Design Principles [SOLID principles of Object-Oriented design](object-oriented_design). SOLID principles are guidelines that seek to balance simplicity of implementation, scalability, and durability. Mark Nijhof has written a great book on SOLID principles, and is available for free so go and get it SOLID – Software Development instead of Jenga Games – Mark Nijhof. The first five principles are class design principles. They are: Single Reponsibility Principle: The class must have one, and one, the reason for the change. => There should never be more than one reason for the class to change. Open Closed Principle: You should be able to expand class behavior, without modifying it => => entities, (classes, modules, functions etc.), must be open for extensions to be closed for modification. Liskov Substitution Principles: Lowered classes should be substituted for their base class. => Functions that use object references to basic classes should be able to use objects from a derived class without realizing it. Principle of Interface Separation: Create a smooth graying interface that is client-specific. => Clients should not be forced to rely on interfaces they do not use. Dependency Inversion Principle: Depends on the alphabet, not the conclusion. => High-level modules should not depend on low-level modules. Both have to rely on abstraction. Abstraction should not depend on details. Details should depend on abstraction. The next six principles are about the package. In this context, packages are binary that can be sent such as .jar files, or etc. as opposed to namespaces such as java packages or C++ namespaces. The first three package principles are about package cohesion, they tell us what to put in the package: REP: Equivalence Principle of Re-Use Release: The re-use granules are release details. CCP: General Closure Principle: Classes that change together are packaged together. CRP: General Re-Use Principle: Classes used together are packaged together. The last three principles are about inter-package connectors, and talk about metrics that evaluate the structure of system packages. ADP: Acyclic Dependency Principles: Package dependency graphs must not have cycles. SDP: Principle of Stable Dependency: Depends on the direction of stability. SAP: Principle of Stable Abstraction: Abstraction increases with stability. Manage Dependency Dependency Management is a problem most of us face. Every time we bring up on our screens a bunch of nasty tangled inheritance codes, we experience the result of poor dependency management. Poor dependency management leads to hard-to-change, fragile, and reusable code. Indeed, I am talking about some of the different design smells in ppp books, all related to dependency management. On the other hand, when dependencies are well managed, the code remains flexible, powerful, and reusable. So dependency management, and therefore these principles, are at the foudation -ilities that software developers want. Solid principles: Helps manage dependency, Improved sustainability, flexibility, resilience, and reusability. SOLID Resource Transaction script pattern Sets business logic based on the procedure by which each procedure handles a single request from the presentation. Most business applications can be considered a series of transactions. Transactions can see some information arranged in a certain way, others will make changes to it. Each interaction between the client system and the server system contains a certain amount Logic. In some cases it can be as simple as displaying information in a database. In others it may involve many steps of validation and calculation. The Transaction Script sets all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken down into subprojects. In his book Enterprise Application Patterns, Fowler notes that transaction script patterns are OK for many simple business applications, and avoids the need for complex OO database mapping layers. Serverless architecture Words without servers are a bit of a misnomer. Whether you use compute services like AWS Lambda to run your code or interact with APIs, there are still servers running in the background. The difference is that this server is hidden from us. There's no infrastructure for us to think about. There is no way to change the underlying operating system. Others take care of the nitty-gritty details of infrastructure management, freeing up our time for other things. Serverless is about running code in computing services and interacting with services and APIs to get the job done. There are 5 serverless architectural principles that describe how an ideal serverless system should be built. Use these principles to help guide your decisions when you create serverless architectures. Use compute services to run code on demand (serverless), Write single purpose stateless functions (Single Responsibility Principles (SRP)- SOLID design i.e. Nano-Compute), Push-based, event-based pipe design, Create thicker and more powerful front ends, Embrace third-party services There are two well-known architectures when going serverless, calculated as the back end which is the back end for web and mobile applications and computing as glue which is a pipeline built to perform workflows. Both architectures are free and chances are you will build and combine these architectures if you end up working on any kind of system without real world servers. The architecture of computing as a back-end explains the approach in which serverless computing services such as Lambda and third-party services are used to build back end for web, mobile, and desktop applications. A good serverless system tries to minimize the scope and footprint of Lambda functions so that these functions only perform a bare minimum (Nano-Compute) and mainly focus on tasks that should not be performed at the front end due to privacy or security concerns. Computing as a glue architecture illustrates the idea that we can use Lambda functions to create execution channels and workflows Strong. This often involves using Lambda as glue between different services, coordinating, and calling them. With this architectural style, the developer's main focus is on design coordination, and data flow. The parallelism of serverless computing services like Lambda helps make this architecture interesting. Nano-Compute According to Paul Johnston, not all use cases are suitable for walking in Lambda (Nano Compute). Let's hear him. When using AWS Lambda and similar, you're basically running nano-containers with a single function. There are startup costs in terms of time and load, and there's also the fact you don't know how fast the solution will run compared to, say, a piece of hardware you have. Once loaded Lambda will warm up and so will work faster, so that's okay, but you still don't have the guaranteed availability of resources in the same way when you have a server. So you have problems with response speed, but not reliability or even scalability. The whole idea of a single function is the lack of dependencies, which makes it faster, but that doesn't mean it should be easier to develop or give you an easy way of expanding the function to do more (you may need other functions!). Basically, a small amount of computing is great for a small amount of load, but for something larger, or running longer it might be a little more complex. With Lambda, you have a maximum of 5 minutes, which is very useful, but the problem is that for long-running tasks you have to save data and restart new functions. Basically, divide the task into several Lambdas. This means that any large-scale data task or long-running processing task may be better than a Serverless infrastructure, or at least, developed in such a way as to fit (which can be done). Serverless is primarily an asynchronous solution although inherently out of sync. Therefore you should approach most tasks as things you should consume, but not necessarily respond to. As long as you have a mechanism to respond (push) then the user's application/mobile/web solution doesn't have to worry about anything. The exception is when you need a real-time (or almost real-time) solution. In this scenario you have to be confident with your services and solutions to be fault-tolerant but to be honest, the best solution is probably the third party that provides this for you rather than developing your own serverless nano computing solution. If you need to do something complex with a computer program, then it may not be appropriate. The reference implementation for the Lambda function is image processing, but imagine it is a video processing function. It may need more memory and computing than can be collected from Lambda's computing functions, so this may be problematic. The problem is memory and computing are available, rather than whether the language can handle complexity itself. This is a call per function rather than an architectural call. If you can disassemble complexity into some of the Maybe it's okay. I would suggest that there are several times that Serverless is not appropriate: Whenever you need real-time importance at the business level (you can do asynchronous), Whenever you need a long-term task that cannot be divided into multiple compute cycles, Whenever you need complex computing with high memory/computational requirements, whenever you need excellent response speed Whenever you need reliability where you cannot guarantee service availability to a high enough level. Serverless Data Processing Common uses for serverless technology are data processing, conversion, manipulation, and transcoding. Lambda functions can be used to process CSV, JSON, and XML files; collage and data aggregation; image resizing peng; and format conversion. Lambda and AWS services are perfect for building event-based pipelines for data processing tasks: many of them have been designed to integrate and run simultaneously (for example, pushing files into S3 buckets and letting Lambda functions run automatically and change files). Data Processing is an excellent use case for serverless technology, especially when you use Lambda functions in concert with other services. Serverless Real Time Analysis data—such as logs, system events, transactions, or user clicks—can be achieved using services such as Amazon Kinesis. The Lambda function can react to new records in streams and processes, store, or discard data quickly. The Lambda function can be configured to run when there is a certain number (batch size) of records available for processing, so it doesn't have to be executed for each individual record added to the stream. Kinesis flow and Lambda functions are suitable for applications that produce a lot of data that needs to be analyzed, collected, and stored. When it comes to Kinesis, the number of functions launched to process messages from the stream is equal to the number of fractions (therefore, there is one Lambda function per shard). Furthermore, if the Lambda function fails to process the batch it will try again. It can continue running for up to 24 hours (i.e. how long Kinesis will store data before it expires) if processing fails each time. However, even with these small gotchas, the flow combination of Kinesis and Lambda is powerful if you want to do real-time processing and analytics. Serverless Legacy API Wrapper One of the innovative use cases of AWS API Gateway and Lambda is legacy API wrappers. Here, developers use API Gateway and Lambda to create new API layers through old APIs and services to make them easier to use. API Gateway is used to create the RESTful interface, and lambda functions are used change request/response data and marshals to a format that can be understood by the old service. This approach makes legacy services easier to consume for modern clients who may not be protocols and data formats. Lambda Functions Serverless Scheduled Services can run on schedule, which makes them effective for recurring tasks such as data backup, import and export, reminders, and alerts. Lambda functions can run on schedule, for example. to periodically ping websites to see if they are online and send an email or text message otherwise. There is a Lambda blueprint available for this particular use case. A blueprint is a template with sample code to choose from when creating a new Lambda function. Lambda functions can also be scheduled to perform nightly downloads of files from the server and send daily account reports to users. Repetitive tasks like file backup and file validation can also be done easily with Lambda thanks to scheduling capabilities that you can set up and forget. AWS Lambda AWS Lambda is a compute service that runs code written in JavaScript.js Python, Java, or Scala nodes on AWS infrastructure. Source code is used to isolated containers that have their own memory, disk space, and CPU allocations. The combination of your code, configuration, and depedencies, is commonly referred to as the Lambda function. The Lambda runtime can call the function several times in parallel. Lambda supports push and pull event operating models and integrates with a large number of AWS services. Functions can be called by HTTP request through AWS API Gateway, or executed on scheduler. AWS isn't the only vendor providing lambda architecture runtime. Microsoft Azure Functions, IBM Bluemix OpenWhisk, and Google Cloud Functions are other lambda architecture runtime providers (which may or may not be compatible). AWS API Gateway AWS API Gateway is a service that you can use to create API layers between front end and back end services. API Gateway lifecycle management enables multiple API versions to run at the same time and supports multiple release stages such as development, staging, and production. API Gateway also comes with useful features like caching and throttling requests. AWS Simple Notification Service (SNS) AWS Simple Notification Service (SNS) is a scalable sub pub service designed to deliver messages. The producer or publisher creates and sends a message to a topic. Customers or consumers subscribe to topics and receive messages through one of the supported protocols. SNS stores messages on multiple servers and data centers for redundancy and guarantees delivery at least once. At least once a delivery stipulates that a message will be sent at least once to the customer but, on rare occasions due to the nature of the distributed SNS, it may be sent several times. At least once means that the message can be several times that also means that lambda you have to handle the same message processing several times, implying idempotency. idempotency. cases when a message cannot be sent by SNS to an HTTP endpoint, the message can be configured to re-attempt the delivery at a later time. SNS may also re-try failed shipments to Lambda if throttling is applied. SNS supports messaging content up to 256KB. AWS Simple Storage Service (S3) AWS Simple Storage Service (S3) is Amazon's scalable storage solution. Data in S3 is overs stored in some facilities and servers. The event notification system allows S3 to send events to SNS, SQS, or Lambda when an object is created or deleted. S3 is secure by default, with only owners having access to the resources they create; however, it is possible to set more detailed and flexible access permissions using Access Control Lists (ACLs) and bucket policies. S3 uses bucket and object concepts. Buckets are high-level directories or containers for objects. An object is a combination of data, metadata, and keys. A key is a unique identifier for an object in a bucket. S3 also supports the concept of folders as a means of grouping objects in the S3 console. Folders work by using key name presets. The slash character / in the key name describes the folder. For example, an object with the name of the document key/personal/myfile.txt is represented as a folder called a document that contains a folder called a private one that contains the myfile file.txt in the S3 console. AWS Simple Queue Service (SQS) AWS Simple Queue Service (SQS) is a distributed and error-tolerant Amazon queue service. This ensures message delivery at least once similar to SNS and supports message load up to 256KB. SQS allows multiple publishers and consumers to interact with the same queue, and has a built-in message lifecycle that automatically expires and deletes messages after a pre-defined retention period. Like most AWS products, there are access controls to help control access to queues. SQS integrates with SNS to automatically receive and queue messages. AWS Simple Email Service (SES) AWS Simple Email Service (SES) is a service designed to send and receive email. SES handles email receiving operations such as spam and virus scanning, and email rejection from un trusted sources. Incoming emails can be sent to S3 buckets, used to call Lambda notifications, or create SNS notifications. This can be configured as part of a receipt rule, which tells the SES what to do with the email once it arrives. Sending email with SES is very easy but there are limits, which exist to set the rate and number of messages sent. SES will automatically increase quotas as long as high-quality emails, and not spam, are being sent. AWS Relational Database Service (RDS) AWS Database Service (RDS) is a web service that assists in setting up and operating relational databases within the AWS infrastructure. RDS RDS support Aurora, MySQL, MariaDB, Oracle, MS-SQL, and PostgreSQL database engines. It takes care of routine tasks such as provisioning, backup, patching, recovery, repair, and failure detection. Monitoring and metrics, database snapshots, and some Availability Zone (AZ) support are provided out of the box. RDS uses SNS to communicate notifications when an event occurs. This makes it easy to respond to database events such as creation, deletion, failover, recovery, and recovery when it occurs. AWS DynamoDB AWS DynamoDB is Amazon's NoSQL solution. Tables, items, and attributes are dynamo's main concepts. The table stores a set of items. The item consists of a set of attributes. Each attribute is a simple piece of data such as a person's name or phone number. Each item can be uniquely identified. Lambda integrates with DynamoDB tables and can be triggered by table updates. AWS CloudSearch AWS CloudSearch is a search solution from AWS that supports structured data and plain text. CloudSearch takes a snippet of the data as JSON or XML and generates a queried index. The service supports boolean search, presets, and range, full-text search, faceting, highlighting, and autocomplete. Each document provided to CloudSearch is provided with an ID, generated by the user, that makes the document uniquely identifiable. Search requests can be performed using GET requests. Results can be returned as JSON or XML, and can be sorted, cited, and include useful metadata such as relevance scores. AWS CloudFront AWS CloudFront is a web service that accelerates the distribution of your static and dynamic web content, for example, .html, .css, .php, images, and media, to end users. AWS Route 53 AWS Route 53 is a highly available and scalable Domain Name System (DNS) web service. AWS Virtual Private Cloud (VPC) AWS Virtual Private Cloud (Amazon VPC) enables you to launch Amazon Web Services (AWS) resources into a virtual network that you have defined. These virtual networks are very much like traditional networks that you'll operate in your own data centers, with the benefit of using scalable AWS infrastructure. AWS Elastic Transcoder AWS Elastic Transcoder is an AWS service for transcoding media to other formats, resolutions, and bitrates. This service is useful if you need to have a media version that can be played on different devices. Elastic Transcoder comes with a number of presets or templates that specify how the video should be transcoded, but you can specify your own. It integrates with S3 and SNS, which are used for notifications when work is completed or error conditions are raised. Elastic Transcoder also has additional features such as watermarking, transcoding and DRM support. AWS Kinesis AWS Kinesis is a service for real-time processing of big data streaming. It is usually used for quick logs and data intake, data, analytics, and reporting. This is different from SQS where Amazon recommends that Kinesis Streams be used primarily for big data streaming, whereas SQS is used as a reliable hosted queue especially if finer controls over messages, such as visibility deadlines or individual delays, are required. In AWS Kinesis, the shard determines the throughput capacity of a stream. The number of fractions needs to be set when the flow is created; however, resharding is possible if throughput needs to be increased or reduced. By comparison, SQS makes scaling much more transparent. Lambda can integrate with Kinesis to read batches of recordings from the stream as soon as it is detected. AWS Internet of Things (IoT) AWS IoT enables secure two-way communication between Things connected to the Internet (such as sensors, actuators, embedded devices, or smart appliances) and the AWS cloud through MQTT and HTTP. AWS IoT combines the following: authentication and authorization, communication gateways, registry (a way to assign a unique identity to each device), device shadow (persistent device state), rule engine (a service for changing and merute device messages to AWS services). The rules engine, for example, can save files to Amazon's Simple Storage Service (S3), push data into Amazon Simple Queue Service (SQS) queues, and invoke AWS Lambda functions. Amazon's IoT platform makes it easy to build scalable IoT back ends for devices without having to run servers. AWS Cognito AWS Cognito is an identity management service. It integrates with public identity providers like Google, Facebook, Twitter, and Amazon, or with your own system. Cognito supports syncing users' app data across multiple devices and has offline support that allows mobile devices to work even when there is no Internet access. Auth0 Auth0 is a non-AWS identity management product that has some features that Cognito doesn't use. Auth0 integrates with more than 30 identity providers, including Google, Facebook, Twitter, Amazon, LinkedIn, and Windows Live. It provides a way to register new users, through the use of its own user database, without having to integrate with an identity provider. In addition, it has the facility to import users from other databases. As expected, Auth0 supports standard industry protocols including SAML, OpenID Connect, OAuth 2.0, OAuth 1.0, and JSON Web Tokens. It's easy to integrate with AWS Identity & Access Management, and with Cognito. GraphQL GraphQL is a popular data query language developed by Facebook in 2012 and released publicly in 2015. It is designed as an alternative to REST (State Transfer representation) due to the weaknesses REST (multiple round trips, over-fetching, and problems with versioning). GraphQL attempts to solve this problem by providing a hierarchical and declarative way to query queries one end point. GraphQL gives power to clients. Instead of specifying the response structure on the server, it is defined on the client. The client can specify what properties and relationships to return. GraphQL combines data from a variety of sources and returns it to the client on a round trip (which makes it an efficient system for retrieving data). According to Facebook, GraphQL serves millions of requests per second from nearly 1000 different versions of their app. In serverless architectures, GraphQL is typically hosted and run from a single Lambda function that can be connected to a GraphQL Gateway API that can request and write to multiple data sources (such as DynamoDB tables) and assemble responses that match the request. Serverless GraphQL is a rather interesting approach that you might want to look at next time you need to design an interface for your APIs and query data. There is a reference implementation available on Github implemented in GraphQL.js JavaScript. There is only one decent GraphQL implementation for Scala available: Sangria. For a list of GraphQL implementations see C. T. Lin's Overview of GraphQl. Youtube Reactive Design Patterns - Stuart Sierra - Functional Design Pattern Design Patterns in Simple Example Scala Best Practices A TypeTag summarizes the runtime type representation of several types of T. TypeTags' main use case is to provide access to the deleted type. There are three different TypeTags: scala.reflect.api.TypeTags#TypeTag: A complete type descriptor of the Scala type. For example, TypeTag[List[String]] contains all type information, in this case, based on the type of scaling. List[String]. scala.reflect.ClassTag: A partial type descriptor of the Scala type. For example, ClassTag[List[String]] contains only deleted class type information, in this case, from type scala.collection.immutable.List. ClassTags provide access only to runtime classes of this type. Analogue to scala.reflect.ClassManifest StackOverflow - Scala: What is TypeTag and how do I use it? Functional Programming The software architecture of computing systems is a series of structures necessary to take root about the system, consisting of software elements, the relationship between them, and the properties of both. The architecture of software has emerged as an important subdisciplinary of software engineering. The architecture is roughly partition wise from the whole into several parts, with a certain relationship between the parts. It is this partition that allows a group of people—often separated by organizational, geographic, and even time zone boundaries—to work cooperatively and productively together to solve a much larger problem than can be solved individually. Each group writes software that with other group software through a carefully crafted interface that reveals the minimal and most stable information needed to From these interactions comes quality functionality and attributes—security, modification, performance, and so on—that system stakeholders are interested in. The larger and complex the system, the more important this partition is — and therefore, the architecture. The more demanding these quality attributes are, the more important the architecture. One system is almost certainly partitioned simultaneously in a number of different ways. Each partition produces the creation of an architectural structure: various sets of sections and different relationships between the parts. Each is the result of careful design, carried out to meet the requirements of driving quality attributes and the most important business objectives behind the system. Many projects make the mistake of trying to force a single partition across multiple component domains, such as equating a thread with an object, which is likened to a module, which in turn is likened to a file. Such an approach never works completely, and eventual adjustments must be made, but the damage of the initial intention is often difficult to repair. This always causes problems in development and sometimes in the final product. Architecture is what makes a set of spare parts work together as a coherent and successful whole. Architectural documentation helps architects make informed decisions; it tells developers how to implement it; and it records decisions to provide insight into the future management of the system into the architect's solutions. For almost all systems, quality attributes such as 'performance', 'reliability', 'security', and 'modifiability' are just as important as ensuring that the software calculates the correct answers. The software system's ability to produce the right results doesn't help if it's doing it too long, or the system doesn't stay up long enough to deliver it, or the system reveals the results to your competition or enemies. Architecture is where these concerns are addressed. For example: If you need high performance, you need to: Exploit potential parallelism by outlining work to work together or synchronize processes. Manage interprocessing and volume of network communication and data access frequencies. Can expect expected delays and throughput. Identify potential performance bottlenecks. If your system requires high accuracy, you should pay attention to how data elements are defined and used and how their values flow throughout the system. If security is important, you need to: Authorize usage relationships and communication restrictions between sections. Identify the parts of the system where unauthorized interference will do the most damage. Perhaps introduce a special element that has gained a high level of trust. If you need to support modifiability and You have to carefully separate the concerns between parts of the system, so when a change affects a single element, it doesn't tear the entire system apart. If you want to spread the system gradually, by releasing larger subsets in a row, you should maintain a dependent relationship between the pieces that do not materialize, to avoid the syndrome of 'nothing works until everything works'. The solution to this concern is pure architecture in nature. It is up to the architect to find the solution and communicate it effectively to those who will implement it. Documentation architecture has three obligations related to quality attributes: First, it must show the requirements of which quality attributes drive the design. Second, it must capture the selected solution to meet the quality attribute requirements. Finally, it should capture a convincing argument as to why the solution provides the necessary quality attributes. The goal is to capture enough information so that the architecture can be analyzed to see if, in fact, the system derived from it will have the necessary attributes of quality. What is Software Architecture? If we agree on what it means to document the software architecture, we should establish the same basis for what we document. There is no universal definition of software architecture that exists. The Software Engineering Institute's website collects definitions from literature and from practitioners around the world; so far, more than 150 definitions have been collected. It seems that the new fields are trying to nail their standard definitions or key terms as soon as possible. As the field matures, the basic concept becomes more important than the ironclad definition, and this impulse seems to fade. When object-oriented development is in its in its in in its in its ins and outs, you can bring the OO meeting to a screeching halt by putting on your best plain face and asking, What exactly is an object? This largely ends when people realize that scattered definition plots have clear centroids (if not articulated), from which very useful advances can be made. Sometimes pretty close is, well, pretty close. This seems to be the case with software architecture. Seeing great efforts to nail down its definition gives us a good look at our own centroids. The centroid takes a largely structural perspective on the architecture of the software: The architecture of the software consists of elements, connections or relationships between them, and, usually, some aspect or other aspect, such as configuration (take your choice); constraints or semantics; analysis or properties; or the reasons, requirements, or needs of stakeholders. This perspective does not preclude each other, nor does it represent a fundamental conflict what is software architecture. Instead, they represent a spectrum within the software architecture community about the emphasis that should be placed on architecture: parts of its constituents, entire entities, ways of behave once built, or building it. Put together, they form a consensus view of software architecture. So we end up with the following definition: the _The software architecture of a computational program or system is the structure or structure of the system, consisting of the elements of the software, the externally visible properties of those elements, and the relationship between them. With 'externally visible properties,'' we refer to assumptions that other components can make, such as services provided, performance characteristics, error handling, shared resource usage, and so on. The architecture definition of the software was previously in accordance with the purpose of knowing what to document about the architecture and thus came up with a Software Architecture Document (SAD) that could be used by several stakeholders. This definition emphasizes the plurality of structures that exist in each software system. This structure, carefully chosen and designed by architects, is key to achieving and reasoning about the design objectives of the system. And those structures are the key to understanding architecture. Therefore, they are the focus of approaches to documenting software architecture. The structure consists of elements, the relationship between the elements, and the essential properties of both. So documenting structures requires documenting those things. What is the Difference Between Architecture and Design? The question of how architecture differs from design has been biting the heels of the software development community for years. This is a question I often hear when teaching introductory courses on architecture. It is important here because the question relates to what we should put in the architectural document and what we should put elsewhere. The first thing we can say is that obviously architecture is design, but not all designs are architecture. That is, many design decisions are not bound by architecture and are happily left to the wisdom and good judgment of downstream designers and even executors. Architecture sets limits on downstream activities, and those activities must produce artifacts—finer designs and code—that fit the architecture. It's tempting to stop there, but if you notice you've noticed that we've just translated the question: Architecture consists of architectural design decisions, and others are nonarchistektural. So what decisions are nonarchitectural? That is, what design decisions do architects make at the discretion of others? To answer this question, we return to the main purpose of architecture, namely to ensure the satisfaction of the quality and behavioral requirements of the system business objectives. Architects do this by making design decisions that manifest themselves in the architectural structure of the system. Thus, architectural architecture is a system that allows the system to meet its quality attributes and behavioral requirements. All other decisions are nonarchistektural. Clearly any design decision that results in the nature of the element is invisible – that is, makes no difference beyond the element – nonarchistektural. A common example is the selection of data structures, along with algorithms for managing and accessing those data structures. You may have been hoping for more concrete answers, such as 'the first three levels of module decomposition are architecture, but every subsequent decomposition is not.' Or, 'classes, packages and their relationships in UML class charts are architectures, but sequence diagrams are not.' Or 'defining a SOA system service is architecture, but designing the internal structure of every component of a service provider is not.' But they don't work because they attract arbitrariness and such efforts become practical eventually becoming impractical because true architecture bleeds on those boundaries. Here are some of the sometimes audible artificial definitions: First, 'architecture is a small set of big design decisions.' Some people define a 'small set' by insisting that architectural documents should not be more than 50 pages long. Or 80. Or 30. Their feeling, it seems, is that architecture is a series of design decisions that you can squeeze into a certain page quota, and everything beyond that is not. This is, of course, nonsense. Another nonanswer that sounds is 'architecture is what you get before you start adding details to the design.' Terminology often directs our thinking, rather than presenting it. A sinister example that puts us in the wrong set of minds is 'detailed design.' Detailed design is what many people say follows architecture. The term is everywhere, and needs to be stamped out. This implies that the difference between architectural design and nonarsitektur is something called 'detail.' Architecture is apparently not allowed to be detailed, because if it is, well, you do detailed design then, right? Never mind that we don't know how to measure 'detail' or set thresholds when there is too much to be architectural. If your design starts to look 'detailed' then you are not doing architecture and you will be reported to the Detailed Design Police to exceed your authority. More. It is true that some architectural design decisions may not have much specificity; that is, they preserve the freedom of choosing downstream designers. Some architectural design decisions may not be 'decisions' at all, but constraints are widespread. Plug-ins that populate your Web browser are examples. No architecture nailing a complete set, but the architecture limits the new meet certain standards and interfaces. Or the architect might describe the element by saying,' ' delivering its computational results through this published interface, securely threads, placing no more than three messages on the network per call, and returning the answer in less than 20 ms.' The team that applied the element were free to make whatever design decisions they wanted as long as they fulfilled the architect's recipe for it. On the other hand, some architectural decisions can be very 'detailed,' such as the adoption of certain protocols, XML schema, or communication or technology standards. These decisions are usually made for interoperability purposes or various senses of modifiability (such as scalability or extensibility). Even interface elements, which some decry as 'obvious' outside the realm of architecture, can be very architectural. For example, in a service-oriented architecture (SOA), components interact through a published interface. Important design decisions made when defining these interfaces include operating granularity, data formats, and interaction types (synchronous or asynchronous) for each operation. Or consider elements that process data from real-time sensors. Making the interface process of these elements as streams as opposed to individual data elements will make a big difference in the ability of the element (and hence the system) to meet real-time performance requirements. This decision cannot be left to the element development team; It's up to him. Legitimate questions about detail do arise when considering modules and other hierarchical elements (Hierarchical elements are any kind of element that can consist of similar elements. Modules are hierarchical elements because modules consist of submodules, which are the modules themselves. A task or process is not a hierarchical element). When did you stop? When have you designed enough levels in the hierarchy? Is submodule enough, or do architects need to design sub-sub-submodules? Here's a good test of our claims when architecture stops. Module decomposition is about achieving independent development and modification. Both are achieved by carefully establishing coherent responsibilities for each module. When the modules you design are refined enough to meet the system's independent modifiability and development requirements, you have relinquished your obligations as an architect. Finally, what architecture is sensitive to context. Suppose the architect identifies the element but is satisfied sketching the interface and behavior of the element broadly. If the prescribed element is very large and complex, the team that developed it can choose to give it its own internal substructure, which for all the world looks like architecture. And in the context of that element, it is. But in the context of the overall system, the substructure architecture but only internal design decisions are made by the development team for that element. To summarize, architecture is design, but not all designs are architecture. Architects draw a line between architectural and nonarchural design by making decisions that need to be bound in order for the system to meet its development goals, behavior and quality. All other decisions can be left to downstream designers and implementers. Architectural decisions or not, in context. If a structure is important to achieving your system goals, that structure is architecture. But the design of the elements, or subsystems, that you set out may have to introduce their own structures to meet their purposes, in which case such structures are architectural: for them but not for you. Book Documenting Software Architecture - Landscapes and So on - Paul Clements et al Architecture Software For Developers - Simon Brown (2016) Architecture Software book for Developers is great! It declassified the Software Architecture by continuing to ask questions and where the right definitions put in place when no clear answers could be found based on common sense. Simon Brown did a great job cleaning up the fog of war that existed in software development. This is a must read for everyone who makes the software. Now let's give Simon Brown a lot of money, buy a book and go read! Below are my notes and excerpts from the book: What is software architecture? Software architect is a role, preferably shared and fulfilled by the entire development team, if possible, When the role of architect is fulfilled by one person, the approach should be coding, coaching and collaborative design, Software architecture is about structure and vision, Software architecture is about the high level structure of the software system and how one understands it , Software architecture is about significant decisions that affect the form of software systems , software architecture is about things that are difficult or expensive to change, When a team does not consider the architecture of the software, bad things tend to happen; one can end up with an internal software system that is unstructured and inconsistent that is elusive, difficult to maintain and potentially does not meet one or more important non-functional requirements such as performance, scalability or security. One should explicitly think about the architecture of the software, it provides a way to introduce technical leadership and accumulate successful delivery opportunities in your favor, Software architecture is not about the great design up front, My note about Visualizing Software Architecture The Art of Visualizing Software Architecture teaches methods of communicating software architecture visually which is the static structure of the software system. Below are my notes and quotes from from Shared Vocabulary First of all, it is very important to share the vocabulary that you will use in the diagram. We will use a vocabulary consisting of containers, components and classes/codes. With this shared vocabulary we can draw diagrams at different levels of abstraction. The software system is a simple hierarchy of building blocks, The software system consists of one or more containers each containing one or more components which in turn are implemented by one or more classes, Containers can be web applications, mobile applications, standalone applications, databases, file systems, Microservices, shell scripts, etc., Component A is a grouping or related function that is encapsulated behind a well-defined interface, Components will be implemented by one or more classes so that the component contains one or more classes, In OO: the component consists of a class and interface, In FP: The component can be a module that is a logical grouping of related functions, types, etc., In JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of related functions, types, etc., in JS: A component can be a module that is a logical grouping of : Components can be modules that are logical groupings of related functions, types, etc., In JS: Components can be modules that are logical groupings of related functions, type, etc., In JS: A component can be a module that is a logical grouping of related functions, types, etc., In JS: A component can be a module that is a logical grouping of related functions, types, etc., In JS: A component can be a module that is a logical grouping of related functions, types, etc., consisting of a number of objects and functions, In RDBMS: Components can be a logical grouping of functions; based on a number of tables, views, stored procedures, functions, triggers, etc., Context Diagrams – System People and Software Context diagrams are useful for getting the big picture. It doesn't show much detail but helps set the scene and is a starting point for other charts. The diagram should show how the software system fits into the overall environment. All software systems must have a context diagram. Context diagrams are usually the starting point for creating diagrams and documenting software systems. Context diagrams are often drawn during requirements gathering workshops, to ensure that everyone understands the scope of what will be built. The focus should be on people (actors, roles, personas, etc.) and software systems rather than technology, protocols, and other low-level details. This is a diagram that you will show to non-technical people. The diagram should display people's systems and software. Context diagrams help answer the question: What is the system the software we built? Who used it? How does it fit into the existing environment? The context diagram shows the following: It makes the context and scope explicit, so there are no assumptions, It shows what is added to the existing environment, It is a high-level diagram that technical and non-technical people can use as a starting point for discussion, It provides a starting point for identifying who you potentially need to go and talk as far as understanding the interface between the systems in question. Container Diagrams - High-Level Technology Options Container diagrams illustrate high-level technology options. It shows the high level of form of software architecture (static (static and how responsibility is distributed on it. It shows the main technological options and how containers communicate with each other. This is a simple diagram focusing on technology that is useful for software developers and support/operations staff. Container diagrams help answer the question: What is the form of excessive software systems? What are the high-tech decisions? How are responsibilities distributed throughout the system? How do containers communicate with each other? As a developer, where do I write code to implement features? CAP Theorem In theoretical computer science, The CAP theorem, also named after Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for distributed computer systems to simultaneously provide the following three guarantees: 1: Consistency: Each read receives the latest writing or error 2: Availability: Each request receives a response, with no guarantee that it contains the latest version of information 3: Partition tolerance: The system continues to operate despite the number of messages dropped by the network between nodes No distributed system is safe from network failure, so network partitions generally have to be tolerated. In front of the partition, one is then left with two options: consistency or availability. Choose consistency over availability: When choosing consistency over availability, the system returns an error or timeout if certain information cannot be guaranteed to be updated due to network partitioning. Choose availability over consistency: When choosing availability over consistency, the system will always process the query and try to restore the latest available version of information, even if it cannot guarantee it is up to date due to network partitions. In the absence of network failures, that is, when distributed systems run normally, availability and consistency can be met (in the case of multiple database systems, a level of consistency can be requested with the query). CAP is often misunderstood as if one should choose to leave one of three guarantees at any time. In fact, the choice is really between consistency and availability when partitioning occurs only; at all other times, no trade-offs should be made. Database systems designed with traditional ASAM assurance (Atomity, Consistency, Isolation, Durability) in mind such as RDBMS choose consistency over availability, while systems designed around the BASE philosophy (Eventual Consistency), common in the NoSQL movement for example, choose availability over consistency. PaceLC Theorema is built on CAP by stating that even in the absence of partitions, other trade-offs between latency and consistency occur. PACELC Theorem In theoretical computer science, is the extension for caporema. It states that in the case of network partitions (P) in distributed computer systems, one must choose between availability (A) and consistency (C) (cap intensity), but others (E), even when the system is running normally in the absence of partitions, one must choose between latency (L) and consistency (C). PACELC was built on caporema. Both theorems describe how distributed databases have restrictions and tradeoffs regarding partition consistency, availability, and tolerance. But PACELC goes further and states that trade-offs also exist, this time between latency and consistency, even in the absence of partitions, thus providing a fuller picture of the potential for consistency tradeoffs for distributed systems. High availability requirements imply that the system must replicate data. As soon as the distributed system replicates the data, a tradeoff between consistency and latency appears. Appear.