Software design

Low Coupling

Modules should be as independent as possible from other modules, so that changes to a module don't heavily impact other modules.

High coupling would mean that your module knows the way too much about the inner workings of other modules. Modules that know too much about other modules make changes hard to coordinate and make modules brittle.

By aiming for low coupling, you can easily make changes to the internals of modules without worrying about their impact on other modules in the system. **Low coupling also makes it easier to design, write, and test** code since our modules are not interdependent on each other. We also get the benefit of easy to reuse and compose-able modules. Problems are also isolated to small, self-contained units of code.

High Cohesion

We want to design components that are self-contained: independent, and with a single, well-defined purpose.

Cohesion often refers to how the elements of a module belong together. Related code should be close to each other to make it highly cohesive.

By keeping high cohesion within our code, we end up trying DRY code and reduce duplication of knowledge in our modules. We can easily design, write, and test our code since the code for a module is all located together and works together.

Low cohesion would mean that the code that makes up some functionality is spread out all over your code-base. Not only is it hard to discover what code is related to your module, it is difficult to jump between different modules and keep track of all the code in your head.

SOLID

In object-oriented computer programming, SOLID (first introduced by Robert C. Martin in his 2000 paper Design Principles and Design Patterns) is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.

The Single-responsibility principle: "There should never be more than one reason for a class to change. In other words, every class should have only one responsibility."

The Open–closed principle: "Software entities ... should be open for extension, but closed for modification."

The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it" (Design by contract). Parent classes should be easily substituted with their child classes without blowing up the application.

The Interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface."

The Dependency inversion principle: "Depend upon abstractions, not concretions."

Gang of Four Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software (1994) is a software engineering book describing software design patterns. The book is divided into two parts, with the first two chapters exploring the capabilities and pitfalls of object-oriented programming, and the remaining chapters describing 23 classic software design patterns.

GoF Design Patterns are divided into three categories:

- **Creational**: The design patterns that deal with the creation of an object.
- **Structural**: The design patterns in this category deals with the class structure such as Inheritance and Composition.
- **Behavioral**: This type of design patterns provide solutions for the better interaction between objects, how to provide loose coupling, and flexibility to extend easily in future.

Creational Patterns

The design pattern family that deals with the creation of an object.

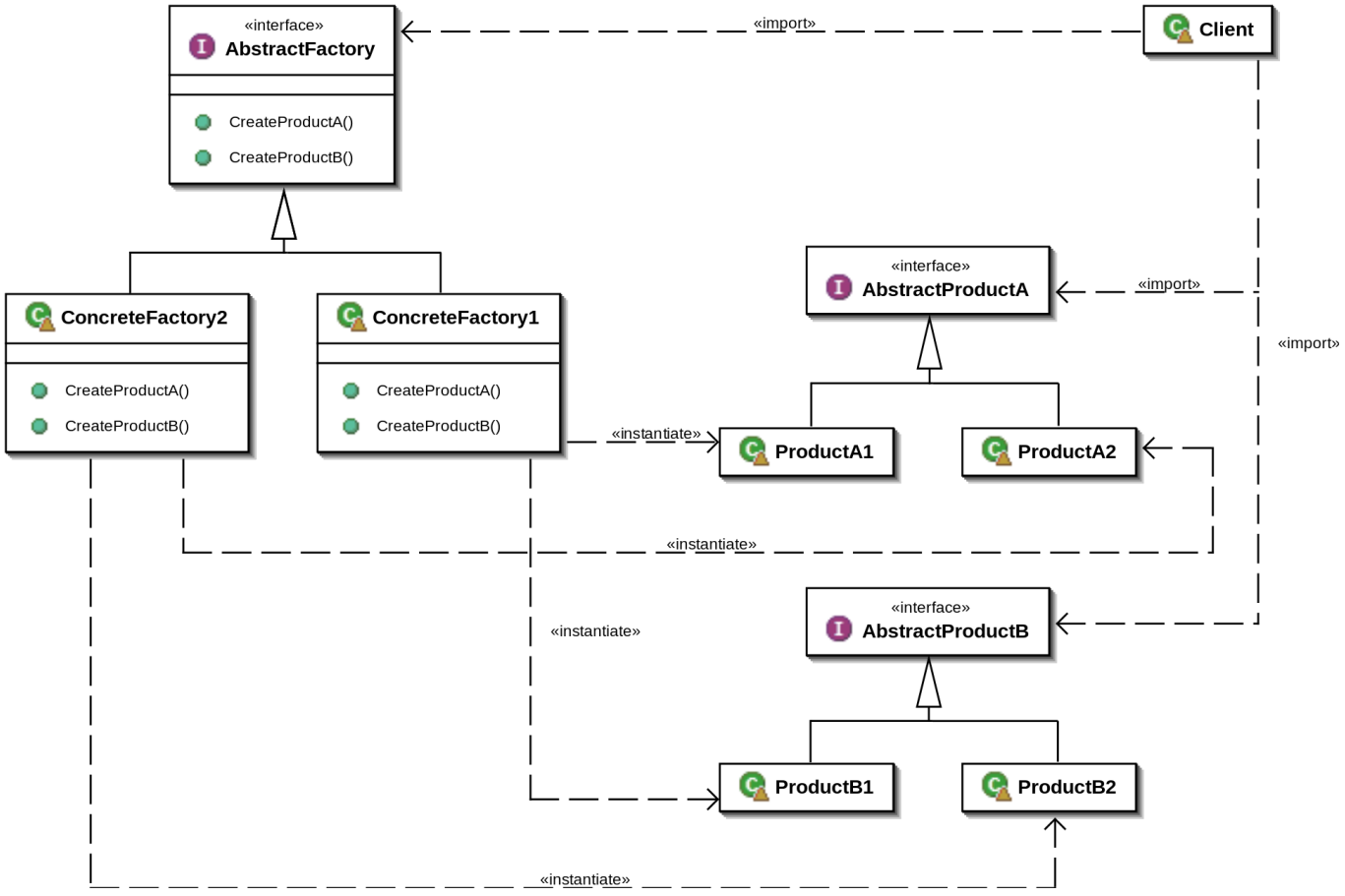
- **Abstract factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory method** creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.

Abstract Factory

Provides an interface for **creating families of related or dependent objects** without specifying their concrete classes.

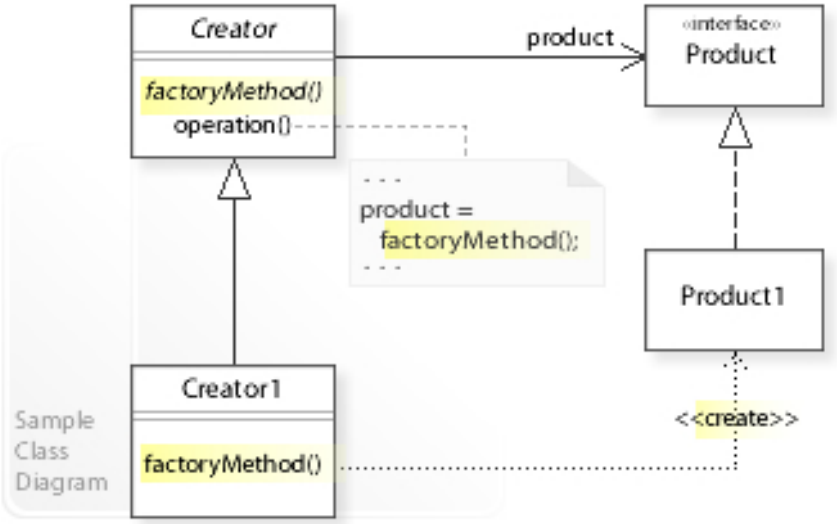
In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client does not know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.

This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.



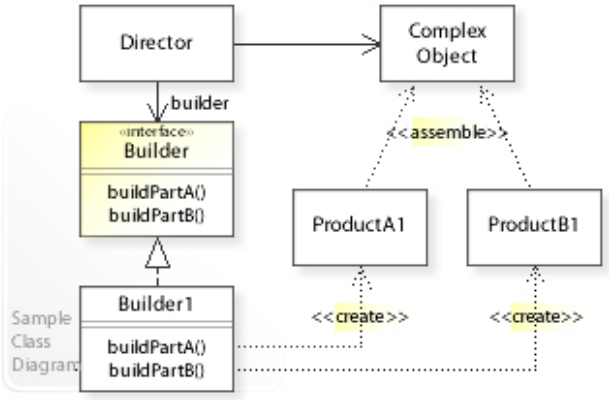
Factory Method

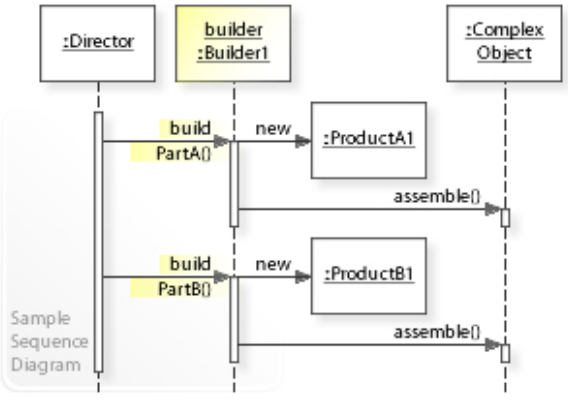
Factory method pattern is a creational pattern that uses factory methods to deal with the problem of **creating objects without having to specify the exact class** of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.



Builder

The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to **separate the construction of a complex object from its representation**.





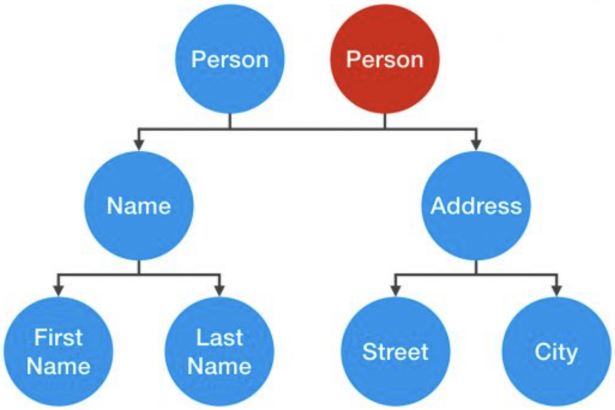
Prototype

It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

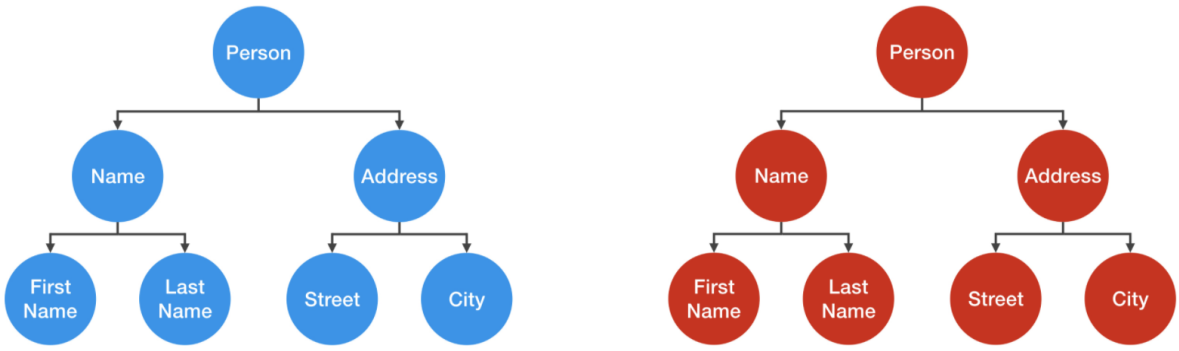
One of the ways we can implement this pattern in Java is by using the clone() method. To do this, we'd implement the Cloneable interface and override Object clone() method.

When we're trying to clone, we should decide between making a shallow or a deep copy.

A **shallow copy** of an object copies the 'main' object, but doesn't copy the inner objects. The 'inner objects' are shared between the original object and its copy. The problem with the shallow copy is that the two objects are not independent. If you modify a non-primitive object referenced by a copy object, the change will be reflected in the 'main' object.



Unlike the shallow copy, a **deep copy** is a fully independent copy of an object. To create a true deep copy, we need to keep copying all of the object's nested elements, until there are only primitive types and "Immutables" left.



Object.clone() - this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned. Thus, this method performs a "**shallow copy**" of this object, not a "deep copy" operation.

Singleton

Singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when **exactly one object is needed** to coordinate actions across the system. The term comes from the mathematical concept of a singleton.

Critics consider the singleton to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.

Example uses:

- The abstract factory, factory method, builder, and prototype patterns can use singletons in their implementation.
- Facade objects are often singletons because only one facade object is required.

The Singleton Pattern is generally considered an anti-pattern for the following reasons:

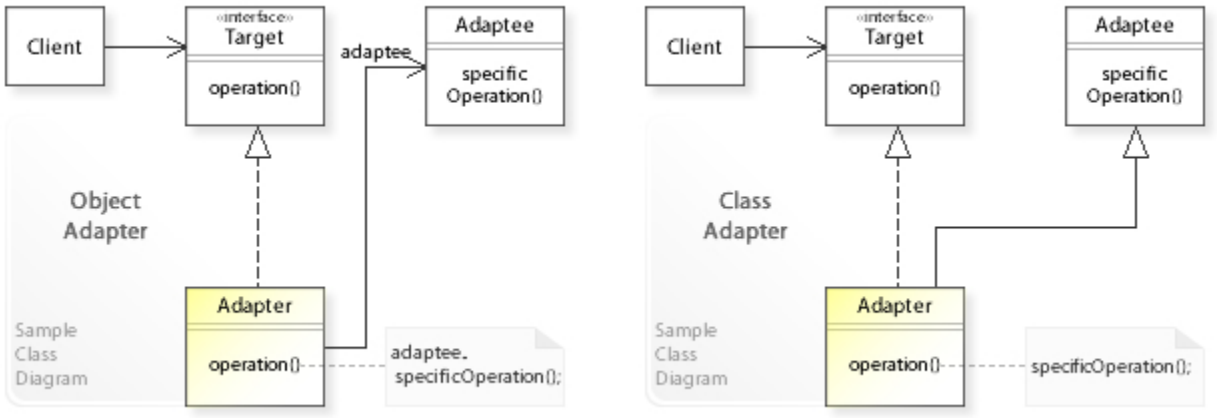
- Singleton classes break object-oriented design principles. It is basically used as global variables
- The use of Singletons makes it difficult to unit test classes because classes must be loosely coupled, allowing them to be tested individually
- Singletons promote tight coupling between classes. Singletons tightly couple the code to the exact object type and remove the scope of polymorphism
- In a garbage collected system, Singletons can become very tricky in regard to memory management

Structural Patterns

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

Adapter

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class. It is often used to make existing classes work with others without modifying their source code.



The adapter design pattern describes how to solve such problems:

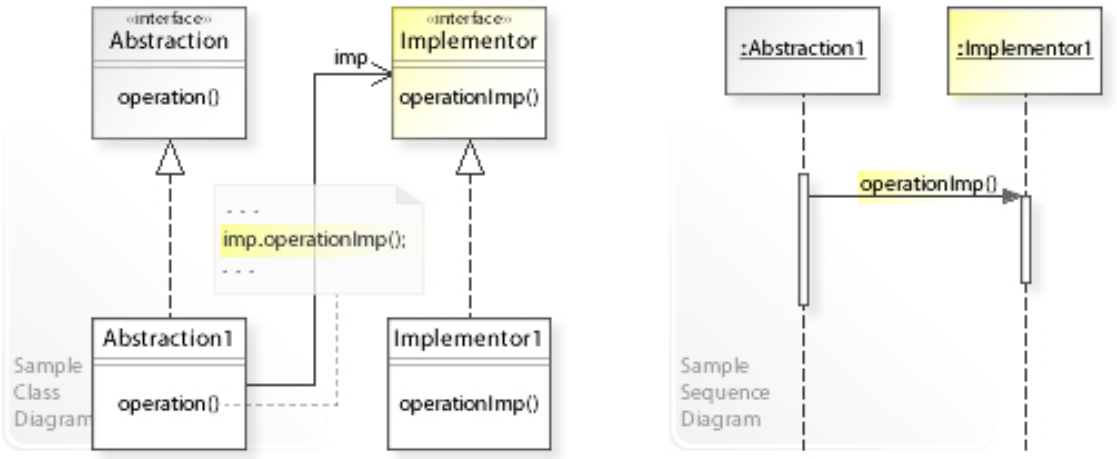
- Define a separate adapter class that converts the (incompatible) interface of a class (adaptee) into another interface (target) clients require.
- Work through an adapter to work with (reuse) classes that do not have the required interface.

The key idea in this pattern is to work through a separate adapter that adapts the interface of an (already existing) class without changing it.

Clients don't know whether they work with a target class directly or through an adapter with a class that does not have the target interface.

Bridge

The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently".



What problems can the Bridge design pattern solve:

- An abstraction and its implementation should be defined and extended independently from each other.
- A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

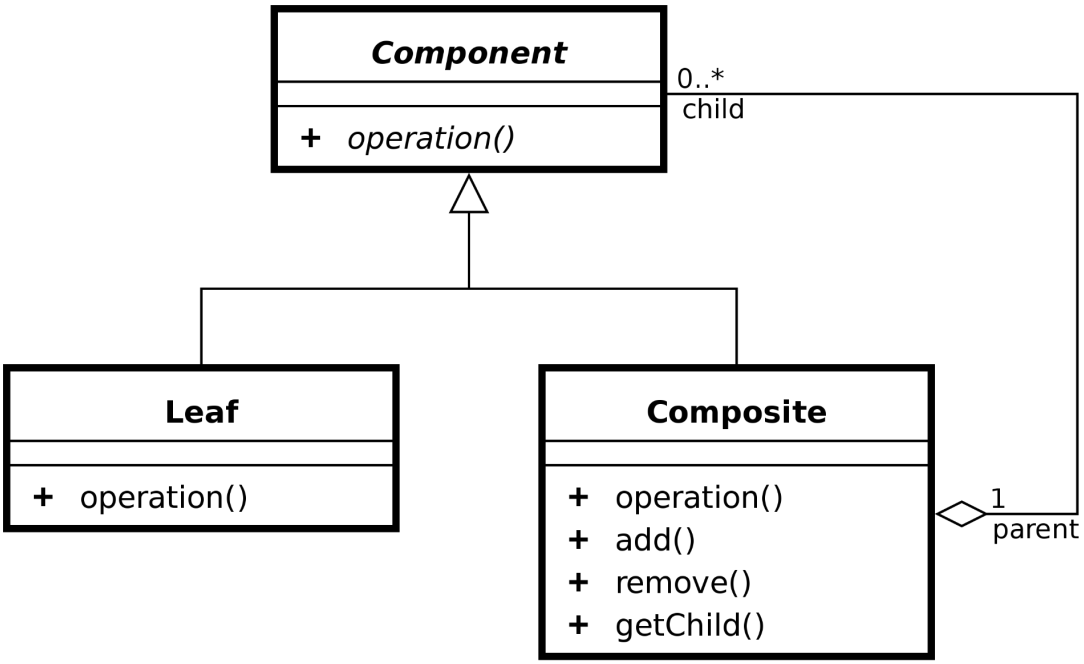
When using subclassing, different subclasses implement an abstract class in different ways. **But an implementation is bound to the abstraction at compile-time and cannot be changed at run-time**.

What solution does the Bridge design pattern describe:

- Separate an abstraction (Abstraction) from its implementation (Implementor) by putting them in separate class hierarchies.
- Implement the Abstraction in terms of (by delegating to) an Implementor object.

Composite

In software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern **lets clients treat individual objects and compositions uniformly**.



What problems can the Composite design pattern solve?

- A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
- A part-whole hierarchy should be represented as tree structure.

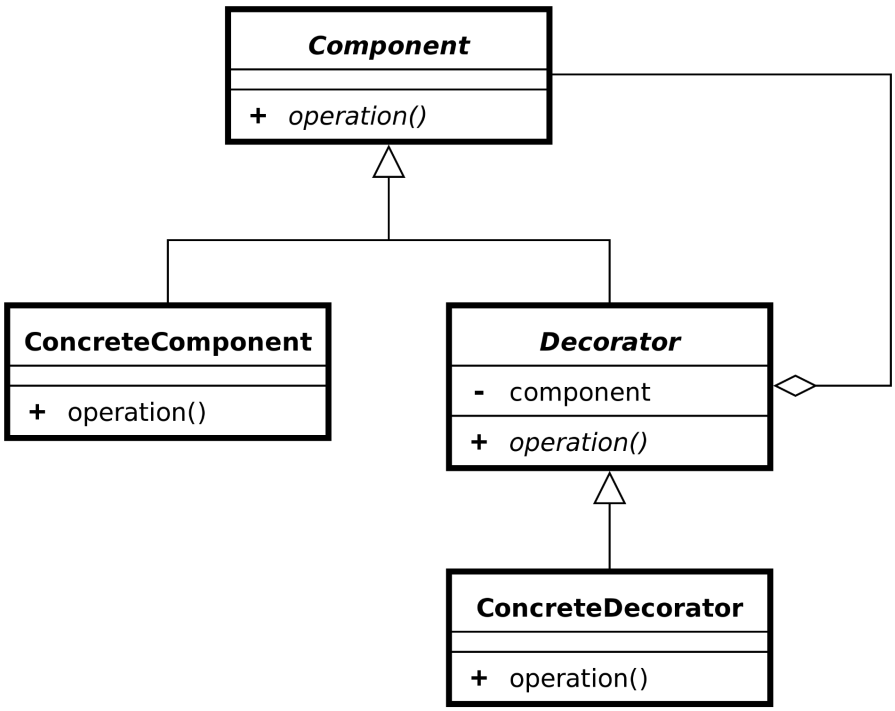
What solution does the Composite design pattern describe?

- Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects.
- Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

Decorator

In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern. Decorator use can be more efficient than subclassing, because an object's behavior can be augmented without defining an entirely new object.



What problems can it solve?

- Responsibilities should be added to (and removed from) an object dynamically at run-time.
- A flexible alternative to subclassing for extending functionality should be provided.
- When using subclassing, different subclasses extend a class in different ways. But an extension is bound to the class at compile-time and can't be changed at run-time.[citation needed]

What solution does it describe?

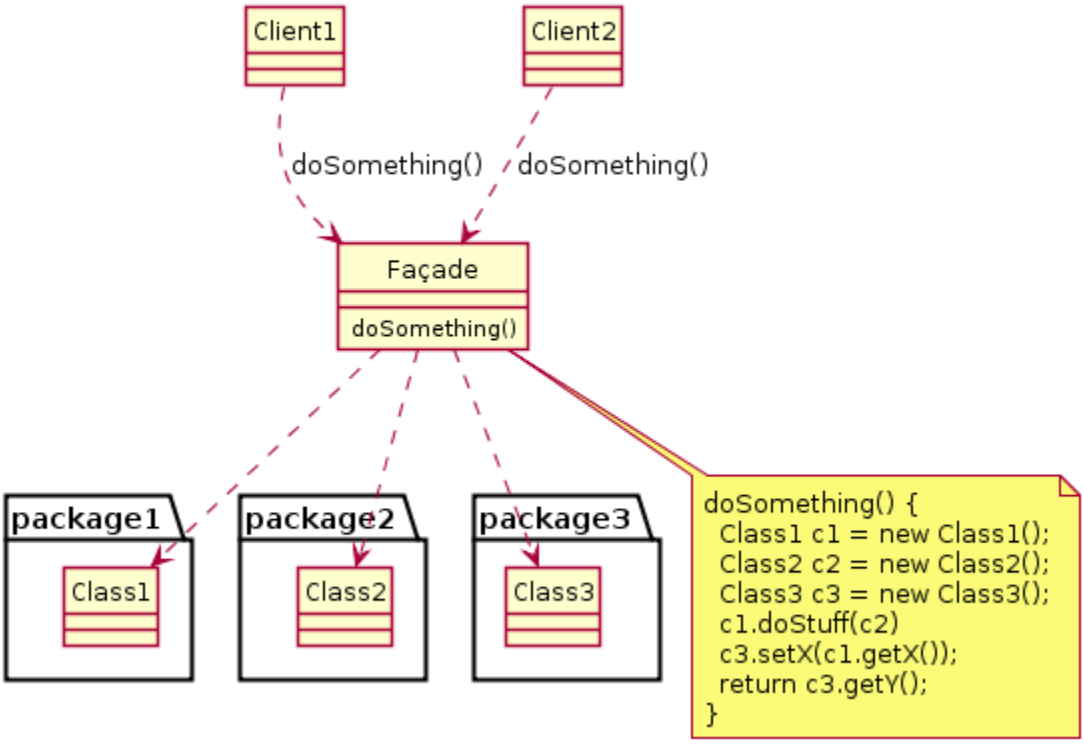
- implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it
- perform additional functionality before/after forwarding a request.

Examples: Java IO (InputStream, OutputStream etc)

Facade

The facade pattern (also spelled façade) is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a **front-facing interface masking more complex underlying or structural code**. A facade can:

- **improve the readability and usability** of a software library by masking interaction with more complex components behind a single (and often simplified) API
- **provide a context-specific interface** to more generic functionality (complete with context-specific input validation)
- serve as a **launching point for a broader refactor** of monolithic or tightly-coupled systems in favor of more loosely-coupled code

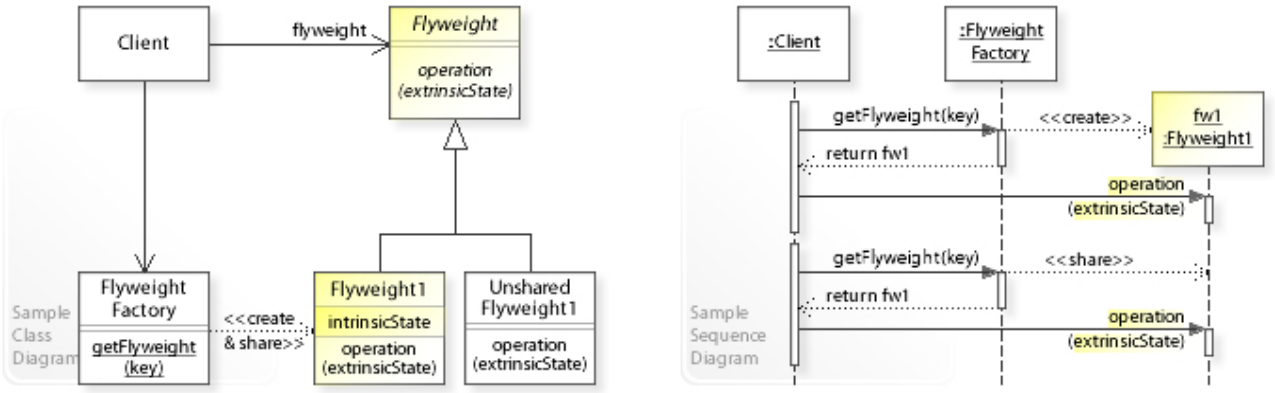


Developers often use the facade design pattern when a system is very complex or difficult to understand because the system has many interdependent classes or because its source code is unavailable. This pattern **hides the complexities of the larger system** and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.

Flyweight

Specifically, a flyweight is an object that minimizes memory usage by sharing as much data as possible with other similar objects. Flyweight is useful when dealing with objects in large numbers if a simple repeated representation would use an unacceptable amount of memory.

Classic example usage of the flyweight pattern is the data structures for graphical representation of characters in a word processor. It might be desirable to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data. However, this would amount to hundreds or thousands of bytes for each character. Instead, for every character there might be a reference to a flyweight glyph object shared by every instance of the same character in the document; only the position of each character (in the document and/or the page) would need to be stored internally.



What problems can the Flyweight design pattern solve?

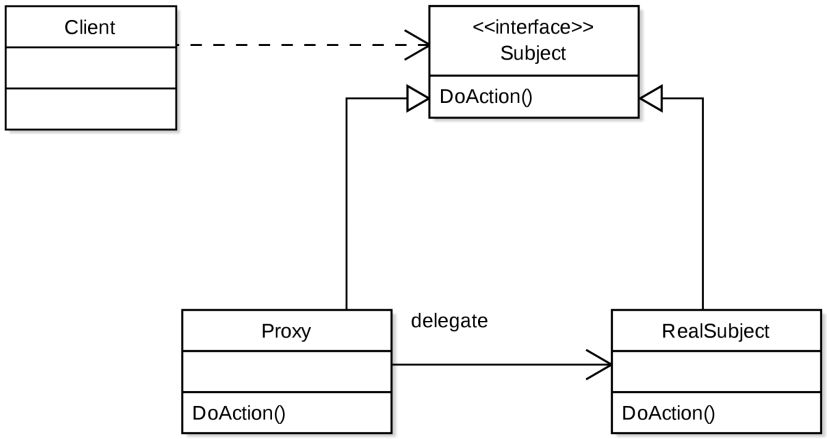
- Large numbers of objects should be supported efficiently.
- Creating large numbers of objects should be avoided.

What solution does the Flyweight design pattern describe? Define Flyweight objects that

- store intrinsic (invariant) state that can be shared and
- provide an interface through which extrinsic (variant) state can be passed in.

Proxy

A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.



What problems can the Proxy design pattern solve?

- The access to an object should be controlled.
- Additional functionality should be provided when accessing an object.
- When accessing sensitive objects, for example, it should be possible to check that clients have the needed access rights.

What solution does the Proxy design pattern describe? Define a separate Proxy object that

- can be used as substitute for another object (Subject) and
- implements additional functionality to control the access to this subject.

Remote proxy - In distributed object communication, a local object represents a remote object (one that belongs to a different address space).

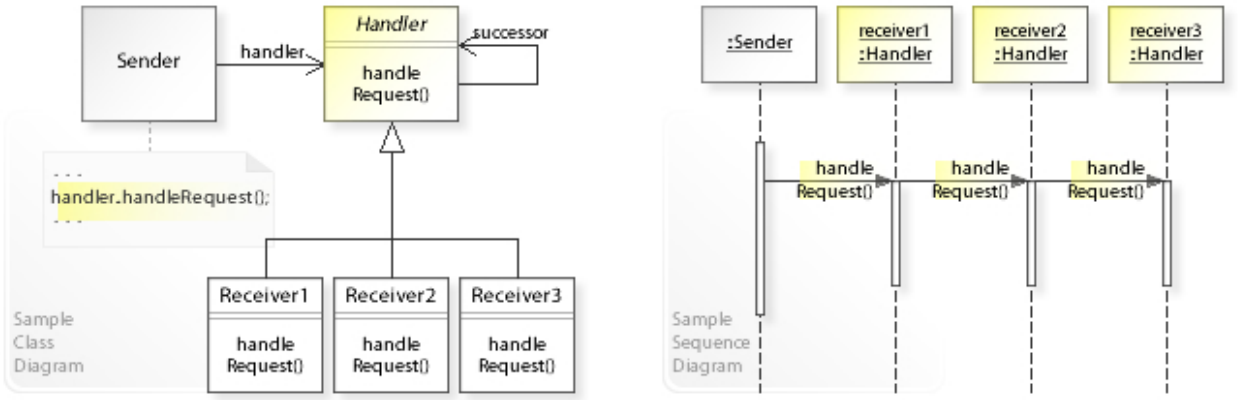
Virtual proxy - When an underlying object is huge in size, it may be represented using a virtual proxy object, loading the real object on demand.

Protection proxy - A protection proxy might be used to control access to a resource based on access rights.

Proxy & Decorator: Proxy's prime purpose is to facilitate ease of use or controlled access, a Decorator attaches additional responsibilities.

Chain of responsibility

The chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.



What problems can the Chain of Responsibility design pattern solve?

- Coupling the sender of a request to its receiver should be avoided.
- It should be possible that more than one receiver can handle a request.

What solution does the Chain of Responsibility design pattern describe?

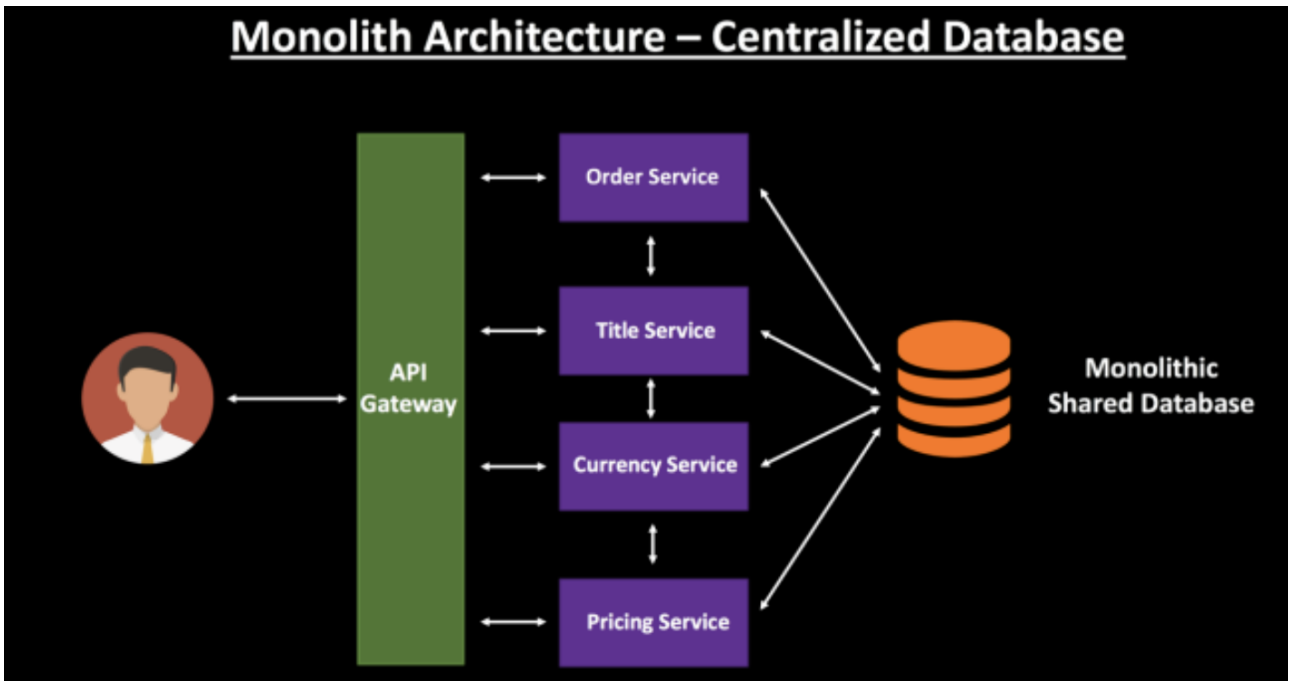
- Define a chain of receiver objects having the responsibility, depending on run-time conditions, to either handle a request or forward it to the next receiver on the chain (if any).

Example: Java ClassLoaders hierarchy

Microservice architecture

Problems With Monolithic Database Design

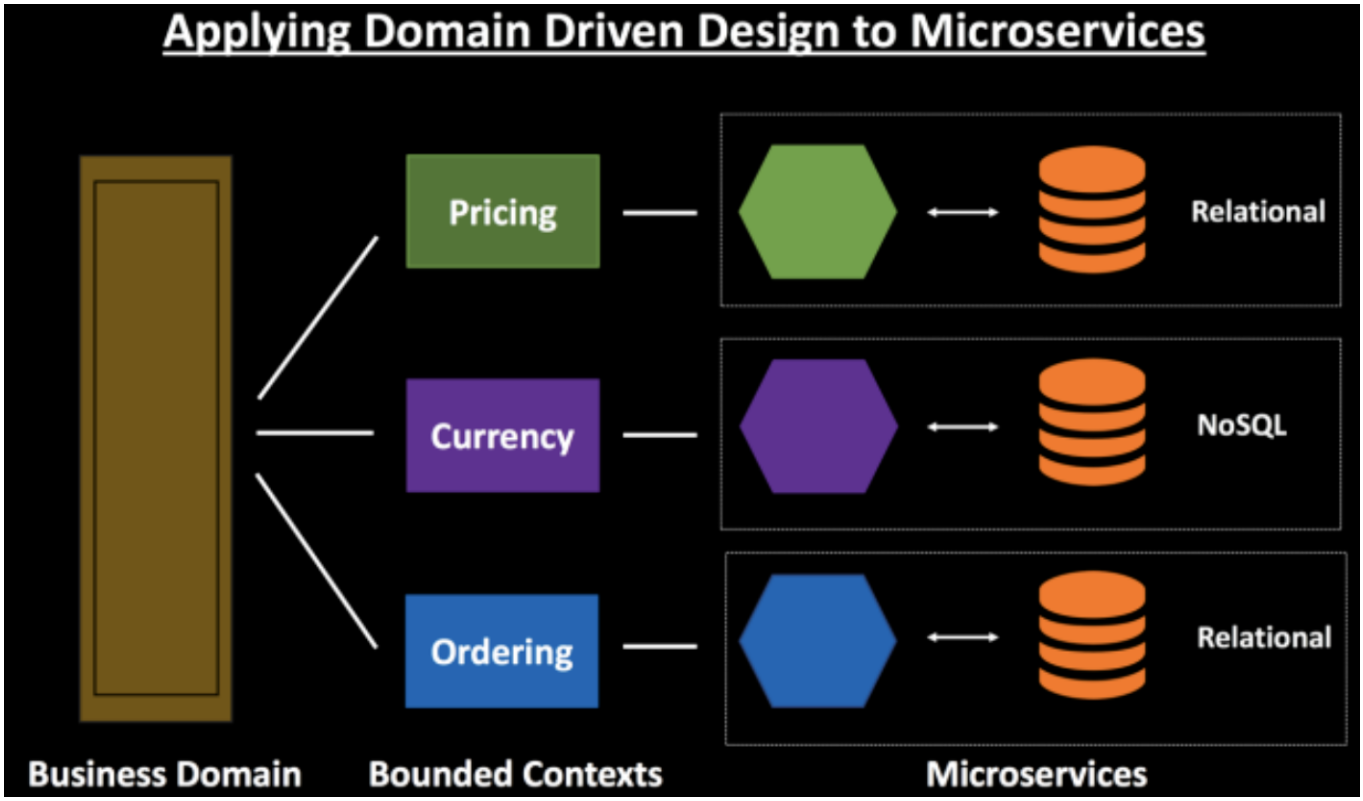
- The traditional design of having a Monolithic Database for multiple services creates a **tight coupling** and inability to deploy your service changes independently. If there are multiple services accessing the same database, any schema changes would need to be coordinated amongst all the services, which, in the real world, can cause additional work and delay in deploying changes.
- It is **difficult to scale** individual services with this design since you only have the option to scale out the entire monolithic database.
- Improving application performance becomes a challenge. With a single shared database, over a period of time, you end up having huge tables. This makes data retrieval difficult since you have to **join multiple big sized tables** to fetch the required data.
- Most of the times you have a relational store as your monolith database. **This constraints all your services to use a relational database**. However, there will be scenarios where a No-SQL datastore might be a better fit for your services and hence you don't want to be tightly coupled to a centralized datastore.



Domain driven design

Microservices should follow **Domain Driven Design** and have bounded contexts. You need to design your application **based on domains**, which aligns with the functionality of your application. It's like following the Code First approach over Data First approach - hence you **design your models first**. This is a fundamentally different approach than the traditional mentality of first designing your database tables when starting to work on a new requirement or greenfield project. You should always try to maintain the integrity of your Business model.

You should design your microservices architecture in such a way that each **individual microservice has its own separate database with its own domain data**. This will allow you to independently deploy and scale your microservices.



While designing your database, look at the application functionality and determine if it needs a relational schema or not. Keep your mind open towards a NoSQL DB as well if it fits your criteria.

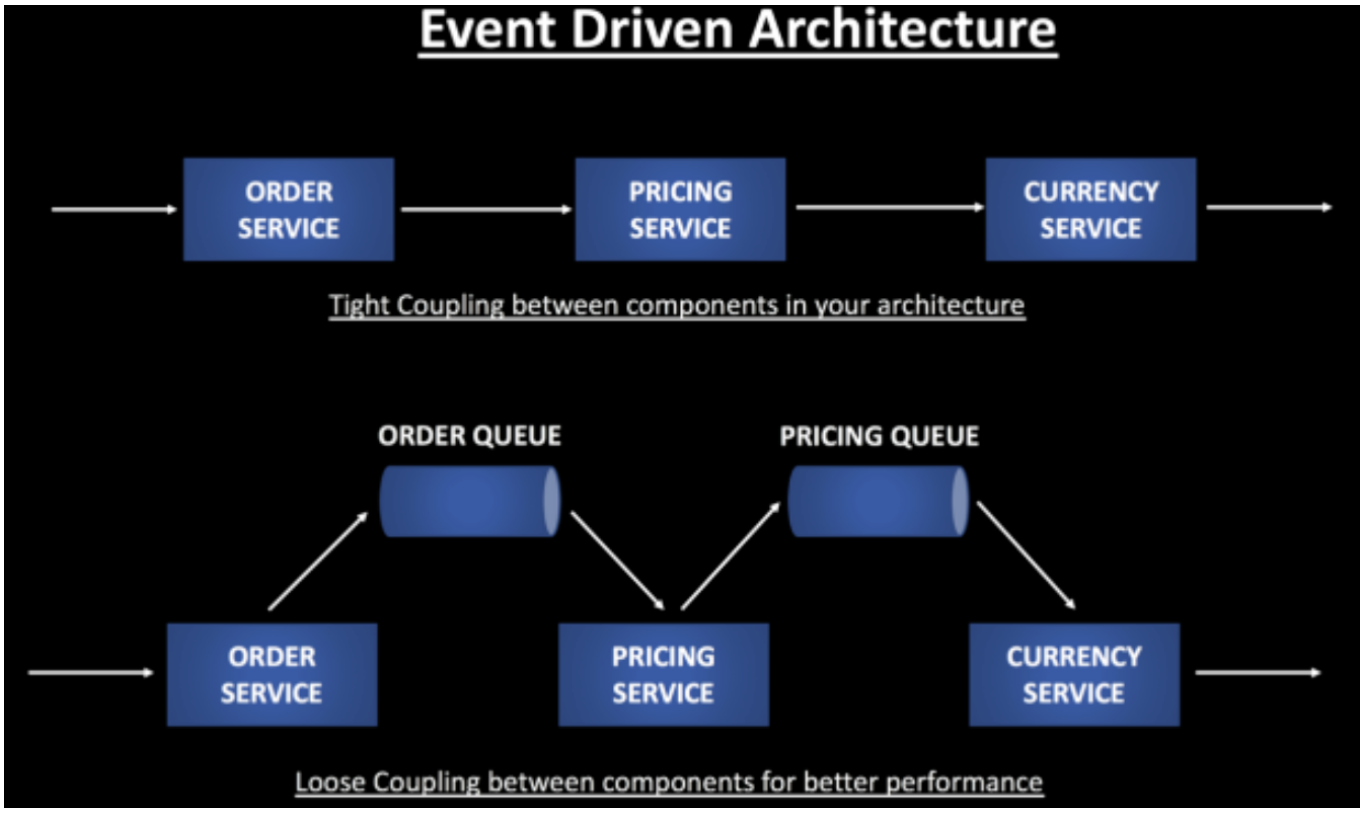
Databases should be treated as private to each microservice. No other microservice can directly modify data stored inside the database in another microservice.

Event-Driven Architecture

Event-Driven Architecture is a common pattern to maintain data consistency across different services. Instead of waiting for an ACID transaction to complete processing and taking up system resources, you

can make your application more available and performant by offloading the message to a queue. This provides loose coupling between services.

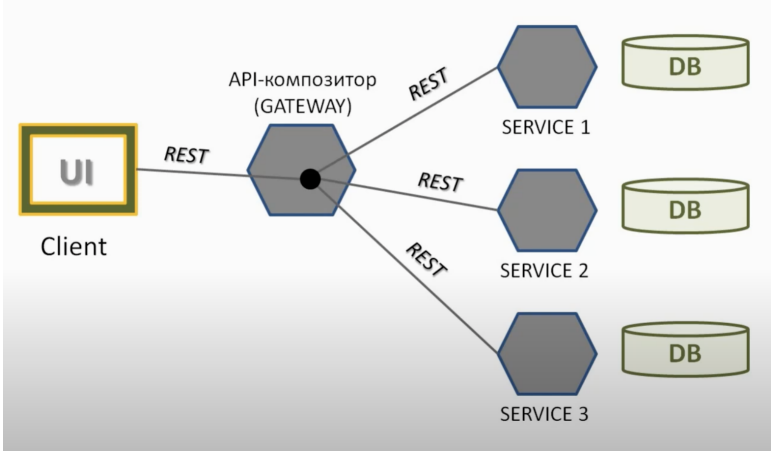
Messages to the queues can be treated as Events and can follow the Pub-Sub model. Publishers publishes a message and is not aware of the Consumer, who has subscribed to the event stream. **Loose coupling between components in your architecture enables to build highly scalable distributed systems**.



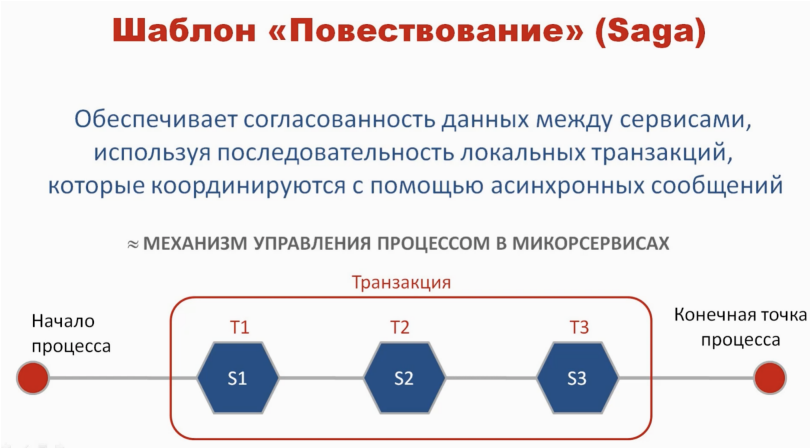
Decomposing the Database

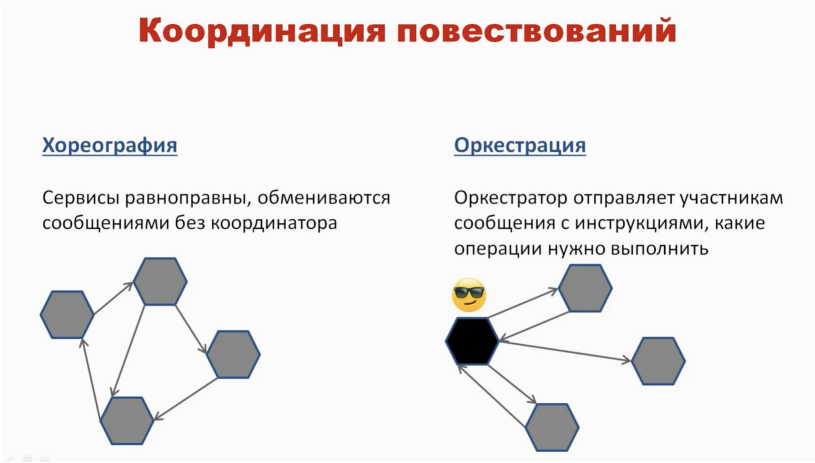
https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/ch04.html

Gateway pattern



Saga pattern





Хореография. Достоинства и недостатки

+ **Простота**
+ **Слабая связанность**
(Участники подписываются на события, не владея непосредственной информацией друг о друге)

- **Они сложнее для понимания**
- **Возникают циклические зависимости между сервисами**

Хореография может хорошо работать с простыми повествованиями, но в более сложных случаях лучше использовать оркестрацию

Оркестрация. Достоинства и недостатки

+ **Упрощенные зависимости**
+ **Улучшенное разделение ответственности и упрощенная бизнес-логика**

- **Риск избыточной централизации бизнес-логики**

Web service API

REST

Representational state transfer (REST) is a software architectural style which uses a subset of HTTP. It is commonly used to create interactive applications that use Web services. A Web service that follows these guidelines is called RESTful. REST is an alternative to, for example, SOAP as a way to access a Web service.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in **one of several formats** via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. **JSON** is the most generally popular programming language to use because, despite its name, it's **language-agnostic**, as well as **readable** by both humans and machines.

- **Client–server architecture**: The principle behind the client–server constraints is the separation of concerns. **Separating the user interface concerns from the data storage concerns** improves the portability of the user interfaces across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

- **Statelessness**: In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, **increasing performance by removing server load caused by retention of session information**.

- **Cacheability**: As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance

- **Layered system**: A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client and server, it won't affect their communications, and there won't be a need to update the client or server code. Intermediary servers can improve system scalability by enabling **load balancing** and by providing **shared caches**. Also, security can be added as a layer on top of the web services, separating business logic from **security logic**. Adding security as a separate layer enforces security policies. Finally, intermediary servers can call multiple other servers to generate a response to the client (Facade pattern).

- **Uniform interface**: The uniform interface constraint is fundamental to the design of any RESTful system. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
    - Resource identification in requests: Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.
    - Resource manipulation through representations: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state.
    - Self-descriptive messages: Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.
    - Hypermedia as the engine of application state (HATEOAS): Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.

REST API examples:

curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "gardener"}' {"id":3,"name":"Samwise Gamgee","role":"gardener"}
curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "ring bearer"}' {"id":3,"name":"Samwise Gamgee","role":"ring bearer"}
curl -v localhost:8080/employees [{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"thief"}]
curl -v localhost:8080/employees/99 {"id":99,"name":"Bilbo Baggins","role":"burglar"}
curl -X DELETE localhost:8080/employees/3

Idempotency

From a RESTful service standpoint, for an operation (or service call) to be **idempotent**, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

SOAP

gRPC

Protobuf