ITNEXT

Deconstructing patterns

The heart of software architecture, part 2

Denys Poltorak Follow 6 min read · Apr 7, 2025

> *This is a section from my book Architectural Metapatterns: the Pattern Language of Software Architecture which is available for free on Leanpub and GitHub. Any feedback is warmly welcome.*

Imagine a dungeon with dragons. It is made of halls connected by tunnels. Each hall is *cohesive*. Tunnels are narrow interfaces that *decouple* them. A hall is amorphous — it can have any shape but it cannot open to another hall except through a tunnel — such are the rules of the game. The tunnels both restrict the freedom of the halls and interconnect them.

SOLID principles

If cohesion and decoupling dictate software architecture, they should surface in its principles. Let's take a look at SOLID:

- *Single responsibility principle*, also known as "do one thing and do it well", is a general advice for keeping unrelated functionality decoupled.
- *Open-closed principle* and *Liskov substitution principle* decouple the logic of the parent class or the code that uses it, correspondingly, from the functionality of its subclasses.
- *Interface segregation principle* decouples independent parts of an object's interface.
- *Dependency inversion principle* decouples an object's users from its implementation.

Please beware that each of those principles involves decoupling which is not free — your software may end up containing too many moving parts and strict rules to remain easy to read and support.

Gang of Four patterns

Let's now discuss something more practical, namely GoF patterns that seem to be ingenious but hacky ways for rearranging roles in your code. They override ordinary OOP rules, which is useful when you need extra flexibility. For example, *creational patterns* interfere with the normally cohesive *select type — create — initialize — use* sequence of operating an object.

Some patterns provide basic decoupling:

- *Adapter* translates between two interacting components so that they may evolve independently.
- *Observer* decouples an event from reactions it causes by registering handlers at runtime.

- *Chain of Responsibility* separates method invocation from method execution. A client's calling a method of an object runs the corresponding method of another object.

Others break the functionality or data of a class into two or more parts, juggling them at runtime:

- *Proxy* separates an object's representation from its implementation, enabling lazy loading or remote access.
- *Flyweight* segregates an immutable data member of a class to save memory by merging multiple instances of identical data.
- *Strategy* and *Decorator* decouple a dimension of an object's functionality to allow runtime changes in or composition of the object's behavior, correspondingly.
- *State* separates an object's behavior into multiple classes based on the object's state.
- *Template Method* decouples several aspects of a class's behavior from its main algorithm and envelops variations of those aspects into its subclasses.
- *Bridge* separates a high-level hierarchy of classes from their low-level implementation details, which may comprise an orthogonal hierarchy.
- *Memento* decouples the lifetime of an object's state from the object itself.

On the other hand, a few patterns gather separate components together:

- *Command* collects all the data required to call a method.
- *Mediator* is a cohesive implementation of multi-object use cases.

- *Composite* and *Facade* represent multiple objects as a cohesive entity. A *composite* broadcasts a call to its interface to every object it contains while a *facade* orchestrates the wrapped subsystem.
- *Abstract Factory* and *Builder* encapsulate *type selection* and *initialization* for a set of related classes, so that the client code gets objects from a set of consistent types. On top of that, *Builder* cross-links the objects it creates into a cohesive subsystem, which is returned to the *builder*'s client as a whole.

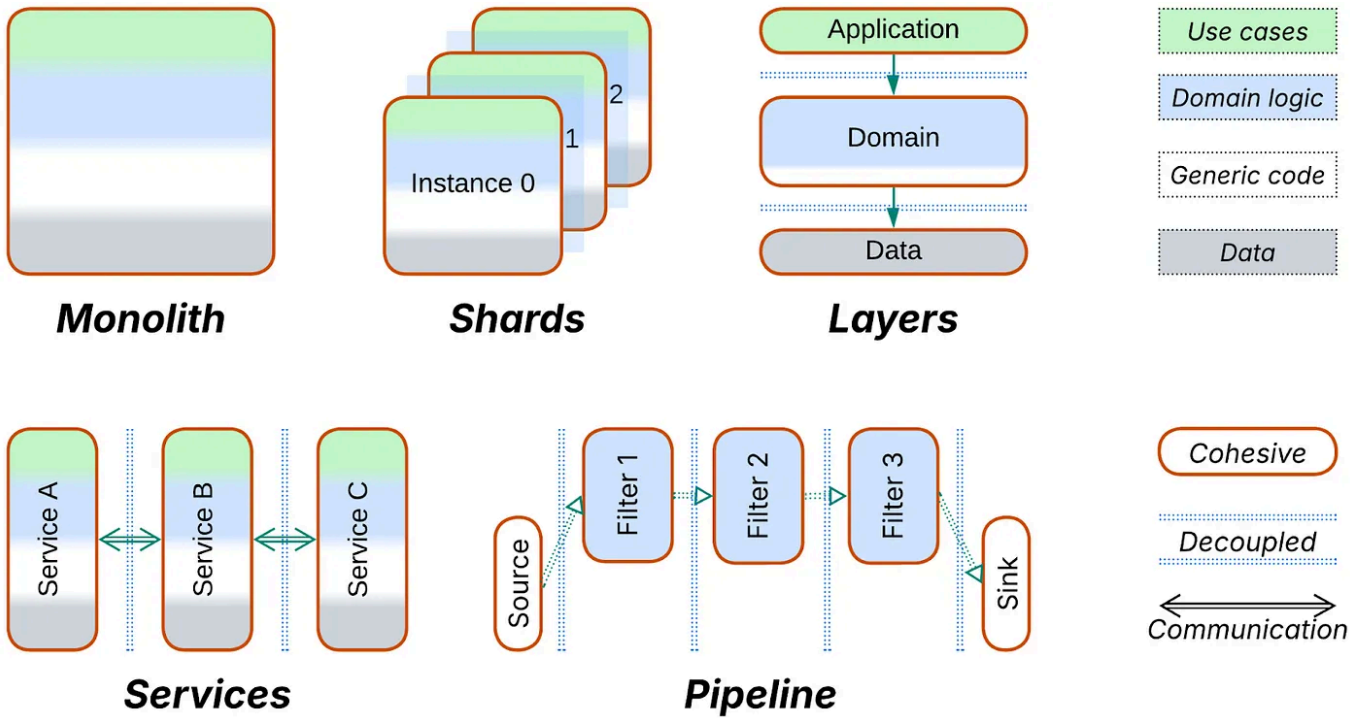
The remaining patterns pick an aspect or two of an object's behavior and move them elsewhere:

- *Iterator* moves the code for traversal of a container's elements from the container's clients into the container's implementation, decoupling its clients from the iteration algorithm.
- *Visitor* collects actions that a client needs to perform on each kind of object in a hierarchy, decoupling them from the classes that constitute the hierarchy.
- *Interpreter* decouples client scenarios from the rest of the system by having them written in a dedicated language and run in a protected environment.
- *Prototype* binds *type selection* and *initialization* together and decouples them from object *creation*.
- *Singleton* binds *creation* and *initialization* of a global object to every call of its methods.
- *Factory Method* decouples *initialization* from *type selection* and hides them from the class's users.

As we see, every GoF pattern boils down to binding (making *cohesive*) and/or separating (*decoupling*) some kind of functionality or responsibilities.

Architectural metapatterns

Finally, let's close the book by iterating over the metapatterns and looking into their roots through the prism of unification and separation.



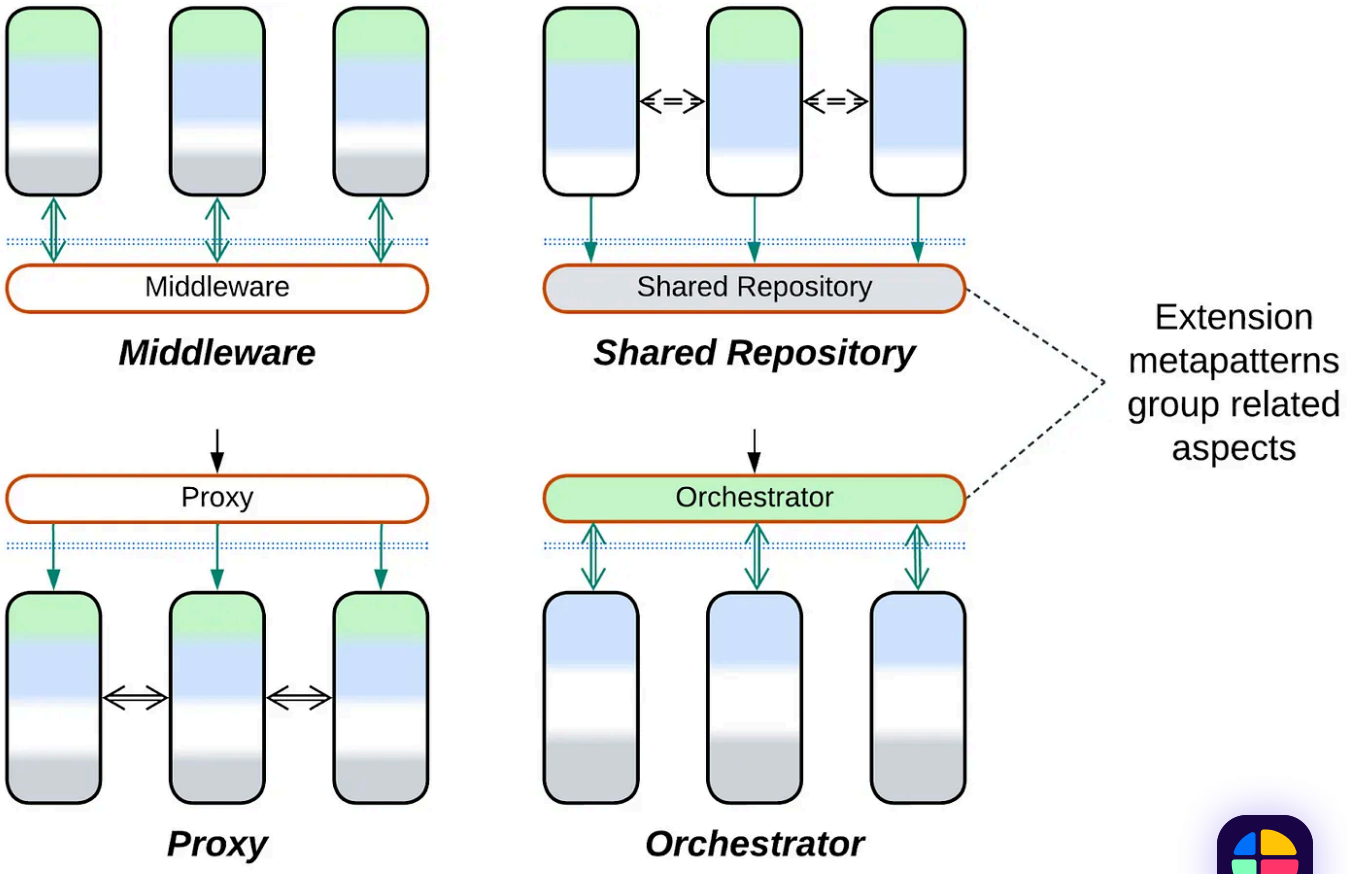
Basic architectures:

- *Monolith* keeps everything together for quick and dirty projects:
  - Total *cohesiveness* results in low latency, cost-efficient performance and easy debugging.
- *Shards* slice a large-scale application into multiple instances:
  - *Decoupling* instances enables scaling but sacrifices consistency of shared data.

- *Layers* separate high-level code from low-level implementation:
  - *Cohesion* within a layer makes it easy to implement and debug.
  - *Decoupled* layers may vary in technologies and properties but are somewhat slower and hard to debug in-depth.

- *Services* divide a complex system into subdomains:
  - *Cohesiveness* of a service keeps it simple and efficient when it does not need to consult other services.
  - *Decoupling* enables development of larger codebases by multiple specialized teams but global use cases become complicated.

- *Pipeline* segregates data processing into self-contained steps:
  - *Decoupling* simplifies reassembling or expanding the system but increases its latency.



Grouping related functionality:

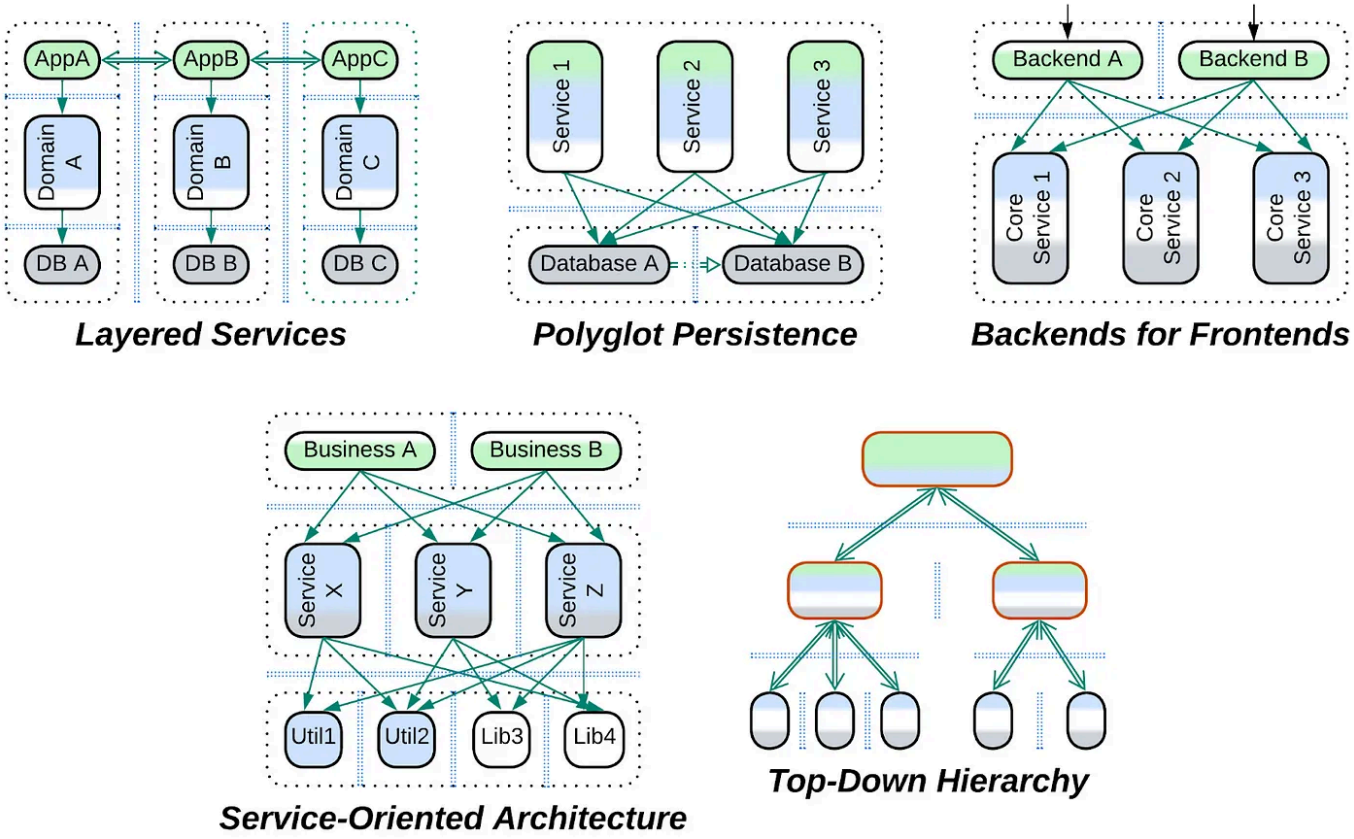
- *Middleware* separates the implementation of communication and/or instance management from the business logic:
  - The *cohesive* communication layer is reliable and uniform, thus easy to learn.
  - *Decoupling* communication concerns from the business logic simplifies the latter.

- *Shared Repository* dissociates data from the code, enabling data-centric programming:
  - *Cohesive* data is consistent and easy to handle.
  - *Decoupled* business logic can be scaled or subdivided independently of the data.

- *Proxy* mediates between a system and its clients, taking care of one or more aspects of their communication:
  - A *cohesive* edge component is easier to manage and secure.
  - *Decoupling* generic aspects simplifies business logic but usually increases latency.

- *Orchestrator* collects a multitude of complex use cases into a dedicated layer:
  - *Cohesive* use cases are easy to comprehend and debug.
  - *Decoupling* use cases from domain logic allows for variation in technologies but increases latency and complicates in-depth debugging.

- *Combined Component* blends two or three of the above layers:
  - *Cohesion* improves performance but reduces flexibility.

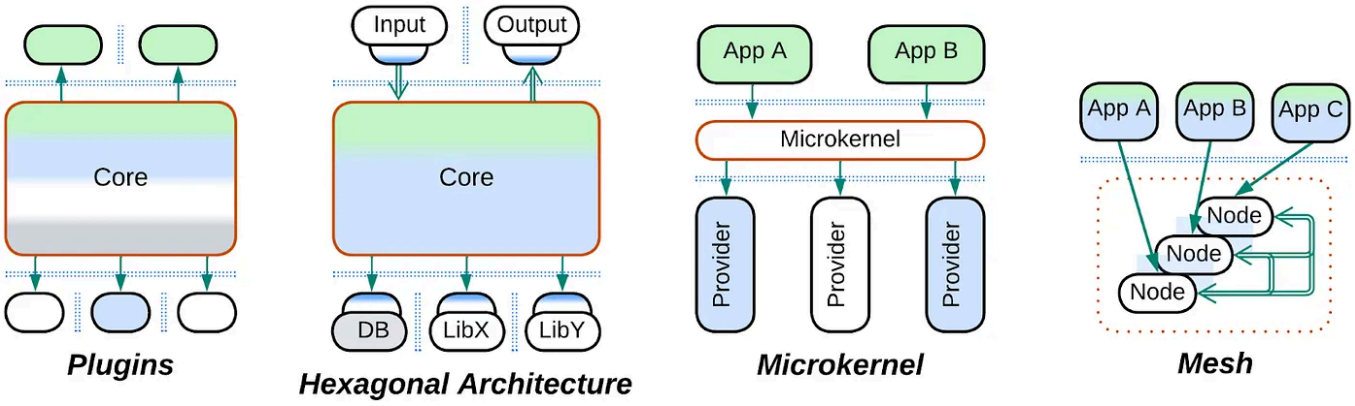


Decoupled systems:

- *Layered Services* first decouple subdomains, then — layers within each subdomain:
  - *Decoupled* subdomains allow for multi-team development and large codebases but complicate global use cases. *Decoupled* layers enable variation in technologies within a subdomain and limit interdependencies between subdomains to a single layer.
- *Polyglot Persistence* divides data among multiple data stores:
  - *Decoupling* improves performance through specialization at the cost of consistency.
- *Backends for Frontends* dedicates a module to every kind of client.
  - *Decoupling* allows for customization on a per-client basis.

- *Service-Oriented Architecture* first segregates a large system into layers, then subdivides each layer into services:
  - *Decoupling* layers strangely enables reuse as any component of an upper layer can access every component below it. *Decoupling* services within the layers allows for multi-team development. Drawbacks include high latency, system complexity and interdependencies.

- *Hierarchy* recursively separates general and specialized logic, tackling complexity:
  - *Cohesive* general and subdomain-specific business logic helps readability and debugging.
  - *Decoupled* layers and subdomains allow for local modification and expansion of the functionality at the cost of performance.



Component implementation:

- *Plugins* separate customizable aspects of a system's behavior:
  - *Decoupling* several aspects of a system allows for it to be fine-tuned but requires careful design and may lower performance.

- *Hexagonal Architecture* isolates the business logic from its external dependencies:

- *Decoupling* protects from vendor lock-in and supports automatic testing at the cost of lost optimization opportunities.

- *Microkernel* disjoins resource consumers from resource producers:
  - *Cohesive* resource management optimizes resource usage.
  - *Decoupling* allows for seamless replacement of resource providers.

- *Mesh* aggregates distributed components into a virtual layer:
  - Virtual *cohesion* hides the complexity of distributed communication from the client code.
  - Actual *decoupling* (distribution) of the nodes enables scaling and fault tolerance.

> *The final article will show how architectural patterns are selected based on your project's needs.*

Software Architecture | System Architecture | Software Design | Design Patterns | Computer Science