Architectural Metapatterns

Sample Chapter

v 1.1 (07-2025)

Denys Poltorak (author)
Lars Noodén (editor)

You’ll find inside

- A structured collection of architectural patterns with hundreds of NoUML diagrams.
- Technology-agnostic knowledge distilled from a multitude of sources.
- Deconstruction of software architecture into its basic principles.

Opentowork

I am looking for a good job. Embedded or high load C / C++. B2B from Ukraine. I can gather a team.

This book needs examples

Several readers told me that the patterns and principles should be illustrated with examples of their use in real-world systems. I cannot write them on my own because the scope of the book is much wider than my professional experience.

I am looking for both inline explanations about individual patterns (see blocks with gray background scattered throughout this book) and for one or two introductory case studies that will detail internal workings and evolution of complex real-world software to show how patterns are used in practice and promote the book to the duplex league.

Please consider sharing your experience as a co-author of a future version of this book.

Short table of contents

About this book

When I was learning programming, there was *Gang of Four*. The book promised to teach software design, and it did to an extent with the case study provided. However, the patterns it described were merely random tools which had little in common. After several years, having reinvented *Hexagonal Architecture* along the way, I learned about *Pattern-Oriented Software Architecture*. The series had many more intriguing patterns, and promised to provide a *system of patterns* or a *pattern language*, but failed to build an intuitive whole. Then there were specialized books with *Domain-Driven Design* and *Microservices* patterns. There was the *Software Architecture Patterns* primer by Mark Richards. Its simplicity felt great, but it had only 5 architectural styles, while his next book, *Fundamentals of Software Architecture*, dived too deeply into practical details and examples to be easily grasped.

Now, having leisure thanks to the war, burnout, unemployment and depression I have had a chance to collect architectural patterns from multiple sources and build a taxonomy of architectures. My goal was to write the very book I lacked in those early years: a shallow but intuitive overview of all the software and system architectures as used in practice, their properties and relations. I hope that it will be of some help both to novice programmers as a kind of a primer on the principles of high-level software design and to adept architects by reminding them of the big picture outside of their areas of expertise.

The book is mostly technology-agnostic. It does not answer practical questions like “Which database should I use?” Instead it inclines towards the understanding of “When should I use a shared database?” Any specific technologies ~~are easy to google~~ can be found ~~over the Internet~~ somewhere in the Noosphere.

This book started as a rather small project to prove that patterns can be intuitively classified (*These nightmarish creatures* can *be felled! They* can *be beaten!*) but grew into a multifaceted compendium of a hundred or so architectures and architectural patterns. It is grounded in the idea that software and system architecture evolves naturally, as opposed to being scientifically planned. Thus, the architectures may exhibit fractal features, just like those in biology – merely because the set of guidelines and forces remains the same for most systems that range from low-end embedded devices to world-wide financial networks. Moreover, in some cases we can see the same patterns applied to hardware design.

The idea of unifying software and system architecture is heretical. I am well aware of that. Still, the industry is in the early stage of alchemy these days: the same things are sold under multitudes of names, being remarketed or reinvented every decade. If this book manages to provide rules of thumb, similar to those of biology (a bat is a mammal, thus it should run on all four, while ostriches, as birds, must fly to Europe each spring), I will be happy with that. *Science makes progress funeral by funeral*.

The latest version of the book is available for free on GitHub and LeanPub. As there is no one who has practiced all the known architectures, it will be full of mistakes. I rely on your goodwill to correct them and improve the text. Critical reviews are warmly welcome: please write an email or contact me on LinkedIn.

Structure of the book

The [first chapter] explains the main idea which makes this book different from others. The following chapters in the [first part] touch on several general topics that are referenced throughout the book.

The next [four parts] iterate over *metapatterns* (clusters of closely related architectural patterns), starting with the simplest one, namely *Monolith*, then heading towards more complex systems that may be derived from *Monolith* by repeatedly dissecting it with interfaces. Each chapter describes a group of related patterns that share benefits and drawbacks, adds in a few references to books and websites, and summarizes the ways the patterns can be transformed into other architectures. The format of these chapters is described in Appendix F.

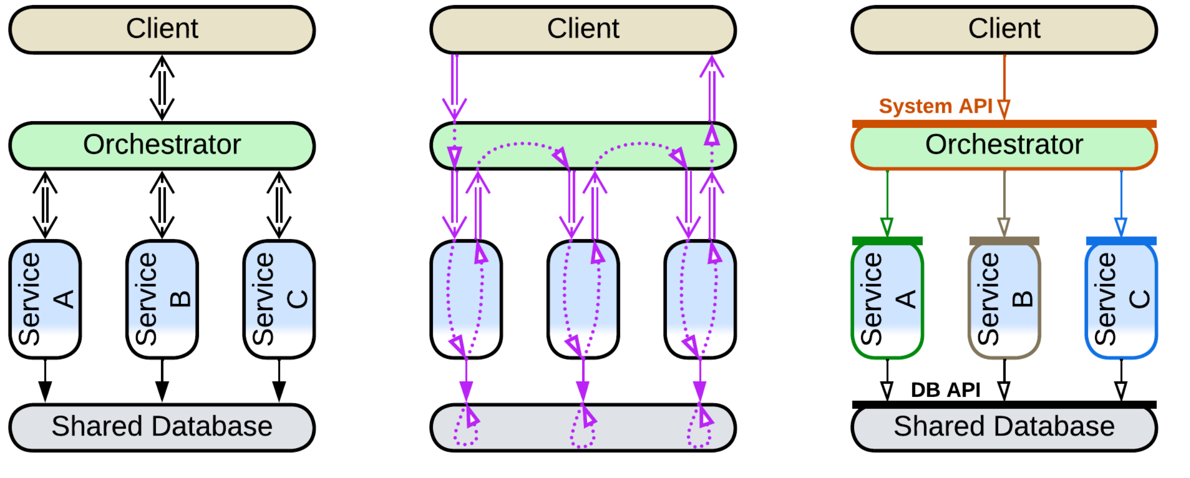
The sixth part of the book is analytics – the fruits of the pattern classification from the earlier parts.

Finally, there are appendices. Appendix B is the list of the books referenced, Appendix E contains detailed evolutions of patterns and Appendix I is the index of the patterns found in the book.

Diagrams

This book makes heavy use of diagrams – to the extent that it can be treated as a kind of visual novel. As it is mostly made of patterns, and *each pattern is an island*, it must not be read sequentially – instead, the reader is advised to use the plentiful cross-links to open whatever (if any) content found to be intriguing and check the corresponding diagram. If it gets your attention, you may read the text below it. If you like the text, you may scroll up or down to see if there are more funny diagrams nearby.

The diagrams are *NoUML* and most of them belong to one of the following kinds:



Structural Diagram ***Sequence Diagram*** ***Dependency Diagram***

Please refer to the following chapter for the legend and the system of coordinates.

Notation

- Pattern names are given in *Title Case Italics* and usually link to the pattern's definition.
- The first mention of a term or a name of a pattern component is *italicized*.
- *Quotes and puns are in full italics*.

- Book references are [[BRACKETED]] and link to the list of the books in Appendix B.
- Supplementary explanations are grayed-out.

Many patterns match terms of the common language – indeed, as a pattern is a generalization of human experience, the more widespread a notion, the faster it is turned into a pattern. Such general-use terms, e.g. layers, services or pipeline, are usually not indicated in any way to preserve the overall readability.

The architectural religions

There are several schools of software architecture:

1. The believers in SOLID.
2. The followers of eight qualities, five views and as-many-as-one-gets certifications.
3. The aspirants to the nameless way of patterns.

In my opinion:

1. SOLID is a silver bullet that tends to produce a DDD-layered kind of Hexagonal Architecture. It lacks the agility of pluralism found with evolutionary ecosystems.
2. Architectural frameworks are overcomplicated thus hard to understand and inflexible.
3. Patterns are like a kind of toolbox, the one which a mechanic is often seen carrying around. A skilled craftsman knows best uses of his tools, and can invent new instruments if something is missing in the standard toolset. However, the toolset's size should be limited for the tools to be familiar to the practitioner and easily carried around.

It is likely that those approaches are best used with systems of different sizes: SOLID is aimed at stand-alone application design while the heavy frameworks and certifications suit distributed enterprise architectures. In such a worldview patterns span everything in between the two extremes.

Patterns of software architecture are abstract just like Plato's Ideas or Forms in philosophy or classes in object-oriented programming. There is only one instance of each given pattern, which is a general idea or a very high-level blueprint for every implementation of the pattern ever seen in the code.

What's wrong with patterns

Too much information is no information or, as they say, *what is not remembered never existed*. There are literally thousands of patterns described for software and system architectures. Nobody knows them all and nobody cares to know them all (if you say you do, you must have already read the Pattern Languages of Programs archives. Have you? Neither have I). Hundreds of patterns are generated yearly in just the conferences alone, not to mention the books and software engineering websites. Old patterns get rebranded or forgotten and reinvented. This is especially true for the discrepancy between the pattern names in software architecture and system architecture. The new *N-tier* is just good old *Layers* under the hood, isn't it?

This undermines the original ideas which brought in the patterns hype:

1. *Patterns as a ubiquitous language*. Nowadays similar, if not identical, patterns bear different names, and some of them are too obscure to be ever heard of (see the PLoP archives).
2. *Patterns as a vessel for knowledge transfer*. If an old pattern is reinvented or plagiarized, most of the old knowledge is lost. There is no continuity of experience.
3. *Pattern language as the ultimate architect's tool*. As patterns are re-invented, so are pattern languages. At best, we have domain-specific or architecture-limited (*DDD*, *Microservices*) systems of patterns. There is no *single unified vision* which pattern enthusiasts of old promised to provide.

Have we been fooled?

TLDR

Compare *Firewall* and *Response Cache*. Both represent a system to its users and implement generic aspects of the system's behavior. Both are *Proxies*.

Take *Saga Execution Component* and *API Composer*. Both are high-level services that make a series of calls into an underlying system – they *orchestrate* it. Both are *Orchestrators*.

It's that simple and stupid. We can classify architectural patterns.

Metapatterns

Is there a way to bring the patterns into order? They are way too many, some obscure, others overly specialized.

We can try. On a subset. And the subset should be:

- *Important* enough to matter for the majority of programmers.
- *Small* enough to fit in one's memory or in a book.
- *Complete* enough to assure that we don't miss anything crucial.

Is there such a set? I believe so.

Architectural patterns

[POSA1] defines three categories of patterns:

- *Architectural patterns* which deal with the overall structure of a system and functions of its components.
- *Design patterns* which describe relations between objects.
- *Idioms* which provide abstractions on top of a given programming language.

Architectural patterns are important by definition (*Architecture is about the important stuff. Whatever that is*). Point 1 (*importance*) – checked.

Any given system has an internal structure. When its developers talk about *architectural style* [POSA1] or draw structural diagrams that usually boils down to a composition of two or three well-known architectural patterns. Choosing architectural patterns as the subject of our study lets us feed on a large body of books and articles that describe similar designs over and over again. Moreover, as soon as a system no longer follows the latest fashions, it is widely advertised as a novelty (or its designers are labeled as old-fashioned and shortsighted), thus we may expect to have heard of nearly all of the architectures which are used in practice. Point 3 (*completeness*) – we have more than enough examples to analyze.

To organize a set of patterns we rely on the concept of

Design space

Design space [POSA1, POSA5] is a model that allocates a dimension for each choice made while architecting the system. Thus it contains all the possible ways for a system to be designed. The only trouble – it is multidimensional, maybe infinite, and the dimensions will differ from system to system.

There is a workaround – we can use a projection from the design space into a 2- or 3-dimensional space which we are more comfortable with. However, projection causes a loss of information. Counterintuitively, that is good for us – similar architectures that differ in small details become identical as soon as the dimensions they differ in disappear. If we could only find 2 or 3 most important dimensions that apply equally well to each pattern in the set that we want to research, that is architectural patterns, which cover all the known system designs.

Structure determines architecture

Systems tend to have an internal structure. Those that don't are derogatively called *Big Balls of Mud* for their peculiar properties. Structure is all about components, their roles and interactions. Many architectural styles, for example, *Layers* and *Pipeline*, are named after their structures, while others, like *Event-Driven Architecture*, highlight some of its aspects, hinting that it is the structure which determines principal properties of a system.

I am not the first person to reach such a conclusion. *Metapatterns* – clusters of patterns of similar structure – were defined shortly after the first collections of design patterns had appeared but they never made a lasting impact on software engineering. I believe that the approach was applied prematurely to analyze the [GoF] patterns, which make quite a random and incomplete subset of design patterns, resulting in an overgeneralization. I intend to plot structures of all the architectural patterns I encounter, group patterns of identical structure together into metapatterns, draw relations between the metapatterns, and maybe show how a system's structure determines its properties. Quite an ambitious plan for a short book, isn't it?

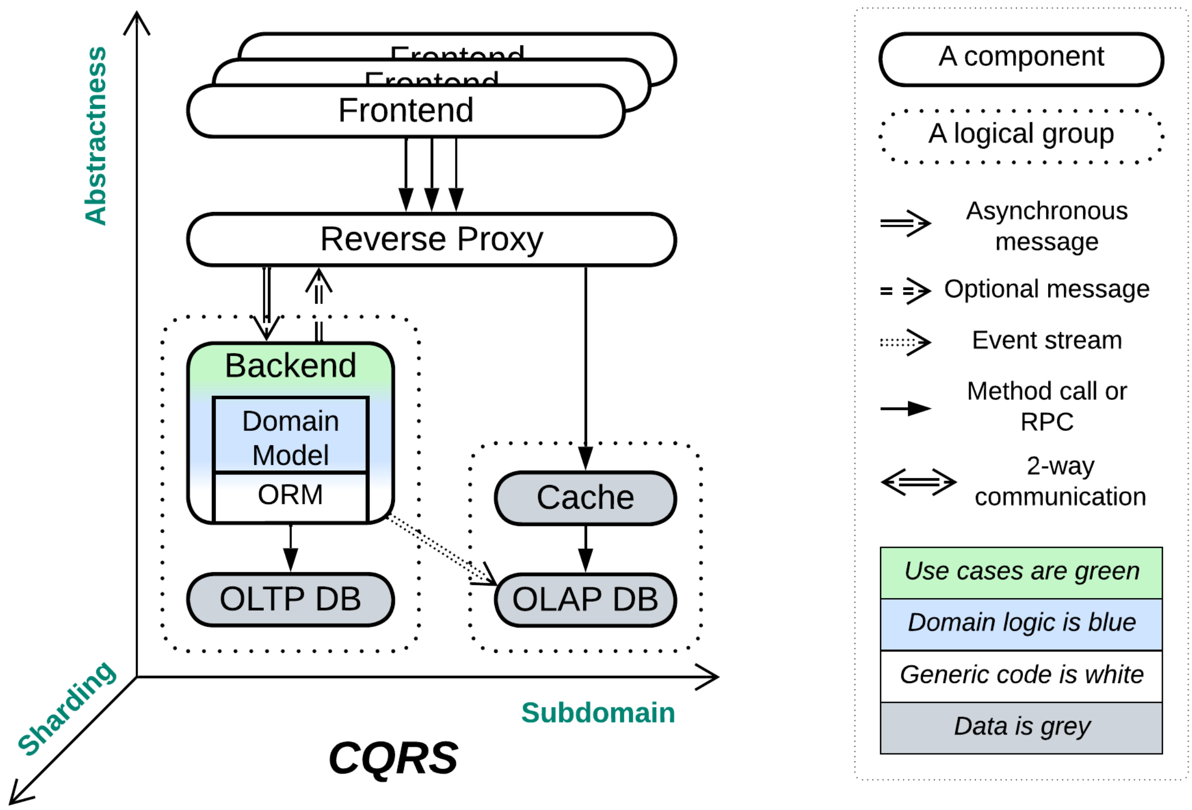
Our set of architectural patterns is still not known to be complete, is not small and, moreover, the way structural diagrams are drawn differs from source to source – we cannot compare them unless we make up a universal system of coordinates.

The system of coordinates

Inventing a generic coordinate system to fit any pattern's representation, from *Iterator* [GoF] to *Half-Sync/Half-Async* [POSA2], may be too hard, but we surely can find something for architectural patterns, as all of them share the scope, namely the system as a whole. Which dimensions an implementation of a system would usually be plotted along?

1. *Abstractness* – there are high-level use cases and low-level details. A single highly abstract operation unrolls into many lower-level ones: Python scripts run on top of a C runtime and assembly drivers; orchestrators call API methods of services, which themselves run SQL queries towards their databases which are full of low-level computations and disk operations.
2. *Subdomain* – any complex system manages multiple subdomains. An OS needs to deal with a variety of peripheral devices and protocols: a video card driver has very little resemblance to an HDD driver or to the TCP/IP stack. An enterprise has multiple departments, each operating a software that fits its needs.
3. *Sharding* – if several instances of a module are deployed, and that fact is an integral part of the architecture, we should represent the multiple instances on our structural diagram.

We'll draw the abstractness axis vertically with higher-level modules positioned towards the upper side of the diagram, the subdomain axis horizontally, and sharding diagonally. Here is an (arbitrary) example of such a diagram:



(A structural diagram for [CQRS](), adapted from [Udi Dahan's article](), to introduce the notation)

Map and reduce

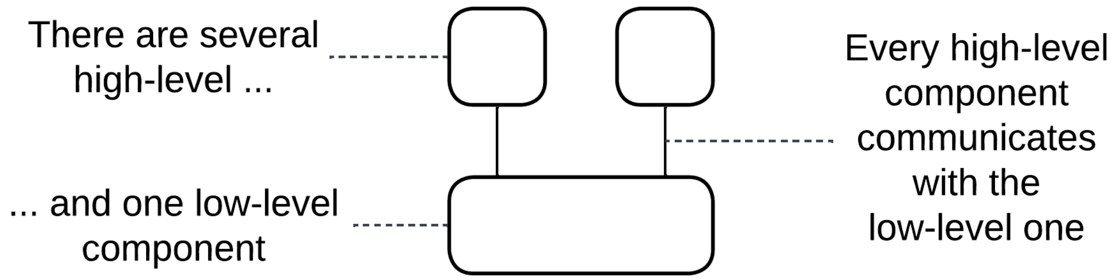
Now that we have the generic coordinates which seem to fit any architectural pattern, we can start mapping our set of architectural patterns into that coordinate system – the process of reducing the multidimensional design space to the few dimensions of structural diagrams which we were looking for. Then, after filtering out minor details, our hundred or so of published patterns should yield a score of clusters of geometrically equivalent diagrams – just because there are very few simple systems that one can draw on a plane before repeating oneself. Each of the clusters will represent an *architectural metapattern* – a generalization of architectural patterns of similar structure and function.

Let's return for a second to our requirements for classifying a set of patterns. The importance (point 1) of architectural patterns was proved before. The reasonable size of the resulting classification (point 2) is granted by the existence of only a few simple 2D or 3D shapes (metapatterns). The completeness of the analysis (point 3) comes from, on one hand, the geometrical approach which makes any blank spaces (possible geometries with no known patterns) obvious, and on the other – from the large sample of architectural patterns which we are classifying.

Godspeed!

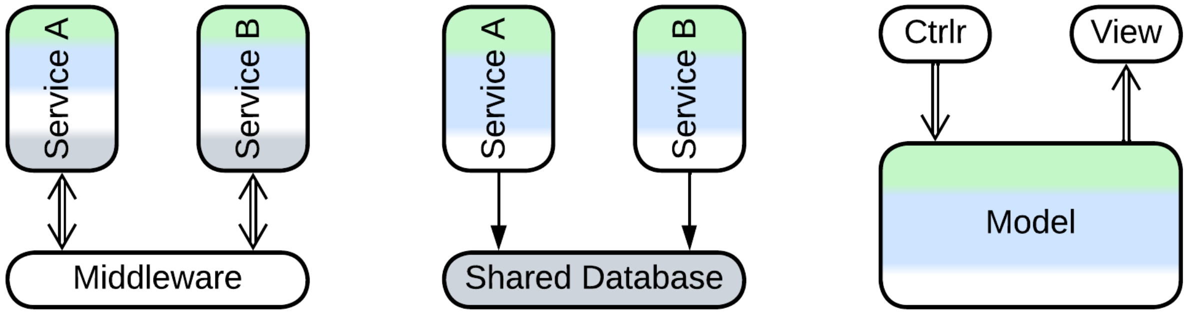
An example of metapatterns

Let's consider the following structure:



It features two (or more in real life) high-level modules that communicate with/via a lower-level module. Which patterns does it match?

- *Middleware* – a software that provides means of communication between other components.
- *Shared Database* – a space for other components to store and exchange data.
- *Model-View-Controller* – a platform-agnostic business logic with customized means of input and output.



My idea of grouping patterns by structure seems to have backfired – we got three distinct patterns that have similar structural diagrams. The first two of them are related – both implement indirect communication, and their distinction is fading as a *Middleware* may feature a persistent storage for messages while a table in a *Shared Database* may be used to orchestrate services. The third one is very different – primarily because the bulk of its code, that is its *business logic*, resides in the lower layer, leaving the upper-level components a minor role.

Notwithstanding, each of the patterns we found is a part of a distinct cluster:

- *Middleware* is also known as *(Message) Broker* [POSA1, POSA4, EIP, MP] and is an integral part of *Message Bus* [EIP], *Service Mesh* [FSA], *Event Mediator* [FSA], *Enterprise Service Bus* [FSA] and *Space-Based Architecture* [SAP, FSA].
- *Shared Database* is a kind of *Shared Repository* [POSA4] (*Shared Memory*, *Shared File System*), and the foundation for *Blackboard* [POSA1, POSA4], *Space-Based Architecture* [SAP, FSA], and *Service-Based Architecture* [FSA].
- *Model-View-Controller* [POSA1, POSA4] is a special kind of *Hexagonal Architecture* (aka *Ports and Adapters*, *Onion Architecture* and *Clean Architecture*) which itself is derived from *Plugins* [PEAA] (*Addons*, *Plug-In Architecture* [FSA], or the misnomer *Microkernel Architecture* [SAP, FSA]).

Our touching on a single geometry of structural diagrams revealed a web of 20 or so pattern names that spreads all around. With such a pace there is a hope of exploring the whole fabric which is known as *pattern language* [GoF, POSA1, POSA2, POSA5].

There are three lessons to learn:

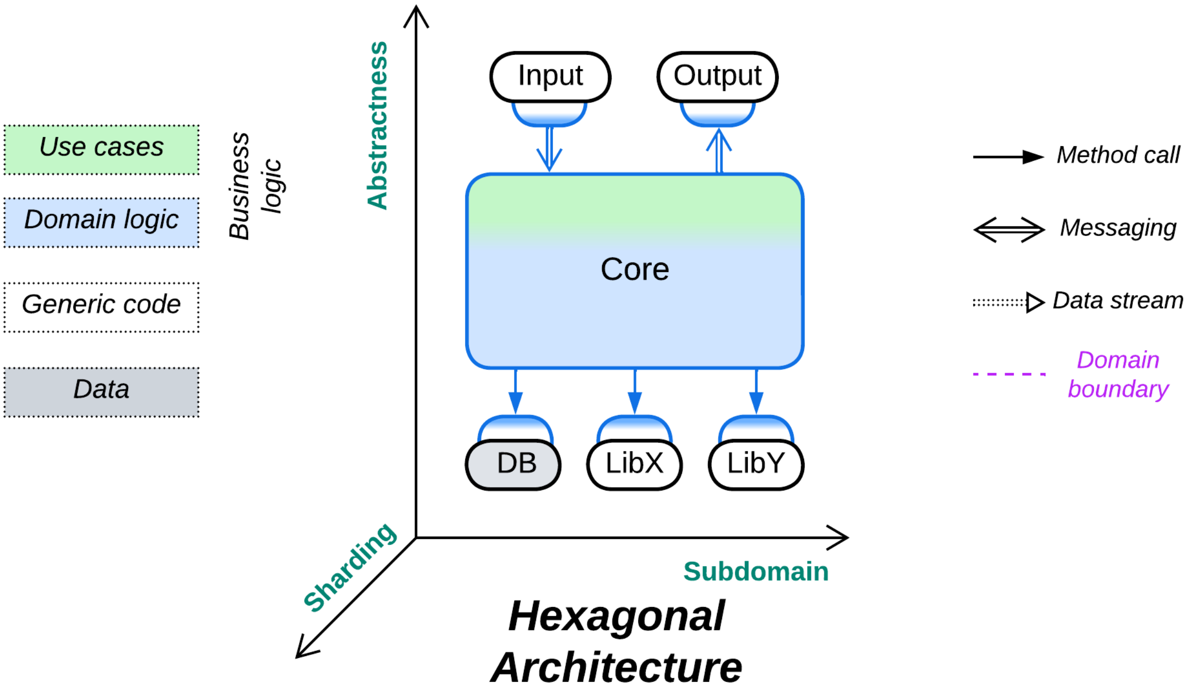
- The distribution of business logic is a crucial aspect of structural diagrams.
- Metapatterns are interrelated in multiple ways, forming a pattern language.

- Each metapattern includes several well-established patterns.

What does that mean

Chemistry has the periodic table. Biology has the tree of life. This book strives towards building something of that kind for software and systems architecture. You can say “That makes no sense! Chemistry and biology are empirical sciences while software architecture isn’t!” Is it?

Hexagonal Architecture



Trust no one. Protect your code from external dependencies.
Known as: Hexagonal Architecture, or originally as Ports and Adapters.
Variants:
By placement of adapters:

- Adapters on the external component's side.
- Adapters on the *core* side.

Examples – Hexagonal Architecture:

- Hexagonal Architecture / Ports and Adapters,
- DDD-Style Hexagonal Architecture [LDDD] / Onion Architecture / Clean Architecture.

Examples – Separated Presentation:

- (*Layered*) Model-View-Presenter (MVP), Model-View-Adapter (MVA), Model-View-ViewModel (MVVM), Model 1 (MVC1), Document-View [POSA1],
- (*Pipelined*) Model-View-Controller (MVC) [POSA1, POSA4] / Action-Domain-Responder (ADR) / Resource-Method-Representation (RMR) / Model 2 (MVC2) / Game Development Engine.

Structure: A monolithic business logic extended with a set of (adapter, component) pairs that encapsulate external dependencies.
Type: Implementation.

Benefits	*Drawbacks*
Isolates business logic from external dependencies	Suboptimal performance
Facilitates the use of stubs/mocks for testing and development	The vendor-independent interfaces must be designed before the start of development
Allows for qualities to vary between the external components and the business logic	
The programmers of business logic don't	

need to learn any external technologies

References: Herberto Graça's chronicles is the main collection of patterns from this chapter. *Hexagonal Architecture* has the original article and a brief summary of its layered variant in [LDDD]. Most of the *Separated Presentation* patterns are featured on Wikipedia and there are collections of them from Martin Fowler, Anthony Ferrara and Derek Greer.

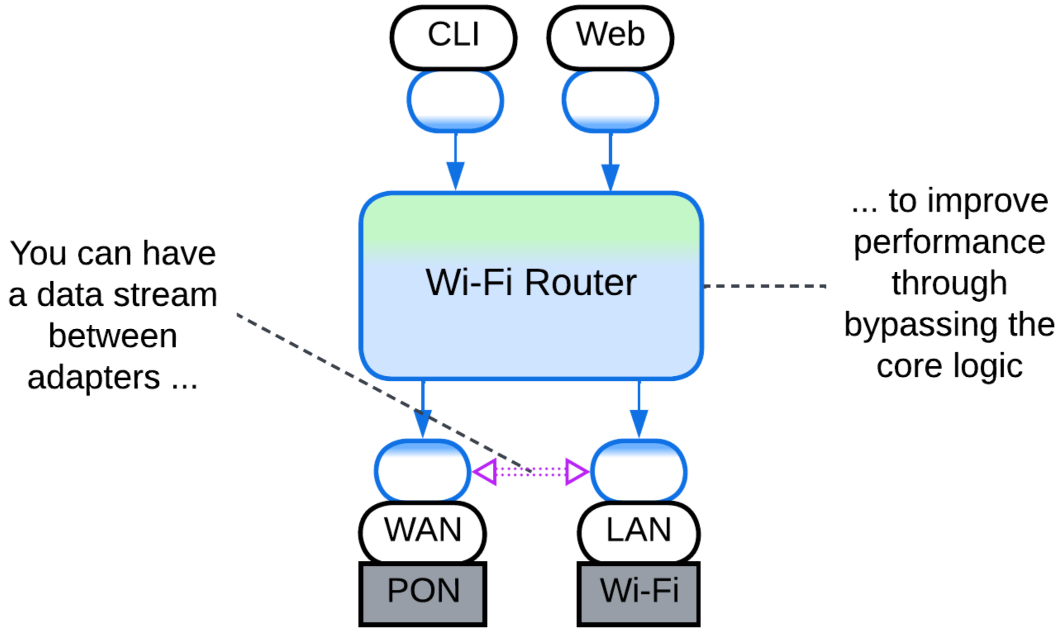
Hexagonal Architecture is a variation of *Plugins* that aims for total self-sufficiency of business logic. Any third-party tools, whether libraries, services or databases, are hidden behind *adapters* [GoF] that translate the external module's interface into a *service provider interface* (*SPI*) defined by the *core* module and called *port*. The core's business logic depends only on the ports that its developers defined – a perfect use of *dependency inversion* – and manipulates interfaces that were designed in the most convenient way. The benefits of this architecture include the core's cross-platform nature, easy development and testing with stubs or mocks, support for event replay and protection from vendor lock-in. It also allows for replacement of any external library at late stages of the project. The flexibility is paid for with a somewhat longer system design stage and lost optimization opportunities. There is also a high risk to design a leaky abstraction – an SPI that looks generic but whose contract matches that of the component it encapsulates, making it much harder than expected to change the component's vendor.

Stubs and *mocks* are *test doubles* – simplistic replacements for real-world components. They are used to run the business logic in isolation – without the need to deploy any heavyweight libraries or services the logic may depend upon. A *stub* supports a single usage scenario in a single test case while a *mock* is more generic – its behavior is programmed on a per test basis.

Performance

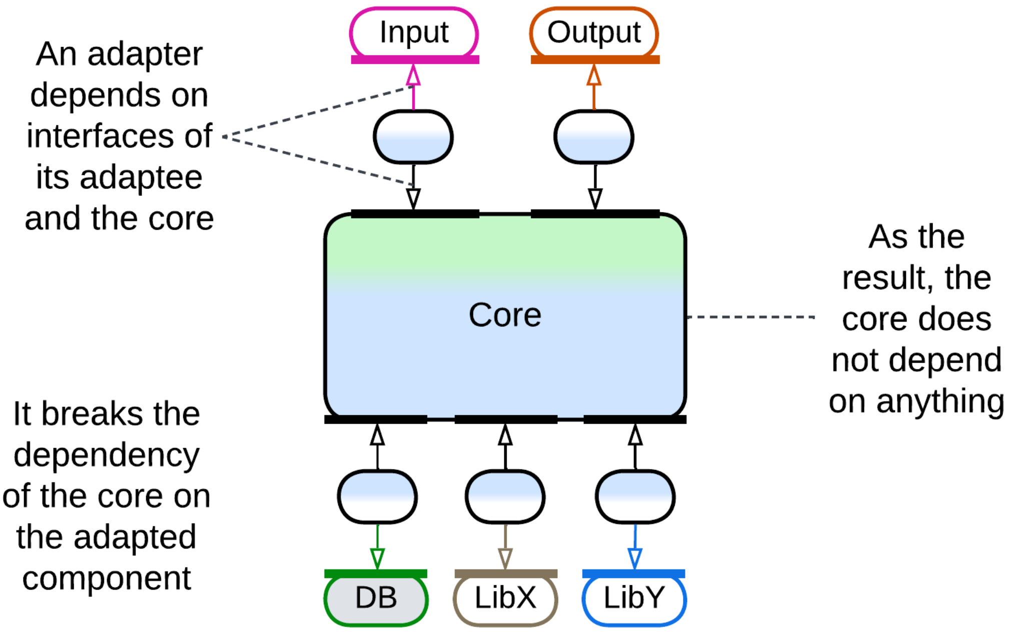
Hexagonal Architecture is a strange beast performance-wise. The generic interfaces (ports) between the core and adapters stand in the way of whole-system optimization and may add context switching. Still, at the same time, each adapter concentrates all the vendor-specific code for its external dependency, which makes the adapter a perfect single place for aggressive optimization by an expert or consultant who is proficient with the adapted third-party software but does not have time to learn the details of your business logic. Thus, some opportunities for optimization are lost while others emerge.

In rare cases the system may benefit from direct communication between the adapters. However, that requires several of them to be compatible or polymorphic, in which case your *Hexagonal Architecture* may in fact be a kind of shallow *Hierarchy*. Examples include a service that uses several databases which are kept in sync through *Change Data Capture* (*CDC*) or a telephony gateway that interconnects various kinds of voice devices.



Dependencies

Each adapter breaks the dependency between the core that contains business logic and an adapted component. This makes all the system's components mutually independent – and easily interchangeable and evolvable – except for the adapters themselves, which are small enough to be rewritten as need arises.



Applicability

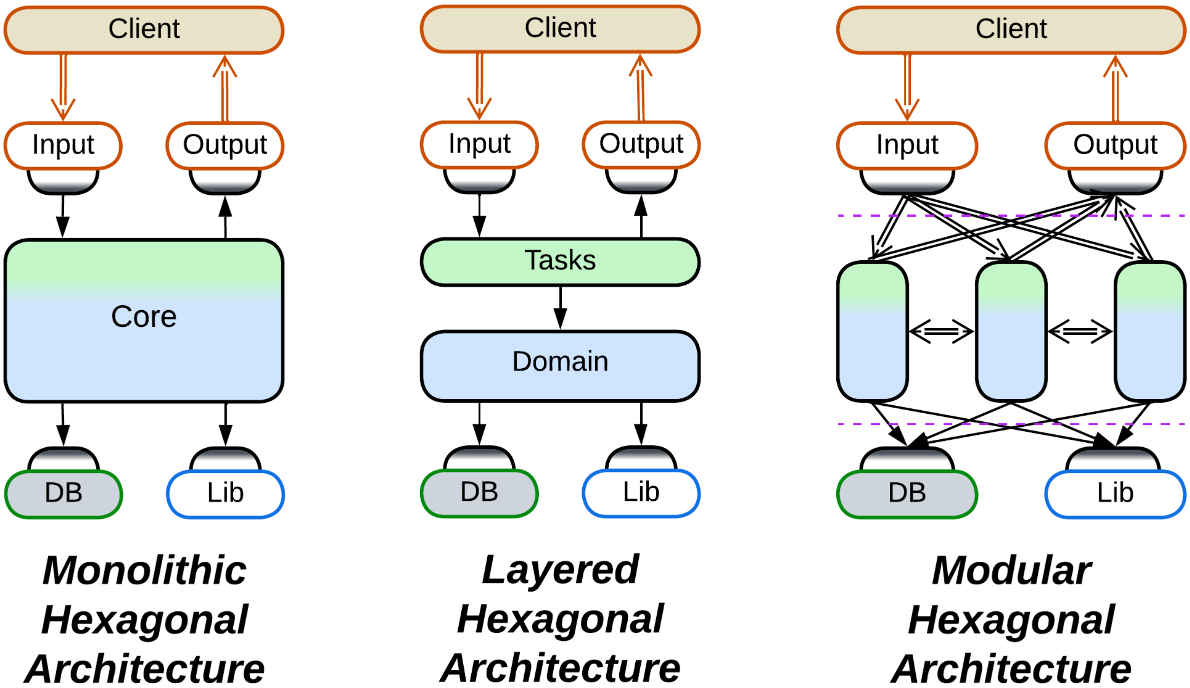
Hexagonal Architecture benefits:

- *Medium-sized or larger components.* The programmers don't need to learn details of external technologies and may concentrate on the business logic instead. The code of the *core* becomes smaller as all the details of managing external components are moved into their *adapters*.

- *Cross-platform development.* The core is naturally cross-platform as it does not depend on any (platform-specific) libraries.
- *Long-lived products*. Technologies come and go, your product remains. Always be ready to change the technologies it uses.
- *Unfamiliar domain.* You don't know how much load you'll need your database to support. You don't know if the library you selected is stable enough for your needs. Be prepared to replace vendors even after the public release of your product.
- *Automated testing.* Stubs and mocks are great for reducing load on test servers. And stubs for the SPIs which you wrote yourself are easy as a pie.
- *Zero bug tolerance.* SPIs allow for event replay. If your business logic is deterministic, you can reproduce your user's bugs in your office.

Hexagonal Architecture is not good for:
- *Small components.* If there is little business logic, there is not much to protect, while the overhead of defining SPIs and writing adapters is high compared to the total development time.
- *Write-and-forget projects.* You don't want to waste your time on long-term survivability of your code.
- *Quick start*. You need to show the results right now. No time for good architecture.
- *Low latency*. The adapters slow down communication. This is somewhat alleviated by creating direct communication channels between the adapters to bypass the core.

Relations

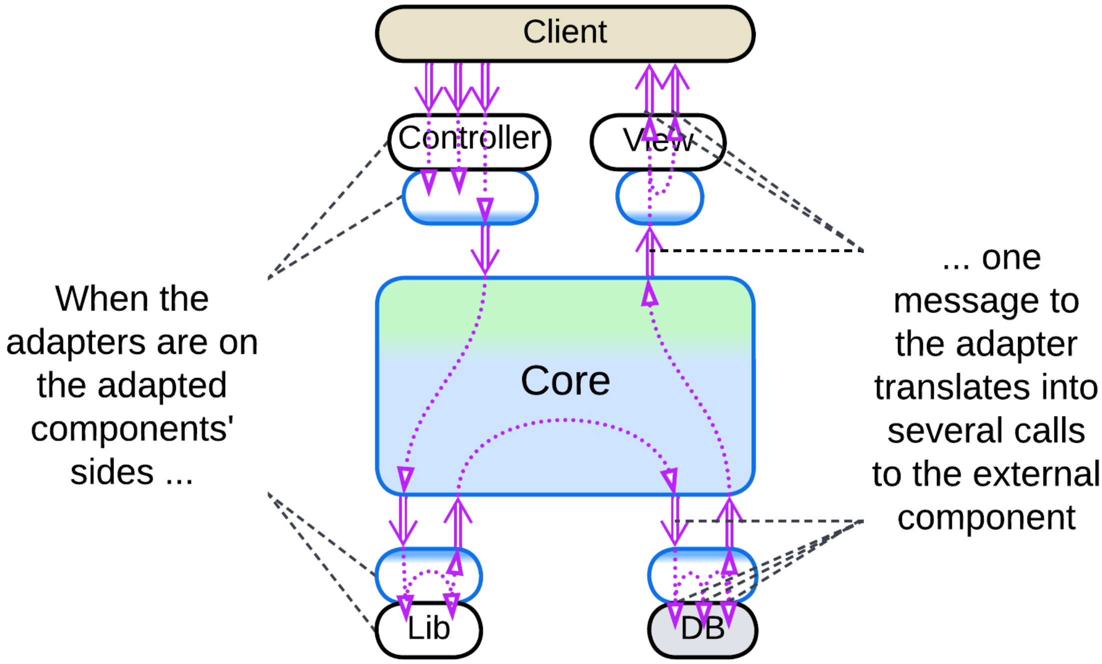


Hexagonal Architecture:
- Is a kind of *Plugins*.
- May be a shallow *Hierarchy*.
- Implements *Monolith* or *Layers*.
- Extends *Monolith*, *Layers* or, rarely, *Services* with one or two layers of *services*.
- The *MVC family of patterns* is also derived from *Pipeline*.

Variants by placement of adapters

One possible variation in a distributed or asynchronous *Hexagonal Architecture* is the deployment of *adapters*, which may reside adjacent to the core or with the components they adapt:

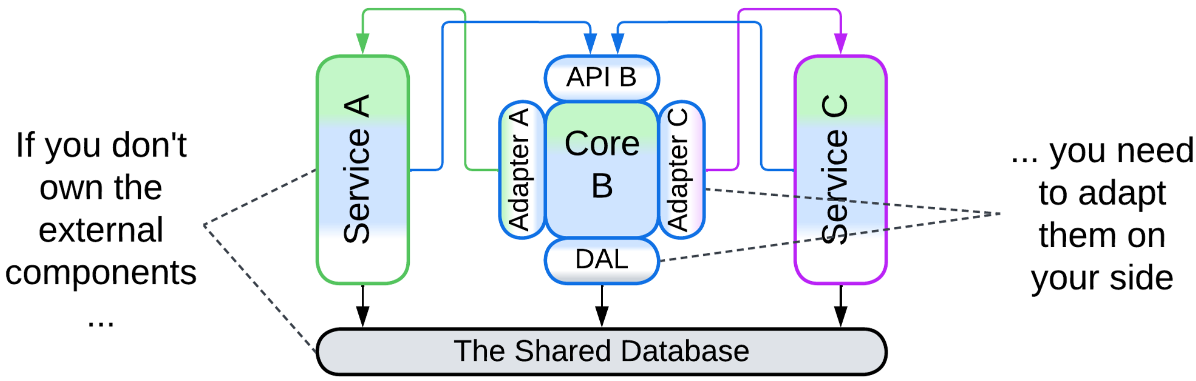
Adapters on the external component side



If your team owns the component adapted, the *adapter* may be placed next to it. That usually makes sense because a single domain message (in the terms of your business logic) tends to unroll into a series of calls to an external component. The fewer messages you send, the faster your system is.

This resembles *Sidecar* [DDS] and *Open Host Service* [DDD].

Adapters on the core side



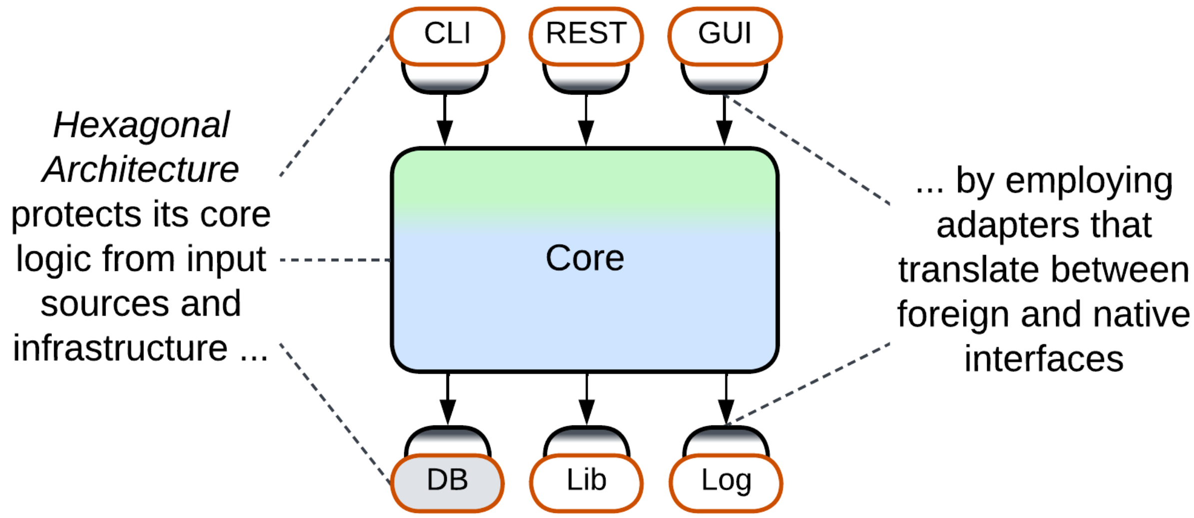
Sometimes you need to adapt an external service which you don't control. In that case the only real option is to place its *adapter* together with your *core* logic. In theory, the adapter can be deployed as a separate component, maybe in a *Sidecar* [DDS], but that may slow down communication.

This approach resembles *Ambassador* [DDS] and *Anticorruption Layer* [DDD].

Examples – Hexagonal Architecture

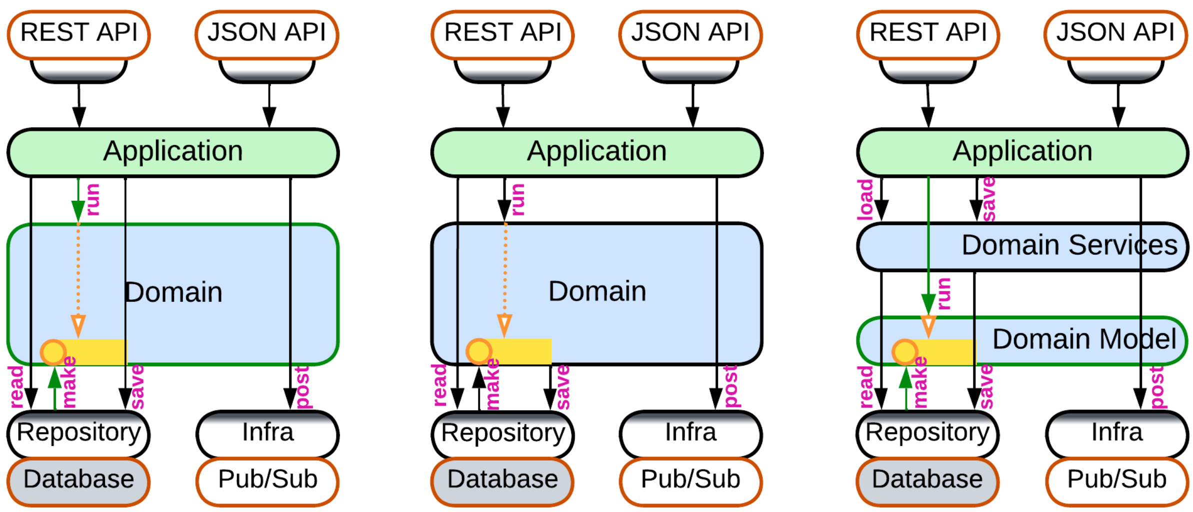
Hexagonal Architecture protects business logic from all its dependencies. It is simple and unambiguous. It does not come in many shapes:

Hexagonal Architecture, Ports and Adapters



Just like *MVC* it is based on, the original *Hexagonal Architecture* (*Ports and Adapters*) does not care about the contents or structure of its *core* – it is all about isolating the core from the environment. The core may have layers or modules or even plugins inside, but the pattern has nothing to say about them.

DDD-Style Hexagonal Architecture, Onion Architecture, Clean Architecture



Puristic DDD Approach ***Pragmatic DDD Approach*** ***Onion Architecture***

As *Hexagonal Architecture* built upon the DDD's idea of isolating business logic with *Adapters*, it was quickly integrated back into DDD [LDDD]. However, as *Ports and Adapters* appeared later than the original DDD book, there is no universal agreement on how the thing should work:

- The cleanest way is for the *domain* layer to have nothing to do with the database – with this approach the *application* asks the *repository* (the database *adapter*) to create *aggregates* (domain objects), then executes its business actions on the aggregates and tells the repository to save the changed aggregates back to the database.
- Others say that in practice the logic inside an aggregate may have to read additional information from the database or even depend on the result of persisting parts of the aggregate. Thus it is the aggregate, not the application, which should save its changes, and the logic of accessing the database leaks into the domain layer.
- *Onion Architecture*, one of early developments of *Hexagonal Architecture* and DDD, always splits the domain layer into a *domain model* and a *domain services*. The *domain model* layer contains classes with business data and business logic, which are loaded and saved by the *domain services* layer just above it. And the upper *application services* layer drives use cases by calling into both domain services and domain model.
- There is also *Clean Architecture* which seems to generalize the approaches above without delving into practical details – thus the way it saves its aggregates remains a mystery.

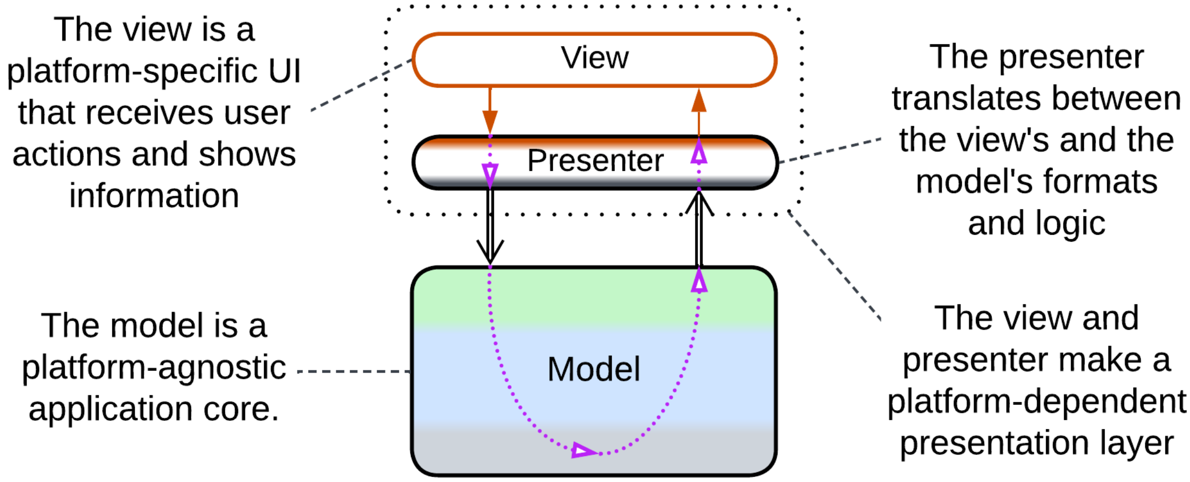
Examples – Separated Presentation

Separated Presentation protects business logic from a dependency on *presentation* (interactions with the system's user via a window, command line, or web page). There is a great variety of such patterns, commonly known as *Model-View-Controller* (MVC) alternatives. They are derived from *Hexagonal Architecture* by omitting every component not directly involved in user interactions and make three structurally distinct groups:

- Bidirectional flow – the *view* (user-facing component) both receives input and produces output and there is often an explicit *adapter* between it and the main system, resulting in *Layers*.
- Unidirectional flow – the *controller* receives input while the *view* produces output, forming a kind of *Pipeline*.
- Hierarchical with multiple *models*, discussed in the *Hierarchy* chapter.

All of them aim at making the business logic presentation-agnostic (thus cross-platform and developed by an independent team), but differ in their complexity, flexibility and best use cases.

Model-View-Presenter (MVP), Model-View-Adapter (MVA), Model-View-ViewModel (MVVM), Model 1 (MVC1), Document-View



MVP-style patterns pass user input and output through one or more presentation *layers*. Each pattern includes:

- *View* – the interface exposed to users.
- An optional intermediate layer that translates between the *view* and *model*. It is the component which differentiates the patterns, both in name and function.
- *Model* – the whole system's business logic and infrastructure, now independent from the method of presentation (CLI, UI or web).

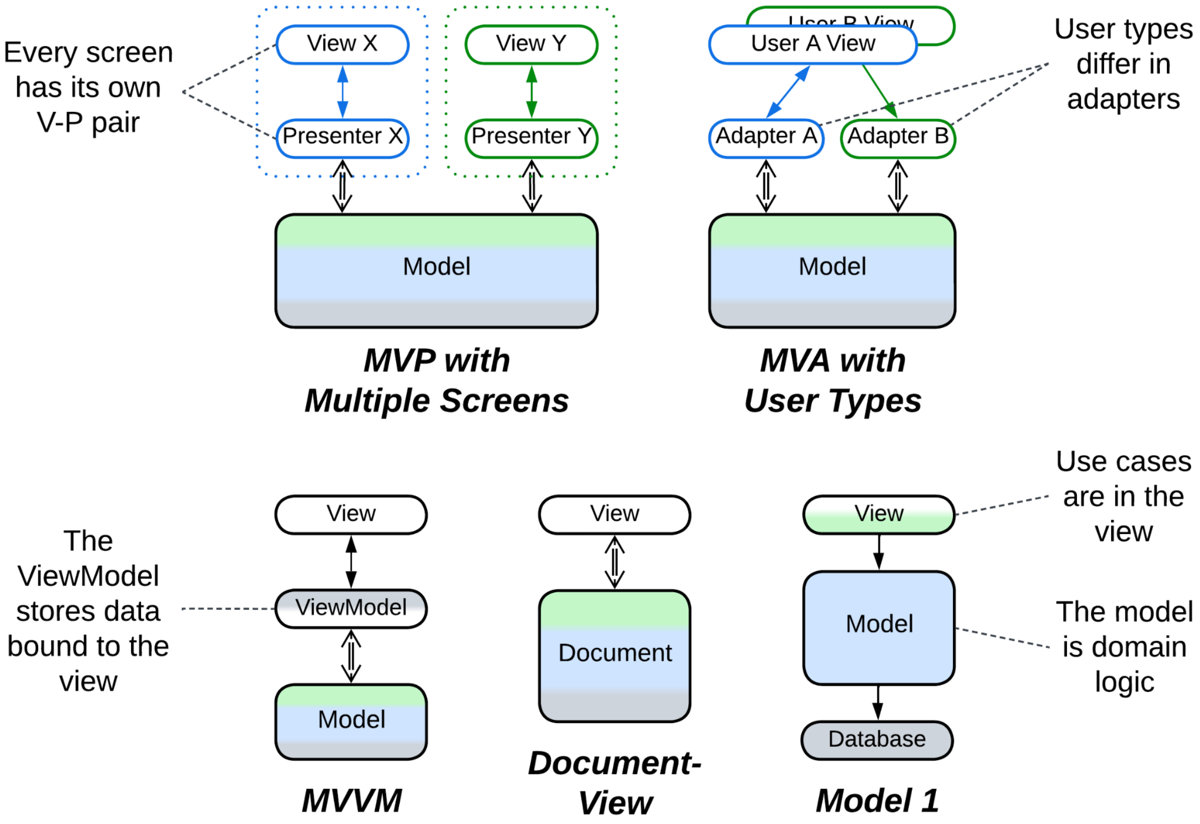
Document-View [POSA1] and *Model 1* (*MVC1*) skip the intermediate layer and connect the *view* directly to the *model* (*document*). These are the simplest *Separated Presentation* patterns for UI and web applications, correspondingly.

In a *Model-View-Presenter* (*MVP*), the *presenter* (*Supervising Controller*) receives input from the *view*, interprets it as a call to one of the model's methods, retrieves the call's results and shows them in the view, which is often completely dumb (*Passive View*). A complex system may feature multiple view-presenter pairs, one per UI screen.

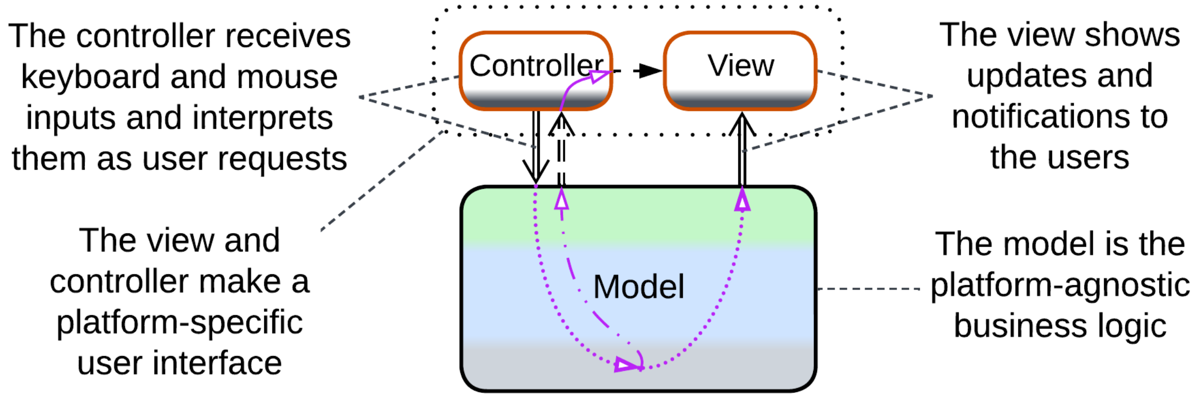
A *Model-View-Adapter* (*MVA*) is quite similar to *MVP*, but it chooses the adapter on a per session basis while reusing the view. For example, an unauthorized user, a normal user, and an admin would access the model through different adapters that would show them only the data and actions available with their permissions.

A *Model-View-ViewModel* (*MVVM*) uses a stateful intermediary (*ViewModel* or *Presentation Model*) which resembles a *Response Cache*, *Materialized View*, *Reporting Database* or *Read Model* of *CQRS* – it stores all the data shown in the view in a form which is convenient for the view to bind to. Changes in the view are propagated to the ViewModel which translates them into requests to the underlying application (the true model). Changes in the model (independent or resulting from user actions) are propagated to the ViewModel and, eventually, to the view.

All those patterns exploit modern OS or GUI frameworks' widgets which handle and process mouse and keyboard input, thus removing the need for a separate (input) *controller* (see below).



Model-View-Controller (MVC), Action-Domain-Responder (ADR), Resource-Method-Representation (RMR), Model 2 (MVC2), Game Development Engine



When your presentation's input and output diverge (raw mouse movement vs 3D graphics in UI, HTTP requests vs HTML pages in websites), it makes sense to separate the presentation layer into dedicated components for input and output.

Model-View-Controller (*MVC*) [POSA1, POSA4] allows for cross-platform development of hand-crafted UI applications (which was necessary before universal UI frameworks emerged) by abstracting the system's *model* (its main logic and data, the *core* of *Hexagonal Architecture*) from its user interface containing platform-specific *controller* (input) and *view* (output):

- The *controller* translates raw input into calls to the business-centric model's API. It may also hide or lock widgets in the view when the model's state changes.

- The *model* is the main UI-agnostic application which executes controller's requests and notifies the view and, optionally, controller when its data changes.
- Upon receiving a notification, the *view* reads, transforms, and presents to the user the subset of the model's data which it covers.

Each widget on the screen [may have](#) its own model-view pair. The absence of an intermediate layer between the view and model makes the view heavyweight as it has to translate the model's data format into something presentable to users – the flaw addressed by the *MVP* (3-layered) patterns discussed above.

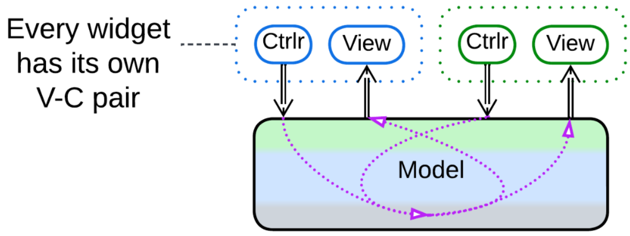
Both [*Action-Domain-Responder*](#) (*ADR*) and [*Resource-Method-Representation*](#) (*RMR*) are web layer patterns. An *action* (*method*) receives a request, calls into a *domain* (*resource*) to make changes and retrieve data and brings the results to a *responder* (*representation*) which prepares the return message or web page. *ADR* is technology-agnostic while *RMR* is HTTP-centric.

[*Model 2*](#) (*MVC2*) is a similar pattern from the Java world with integration logic [implemented](#) in the controller.

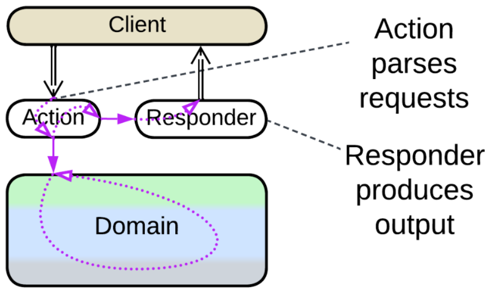
A [*game development engine*](#) creates a higher-level abstraction over input from mouse / keyboard / joystick and output to sound card / GPU while more powerful engines may also [model physics](#) and character interactions. The role is quite similar to what the original *MVC* did, with a couple of differences:

- Games often have to deal with the low-level and very chatty interfaces of hardware components, thus the input and output are at the bottom side of the system diagram.
- The framework itself makes a cohesive layer, becoming a kind of [*Microkernel*](#).

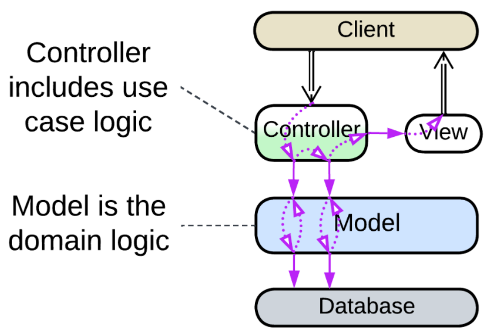
Another difference is that while *MVC* provides for changing target platforms by rewriting its minor components (*view* and *controller*), you are very unlikely to change your game framework – instead, it is the framework itself that makes all the platforms look identical to your code.



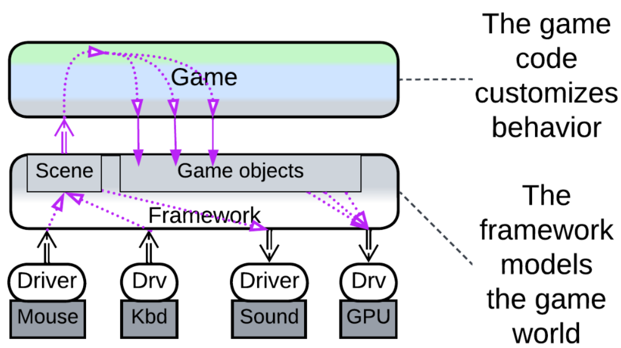
MVC with Multiple Widgets



ADR / RMR



Model 2



Gamedev Engine

Summary

Hexagonal Architecture isolates a component’s business logic from its external dependencies by inserting *adapters* between them. It protects from *vendor lock-in* and allows for late changes of third-party components but requires all the APIs to be designed before programming can start and often hinders performance optimizations

Appendix B. Books referenced.

DDD – Domain-Driven Design: Tackling Complexity in the Heart of Software. *Eric Evans. Addison-Wesley (2003).* (Most of these patterns are also well-described in [LDDD])

DDIA – Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O'Reilly Media, Inc. (2017).*

DDS – Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. *Brendan Burns. O'Reilly Media, Inc. (2018).*

DEDS – Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka. *Ben Stopford. O'Reilly Media, Inc. (2018).*

EIP – Enterprise Integration Patterns. *Gregor Hohpe and Bobby Woolf. Addison-Wesley (2003).*

FSA – Fundamentals of Software Architecture: An Engineering Approach. *Mark Richards and Neal Ford. O'Reilly Media, Inc. (2020).*

GoF – Design Patterns: Elements of Reusable Object-Oriented Software. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley (1994).*

LDDD – Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. *Vlad Khononov. O'Reilly Media, Inc. (2021).* (Duplicates [DDD] thus I marked as [LDDD] only patterns not covered by [DDD])

MP – Microservices Patterns: With Examples in Java. *Chris Richardson. Manning Publications (2018)*.

PEAA – Patterns of Enterprise Application Architecture. *Martin Fowler. Addison-Wesley Professional (2002).*

POSA1 – Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. John Wiley & Sons, Inc. (1996).*

POSA2 – Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. *Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, Inc. (2000).*

POSA3 – Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. *Michael Kircher, Prashant Jain. John Wiley & Sons, Inc. (2004).*

POSA4 – Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. John Wiley & Sons, Ltd. (2007).*

POSA5 – Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt*. *John Wiley & Sons, Ltd. (2007).*

SAHP – Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. *Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani. O'Reilly Media, Inc. (2021).*

SAP – Software Architecture Patterns. *Mark Richards. O'Reilly Media, Inc. (2015).* (All of the architectures referenced here are in [FSA] as well, but [SAP] is free)

Appendix I. Index of patterns.

Clean Architecture
Cluster (group of services)
Combined Component
Command Query Responsibility Segregation (CQRS)
Composed Message Processor
Configuration File
Configurator
Container Orchestrator
Content Delivery Network (CDN)
Control (Orchestrator)
Controller (Orchestrator)
Coordinator (Saga)
CQRS View Database
Create on Demand (temporary instances)
Data Archiving
Data Domain
Data File
Data Grid (Space-Based Architecture)
Data Lake
Data Mesh
Data Product Quantum (DPQ)
Data Warehouse
Database Cache
Database Abstraction Layer (DBAL or DAL)
Dependency Inversion
Deployment Manager
Device Drivers
Direct Server Return
Dispatcher (Proxy)
Distributed Cache
Distributed Middleware
Distributed Monolith
Distributed Runtime (client point of view)
Distributed Runtime (internals)
Document-View
Domain (Uber definition for WSO2-style Cell)
Domain-Driven Design (layers)
Domain-Oriented Microservice Architecture (DOMA)
Domain Services (scope)
Domain-Specific Language (DSL)
Edge Service
Embedded systems (layers)
Enterprise Service Bus (ESB)
Enterprise Service Bus (as Middleware)
Enterprise Service Bus (as Orchestrator)
Enterprise Service-Oriented Architecture
Enterprise SOA
Event Collaboration

Mesh
Message Broker
Message Bus
Message Bus (as Middleware)
Message Translator (adapter)
Messaging Grid (Space-Based Architecture)
Microgateway
Microkernel
Microkernel (Plugins)
Microkernel Architecture (Plugins)
Microservices (architecture)
Microservices (scope)
Middleware
Model 1 (MVC1)
Model 2 (MVC2)
Model-View-Adapter (MVA)
Model-View-Controller (MVC)
Model-View-Presenter (MVP)
Model-View-ViewModel (MVVM)
Modular Monolith
Modulith
Monolith
Monolithic Service
Multitier Architecture
Multi-Worker
Nanoservices (API layer)
Nanoservices (as runtime)
Nanoservices (pipelined)
Nanoservices (scope)
Nanoservices (SOA)
Native Data Product Quantum (sDPQ)
Nearline System
Network of Networks
N-Tier Architecture
Offline System
Onion Architecture
Open Host Service
Operating System
Operating System Abstraction Layer (OSAL or OAL)
Orchestrated Saga
Orchestrated Services
Orchestrated Three-Layered Services
Orchestrator
Orchestrator of Orchestrators
Partition
Peer-to-Peer Networks
Persistent Event Log
Pipeline

Pipes and Filters
Platform Abstraction Layer (PAL)
Plug-In Architecture
Plugins
Polyglot Persistence
Ports and Adapters
Pool (stateless instances)
Presentation-Abstraction-Control (PAC)
Proactor
Process Manager
Processing Grid (Space-Based Architecture)
Proxy
Published Language
Query Service
Rate Limiter
Reactor (multi-threaded)
Reactor (single-threaded)
(Re)Actor-with-Extractors
Read-Only Replica
Read-Through Cache
Reflection (Plugins)
Remote Facade
Replica
Replicated Cache
Replicated Stateless Services (instances)
Reporting Database
Repository
Request Hedging
Response Cache
Resource-Method-Representation (RMR)
Reverse Proxy
Saga Engine (Microkernel)
Saga Execution Component
Saga Orchestrator
Scaled Service
Scatter-Gather
Scheduler
Script
Segmented Microservice Architecture
Separated Presentation
Service-Based Architecture (architecture)
Service-Based Architecture (shared database)
Service Layer (Orchestrator)
Service Mesh
Service Mesh (as Mesh)
Service Mesh (as Middleware)
Service of Services
Service-Oriented Architecture (SOA)

Services
Services of Services
Sharding (persistent slices of data)
Sharding Proxy
Shards
Shared Database
Shared Databases (Polyglot Persistence)
Shared Event Store
Shared File System
Shared Memory
Shared Repository
Sidecar
Software Framework (Microkernel)
Source-Aligned Data Product Quantum (Data Mesh)
Space-Based Architecture (as Mesh)
Space-Based Architecture (as Middleware)
Specialized Databases
Spine-Leaf Architecture
Stamp Coupling
Strategy (Plugins)
Stream Processing
Three-Tier Architecture
Tiers
Top-Down Hierarchy
Transaction Script
Virtualizer
Work Queue
Workflow System
Workflow Owner (Orchestrator)
Wrapper Facade (Orchestrator)
Write-Behind Cache
Write-Through Cache