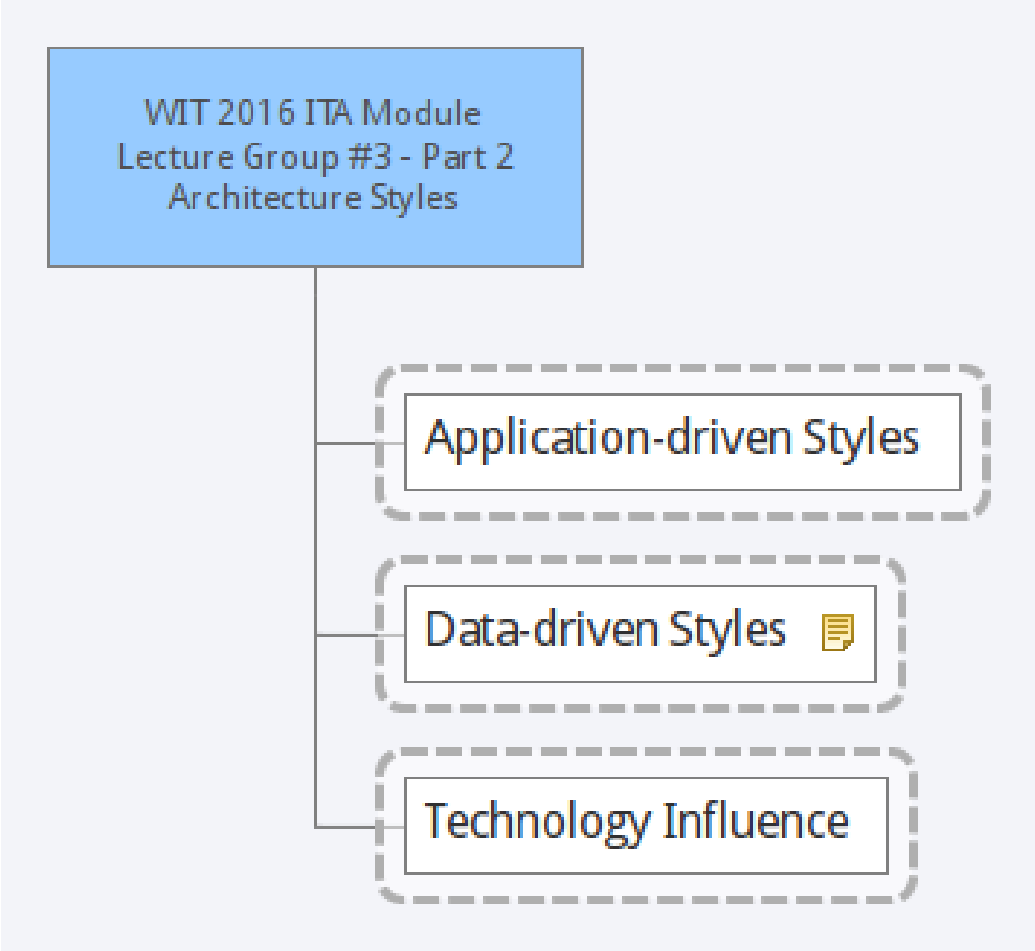
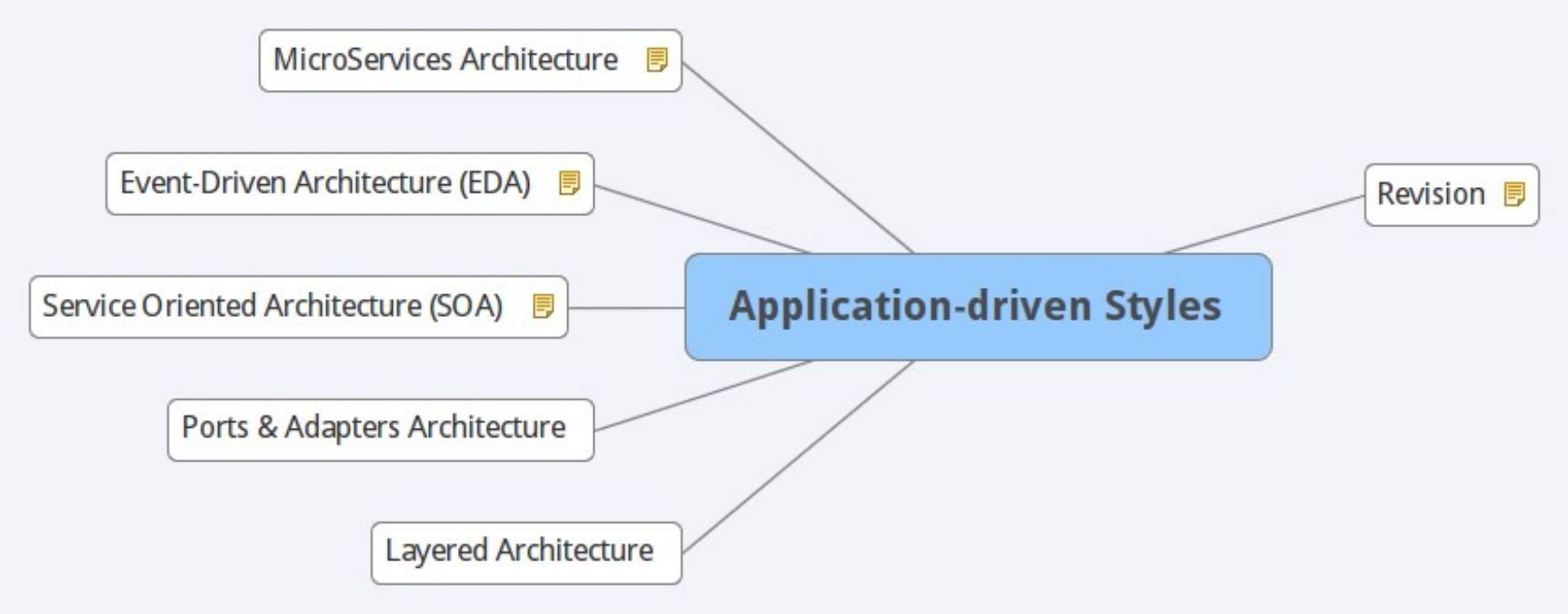
WIT 2016 ITA Module

Architecture Styles
Lecture Group #3 - Part 2

Lecture Group #3 - Part 2
Architecture Styles



Application-driven Styles

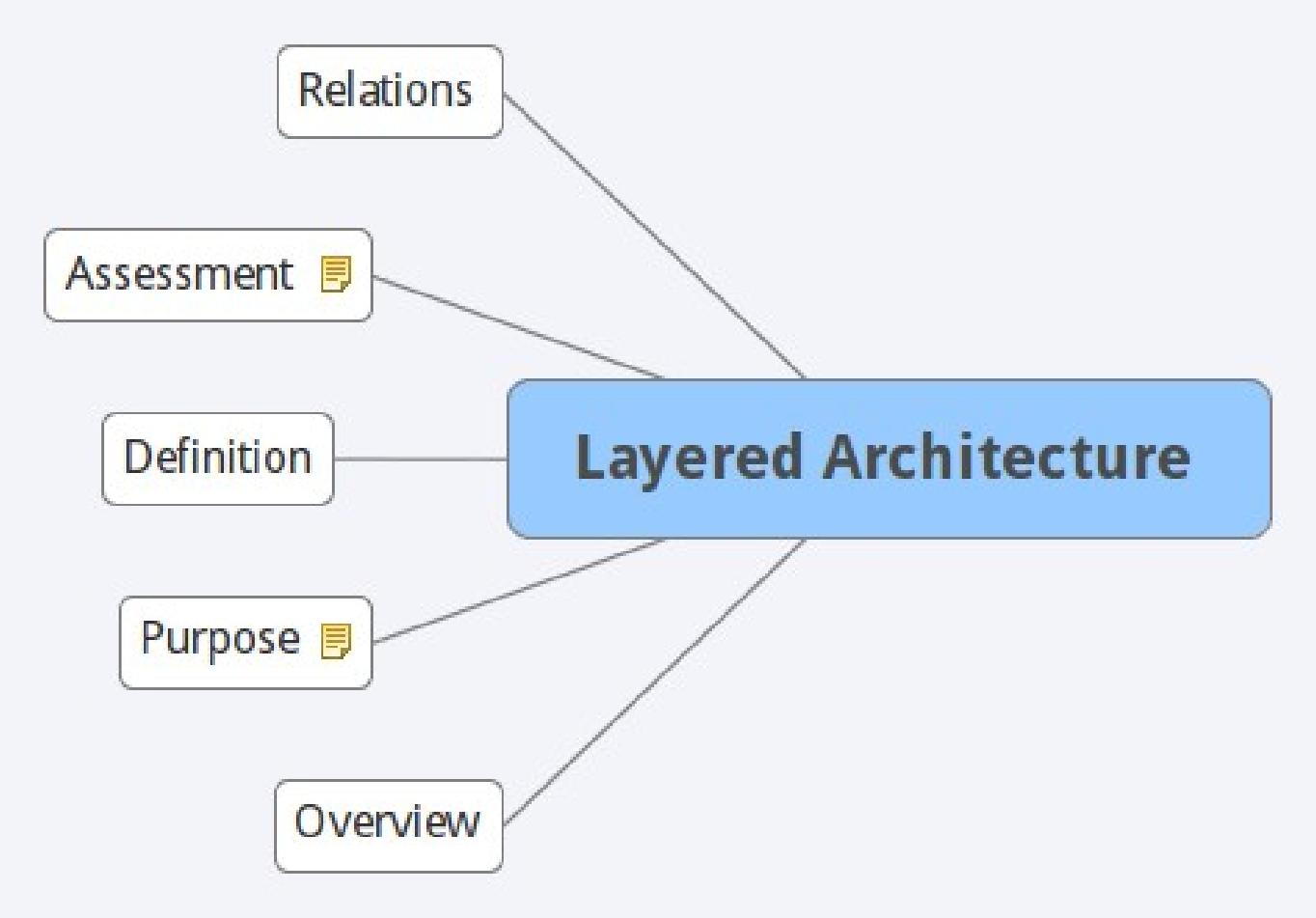


Revision

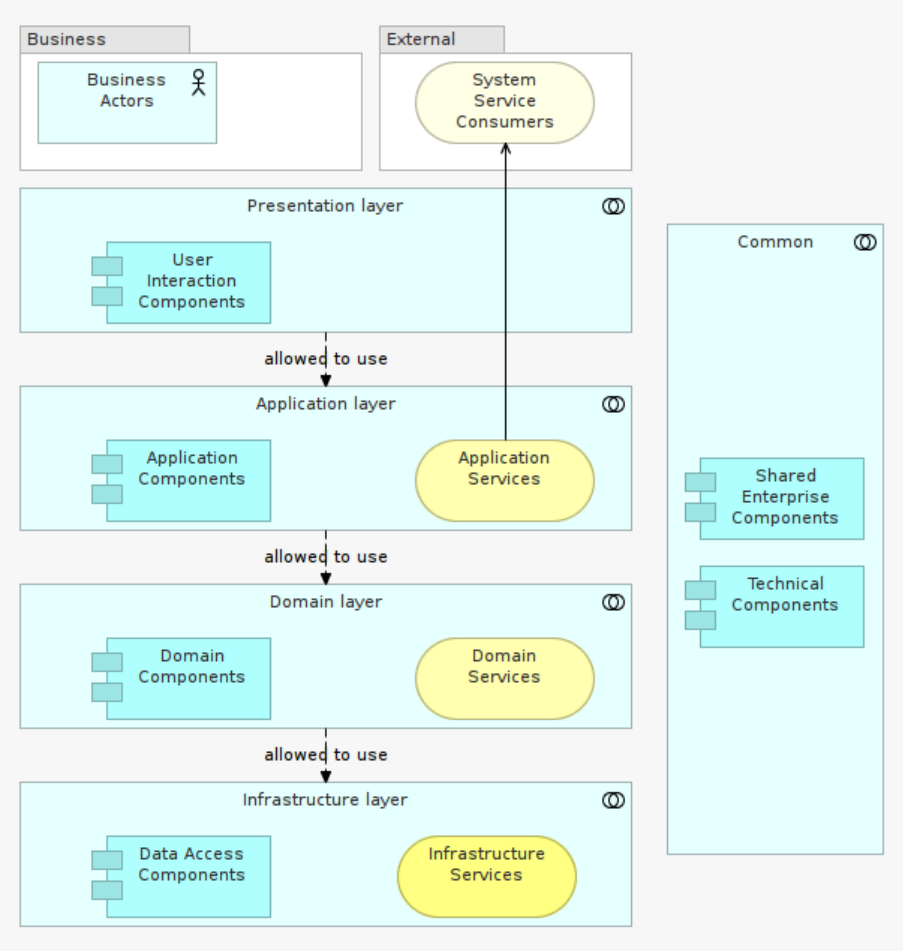
Component: A component is a unit of software that is independently replaceable and upgradeable.

That was short!

Layered Architecture



What does it look like?



Purpose

The purpose of the Layered Architectural Style is to rigorously separate the various concerns of an application into well-defined logical groupings of software components.

Each separate grouping has a distinct role and functionality: a responsibility that a set of similar types of components inherit and share.

These logical groupings are called LAYERS; large/complex enterprise applications are often partitioned into layers.

Structuring an application in layers help to differentiate between the different kinds of tasks performed by its constituting components, making it easier to create a design that supports re-usability.

Each logical grouping contains a number of discrete component types grouped into sub layers, with each sub layer performing a specific type of task.

The Layered style primarily aims to isolate the expression of the domain model (i.e. also referred as core business logic), and eliminate any dependency from infrastructure concerns, user interface experience, or even application logic that is not business logic.

In a layered architectural style, each layer is cohesive and only depend on the layers below.

To isolate components

Layers provide ISOLATION.

Change of one layer minimizes impact to others.

The more distant a layer is from another layer changing, the less impacted it will be.

The closest a layer is from another layer changing, the more likely it is to be impacted.

To separate concerns

Components in a layer collaborate to achieve operations of a same nature within the layer.

A component does not belong to the Presentation layer without behaving like a Presentation component.

A component does not belong to the Business Domain layer without behaving like a Domain component.

To distribute/allocate components

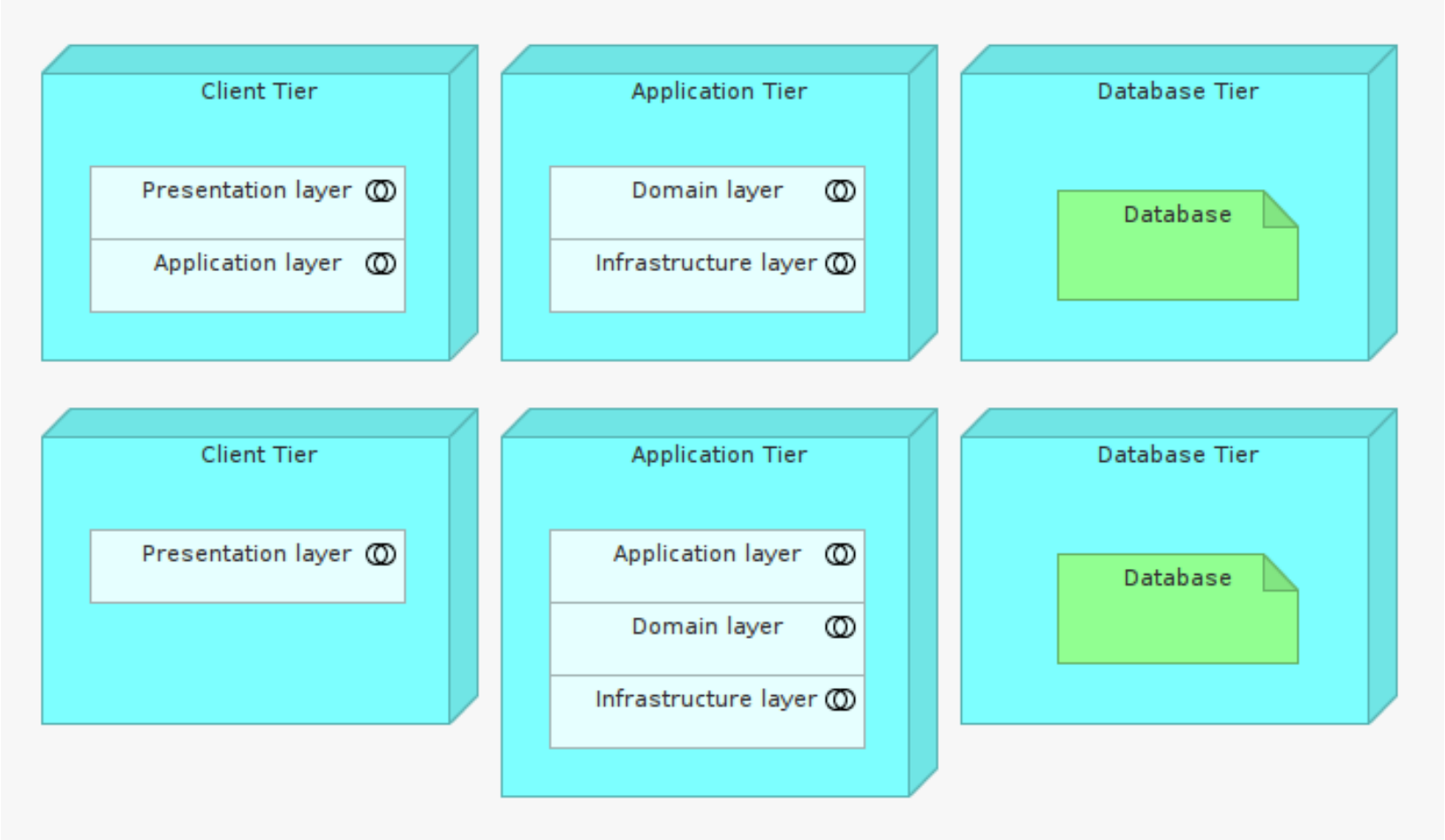
The layered architecture style emerged in a time where components aimed to be distributed so to load balance computing resources.

It provides a clear delineation between layers so to map onto server infrastructure locations, where certain technology or design decisions must be made.

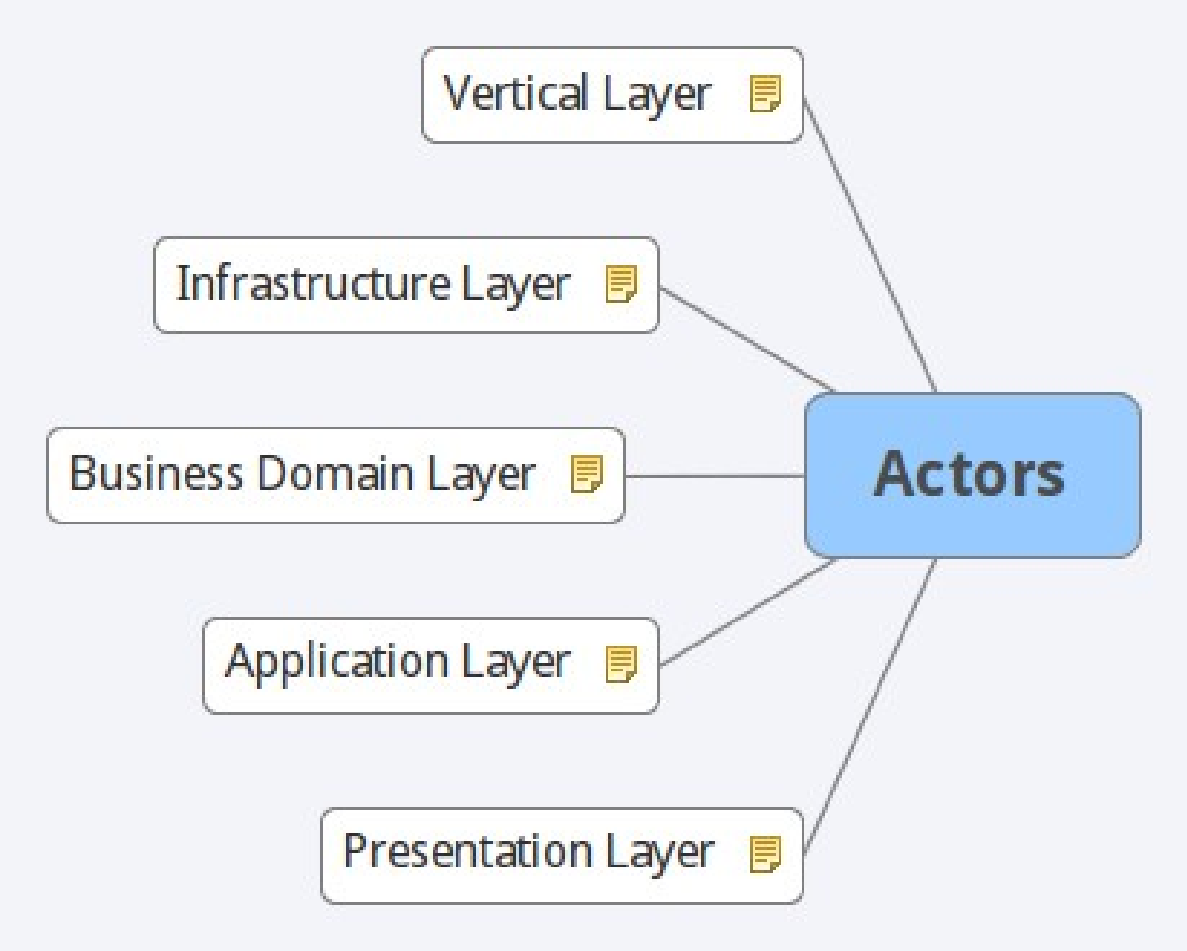
Layers can be mapped to physical tiers infrastructure, for example a three-tier infrastructure.

Layers of a design can be distributed across physical boundaries.

Examples of Physical Tier Allocation



Layers Description



Presentation Layer

This layer holds user-oriented functionality responsible for managing user interaction with the application.

It contain only components that addresses user view rendering and usage scenarios (task work flow, exception flows).

As such, components in this layer provide a bridge into the core business logic encapsulated in the business domain layer.

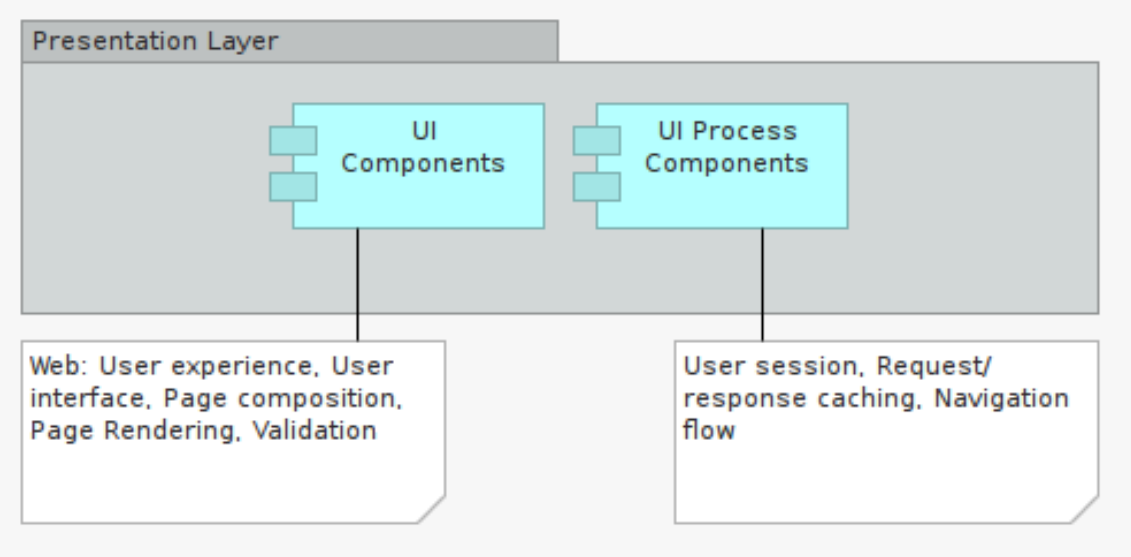
Data validations found in this layer are not the kinds that belong in the domain model – for example form validation doesn't include any business logic validation.

Purpose

The presentation layer contains the components that implement and display the user interface and manage user interaction.

This layer includes controls for user input and display, in addition to components that organize user interaction.

Responsibility



Related Patterns:

- Template View (Server Side Scripting)
- Model View Controller (MVC)
- Model View Presenter (MVP)

Related Frameworks:

- PhoneGAP
- AngularJS
- Struts

Application Layer

The Application Layer defines the application boundary and its set of available operations from the perspective of interfacing client layers.

It establishes a set of available operations that coordinate an application response answering requests from the presentation layer, or from the Application Layer of other inter-connected application / system.

Purpose

The Application Layer provides services to other layers of external applications (e.g. B2B integration).

This layer exposes the business domain functionality of a system via an API (Application Programming Interface).

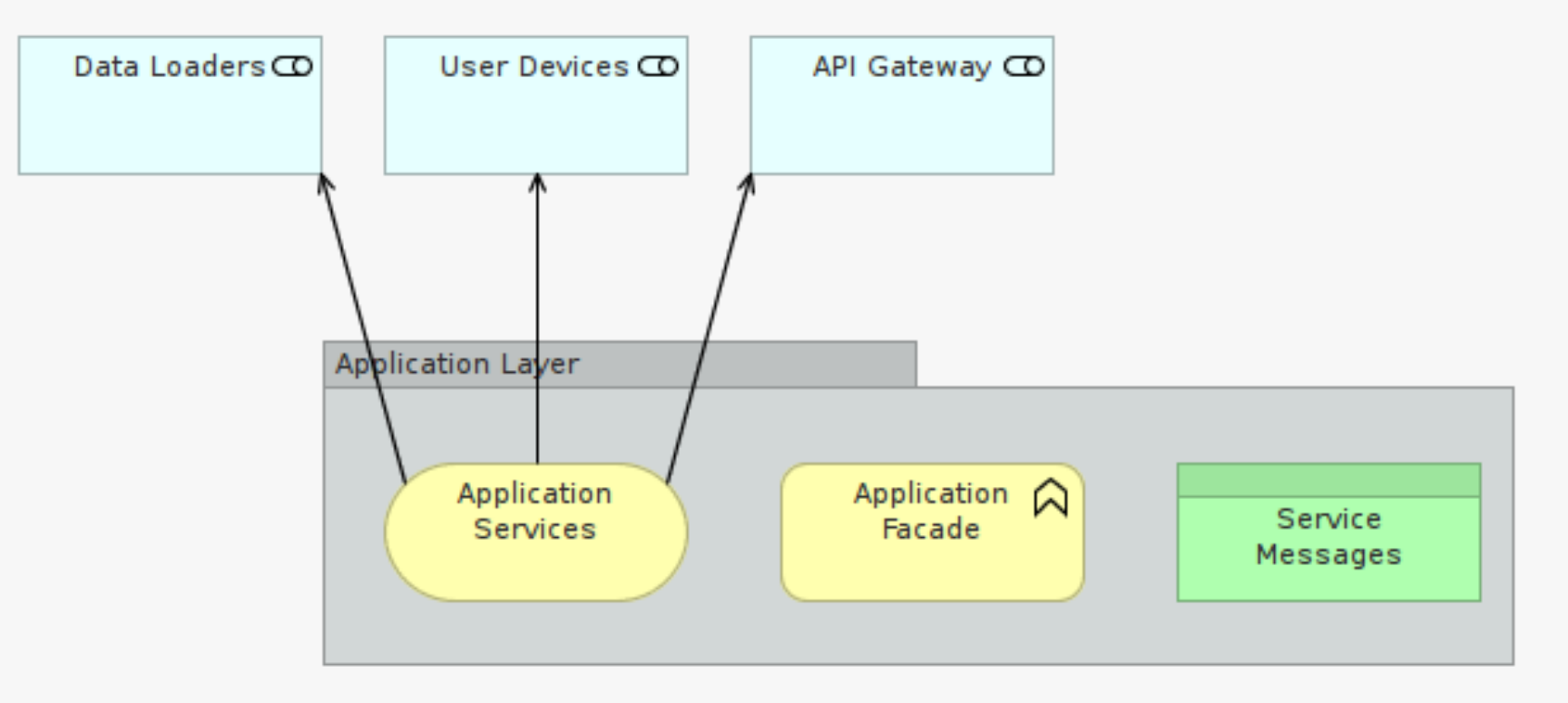
It allows different types of clients to use different channels to access the domain logic of an application.

Components in this layer are the direct clients of the domain model, though themselves possessing no business logic.

They remain very lightweight, coordinating and aggregating data operations performed against domain objects.

They are the primary means of expressing use cases or user stories on the model.

Responsibility



Responsibility

Components provide access application functions via a facade.

A facade combines multiple business operations into a single operation that makes it easier to use the business logic.

It reduces dependencies because external callers do not need to know interaction details of the domain components and the relationships between them.

Avoids coupling the sender of a request to its receiver by allowing more than one object to handle the request.

Encapsulate request processing in a separate command object with a common execution interface.

Related Patterns

Related Patterns:

- Facade
- Backend for Frontend / Aggregate
- Factory

Related Frameworks:

- Restful
- Jearsey
- Spring

Business Domain Layer

This layer implements the core functionality of the system, and encapsulates the core business logic of an application. It generally consists of components, some of which may expose service interfaces that other callers can use.

Purpose

Depending of the variant of Layered style implemented, components from the Presentation Layer or the Application Layer can pass data Business layer.

The application use this data to perform a business process, make a decision, or read/write application state.

This Layer groups business domain logic - i.e the work that an application needs to do for a business domain.

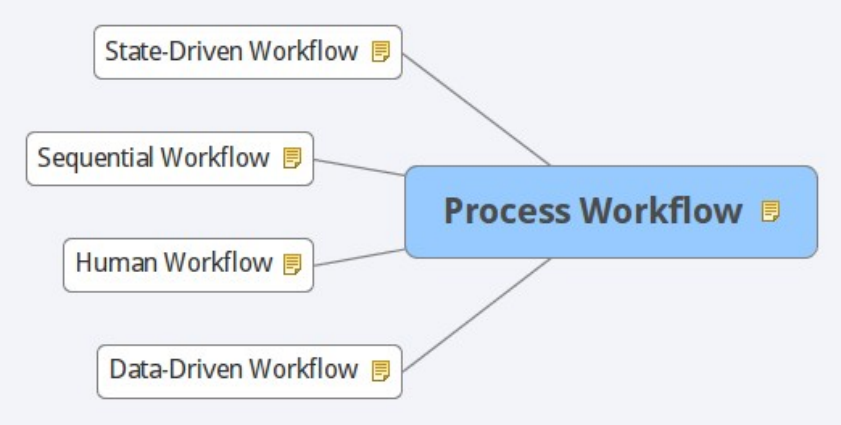
It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch, depending on commands received from above layers.

Responsibility

Actor: Process Workflow

Many business processes involve multiple steps that must be performed in the correct order, and may interact with each other.

Process workflow components define and coordinate long running, multistep business processes, for example a sequence of transactional screens accomplishing a task, requiring further operations or validation at each step.



Actor: Decision Logic

Business logic is defined as any application logic that is concerned with:

(1.) the retrieval, processing, transformation, and management of application data

(2.) the execution of business logic rules

(3.) and ensuring data consistency and validity.

To maximize reuse opportunities, business logic components should not contain any behavior or application logic that is specific to a use case or user story.

Actor: Domain Entity

Domain Entities, encapsulate data necessary to represent real business world elements, such as Customers or Orders in an Enterprise application.

These components store data values and expose them to client components; manage data used by the application; and provide stateful programmatic access to the business data and related functionality.

These components can validate the data contained within the entity and encapsulate domain logic to ensure data consistency / integrity.

For example, an Entity translator component transforms request message data types to business types, and reverses the transformation for responses.

Table Module. A single component that handles the business logic for all rows in a database table or view.

Related

Patterns:

- Domain Model: An object model of the business domain that incorporates both behavior and data.

- Data Mapper. Implement a mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.

- Data Transfer Object. An object that stores the data transported between processes, reducing the number of method calls required.

Related Frameworks:

- BRE/Drools

- Workflow/JPBM

Infrastructure Layer

This layer provides access to data hosted within the boundaries of the system, and data exposed by other networked systems through infrastructure services (e.g. SFTP). The data layer exposes generic interfaces that the components in the business domain layer can consume.

For most enterprise applications this is a database that is primarily responsible for storing persistent data, however this layer also communicates with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems, etc.

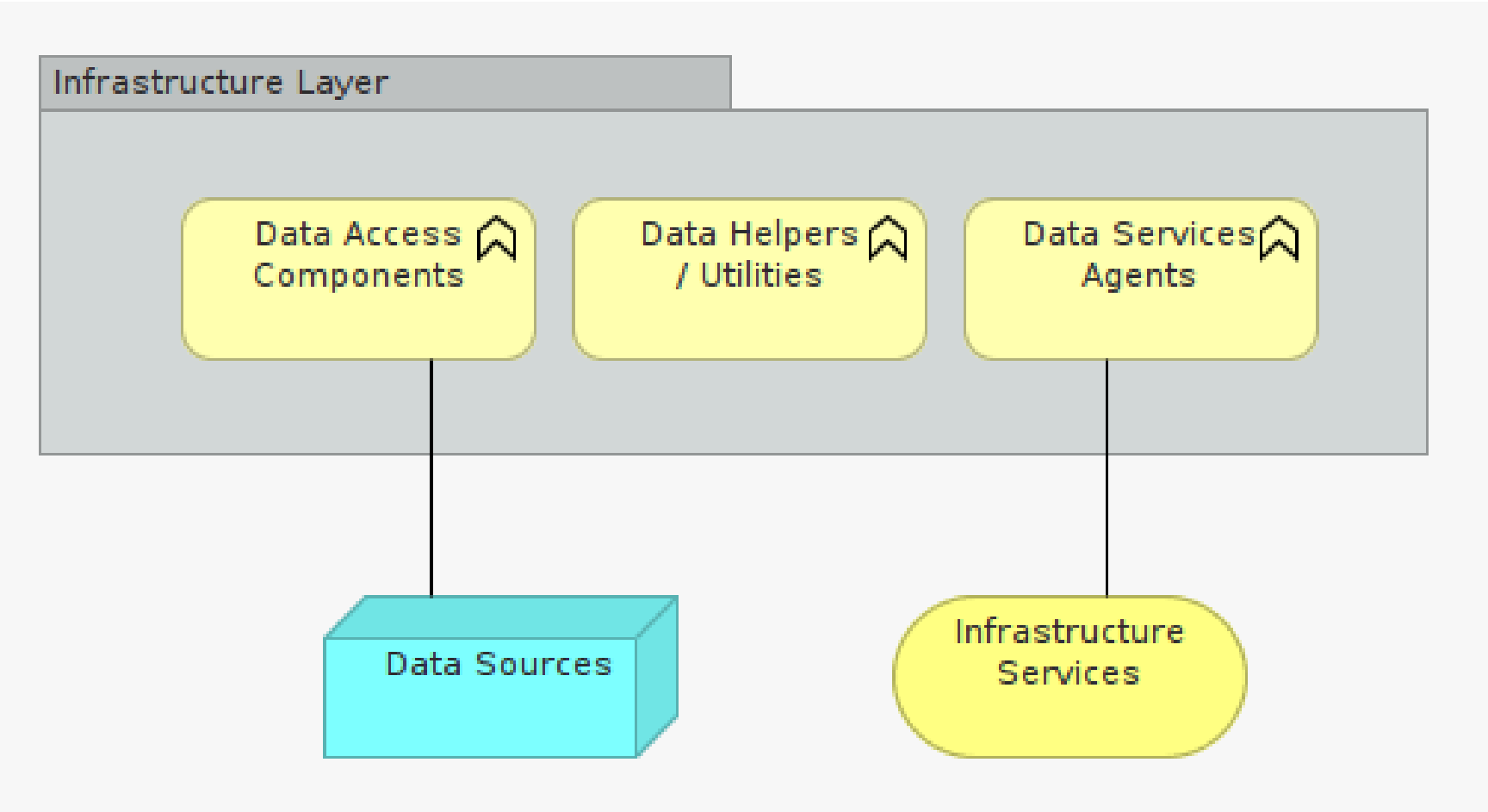
Purpose

The infrastructure layer isolate components bridging core domain model components to a server infrastructure. Responsibilities like data persistence (on file, in databases) and messaging mechanisms reside in this layer.

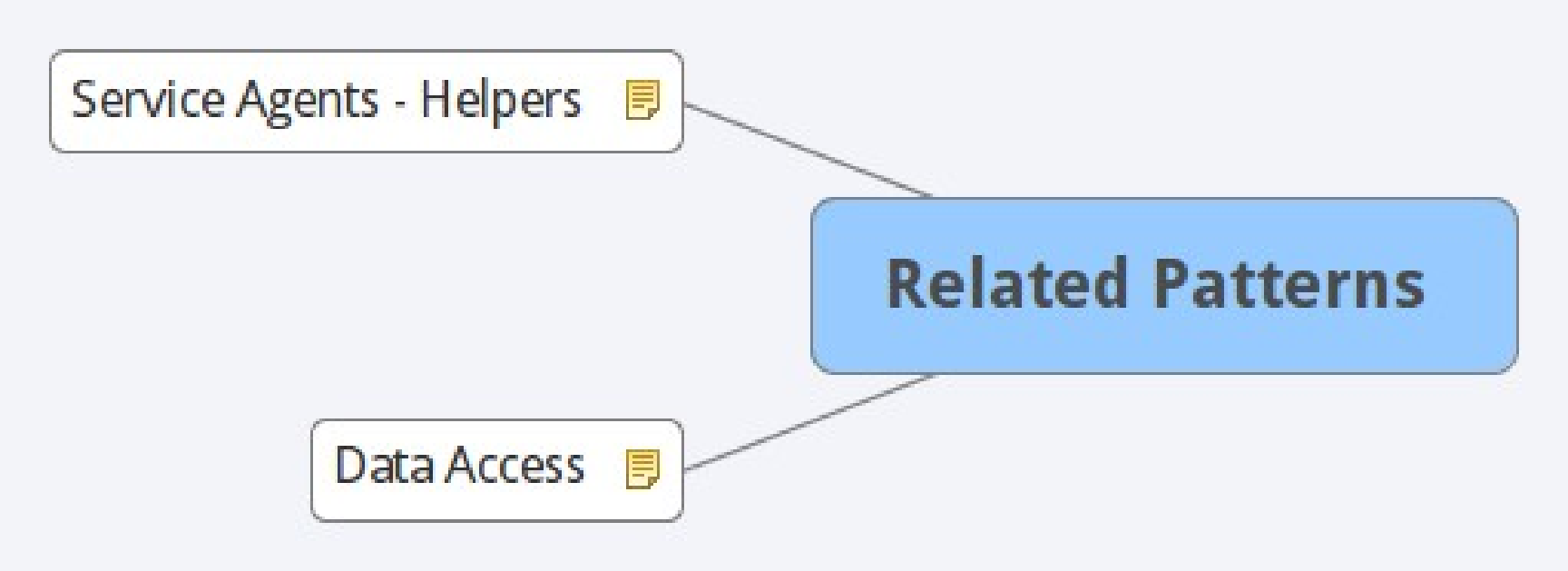
Messages may include those sent by enterprise messaging middleware systems or more basic e-mails (SMTP).

In general any technical component and/or framework that provide low-level services for the application is grouped in the infrastructure layer. The higher-level Layers couple to the lower-level components of the infrastructure layer to reuse the technical facilities provided.

Responsibility



Related Patterns



Actor: Data Access

Object Relational Mapping (ORM)

Row Data Gateway - An object that acts as a gateway to a single record in a data source.

Table Data Gateway - An object that acts as a gateway to a table or view in a data source and centralizes all of the select, insert, update, and delete queries.

Table Module - A single component that handles the business logic for all rows in a database table or view.

Active Record - Include a data access object within a domain entity.

Query Object - An object that represents a database query, for example a Stored Procedures or an SQL statement.

Repository - An in-memory representation of a data source that works with domain entities.

Actor: Service Agents - Helpers

Batch

Partition multiple large batch jobs to run concurrently. Allow multiple batch jobs to run in parallel to minimize the total processing time.

Transactions

Capture Transaction Details. Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.

Transaction Script. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Resources Lock. Ensure that changes made by one session do not conflict with changes made by another session. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.

Helpers / Utilities components

E.g. Markup Translations (XML <=> JSON)

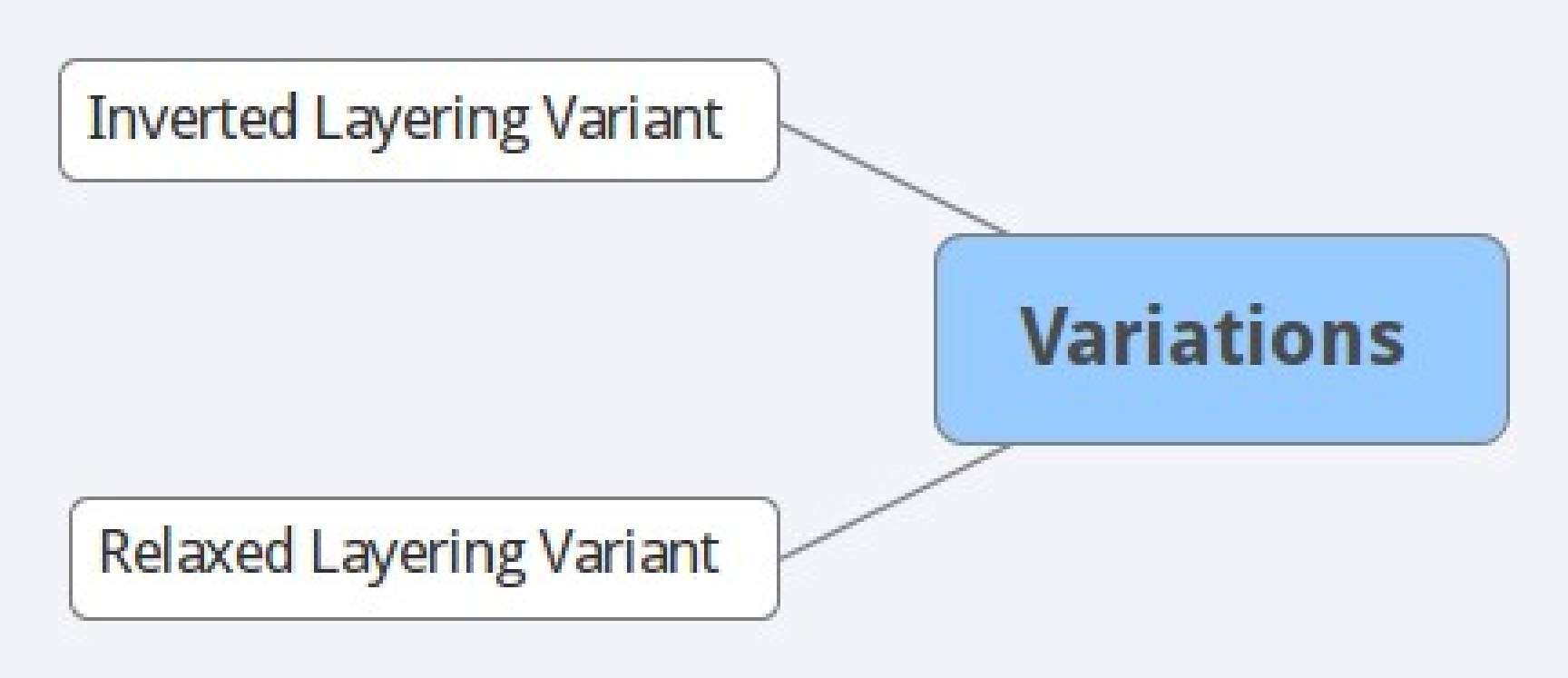
Related Development Frameworks

Hibernate

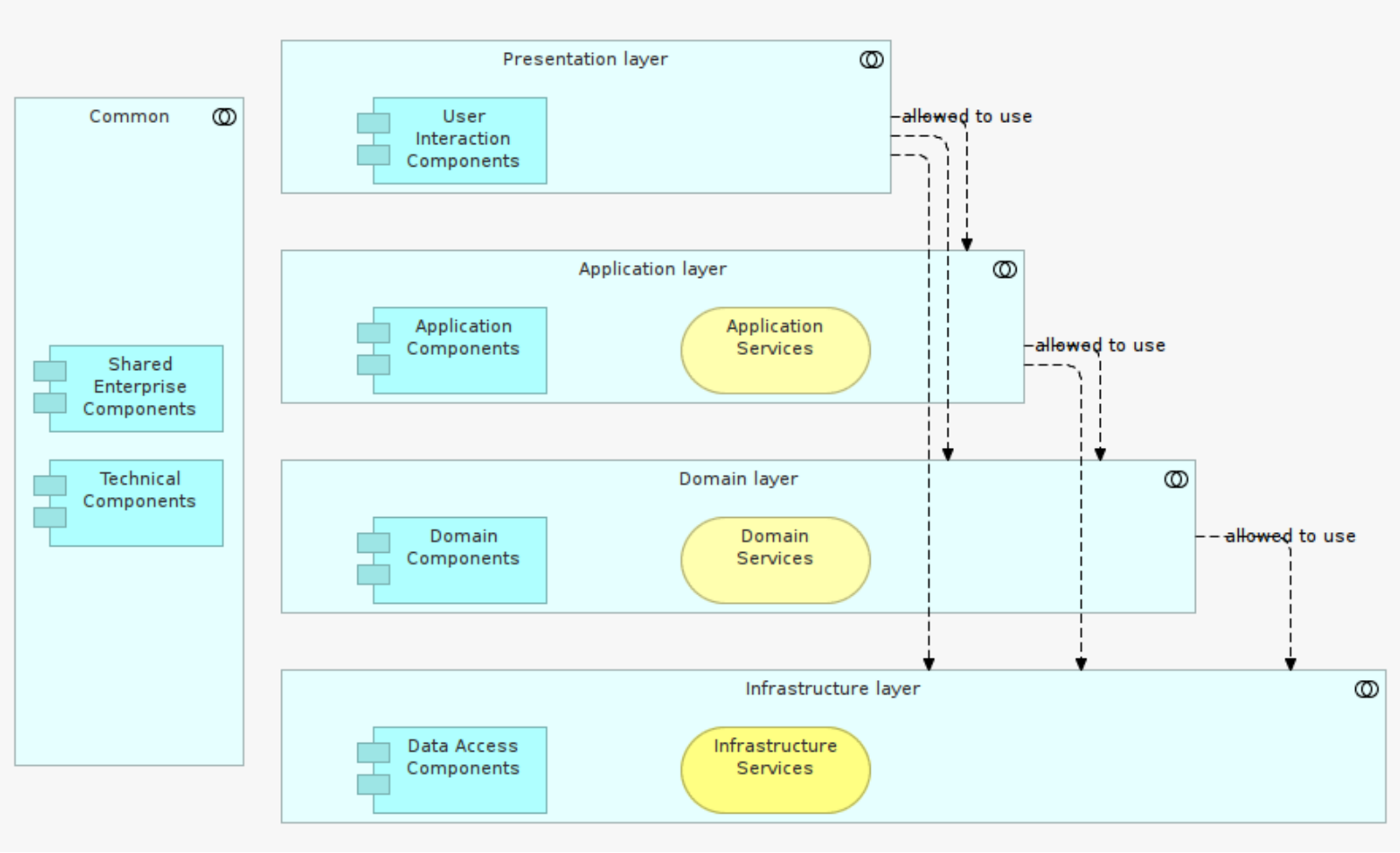
Vertical Layer

If this layer is impacted, changes occurs all layers depending on technical / utility classes defined in the Vertical layer – e.g. security, logging.

Variations



Relaxed Layering Variant



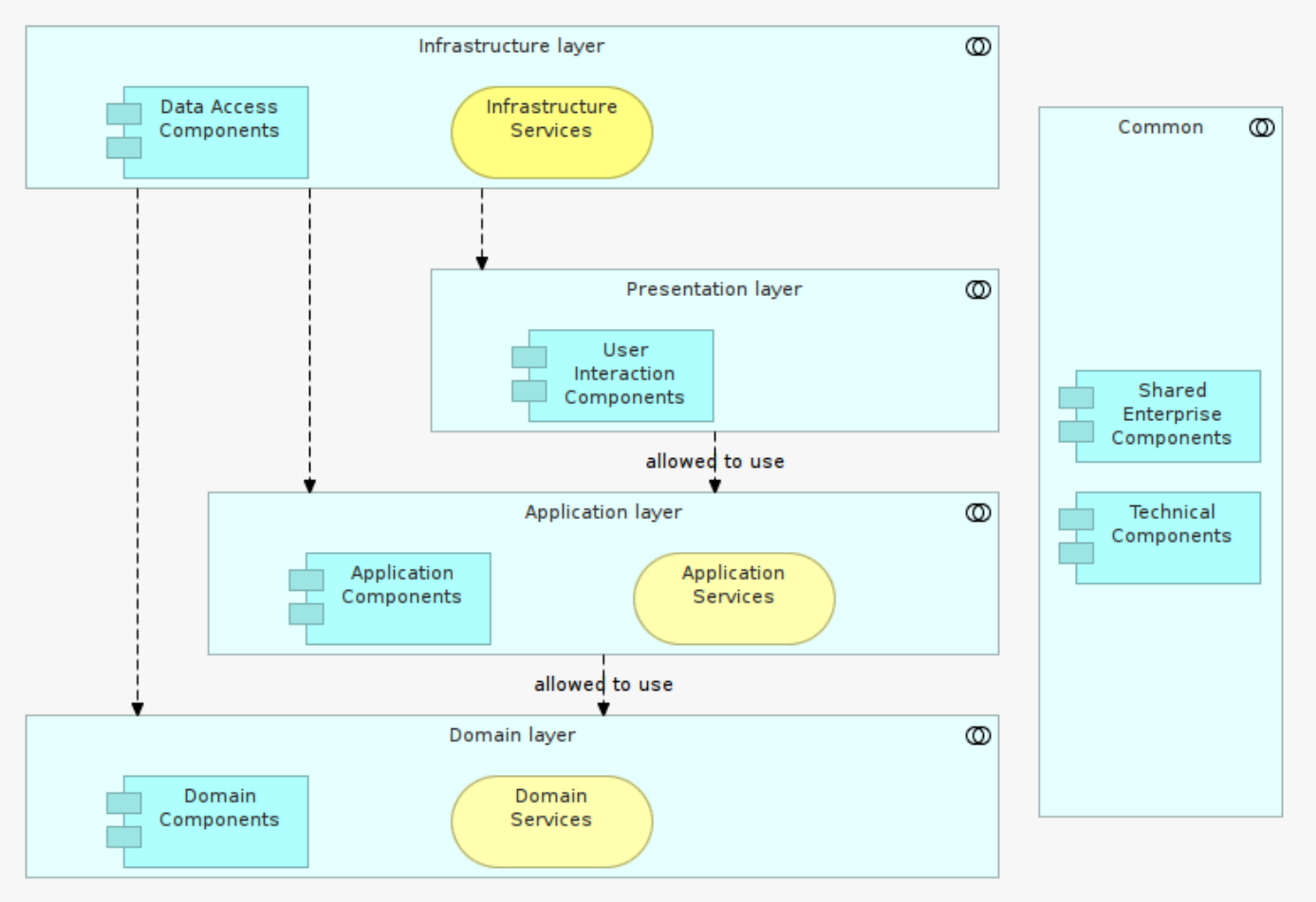
Purpose

For simpler applications, all layers of an architecture may not be required. It is the role of an architect to select the appropriate layers of her design and the type communication between each.

For example, a presentation layer may communicate directly with an infrastructure layer through its services sub-layer layer.

Systematically enforcing the use of all intermediary layers would make of architecture style a system of “pass-through” forcing the creation of boiler plate code.

Inverted Layering Variant



Purpose

We move the Infrastructure Layer above all others, enabling it to implement interfaces for all Layers below.

Above layers implement interface abstractions defined in lower layers.

A component that provides low-level services (e.g. Infrastructure) depends on interfaces defined by higher-level components (e.g. Domain).

Dependency Injection / Inversion of Control patterns enable this architectural style variant, providing ways to acquire the implementations.

Interactions

A layer should not expose internal details on which another layer could depend. When defining an interface for a layer, the primary goal is to enforce loose coupling between layers. Doing so, each layer may couple only to itself and below. The interface to a layer should be designed to minimize dependencies by providing a public API that hides details of the components within the layer.

Rules for Interaction between Layers

Pass-through Request: Request has to go through all layers back and forth.

Top-down Interaction: Higher level layers can interact with layers below, but a lower level layer should never interact with layers above, doing so avoiding any risk of circular dependencies between layers. Event monitoring can be used make components in higher layers aware of changes in lower layers without introducing dependencies.

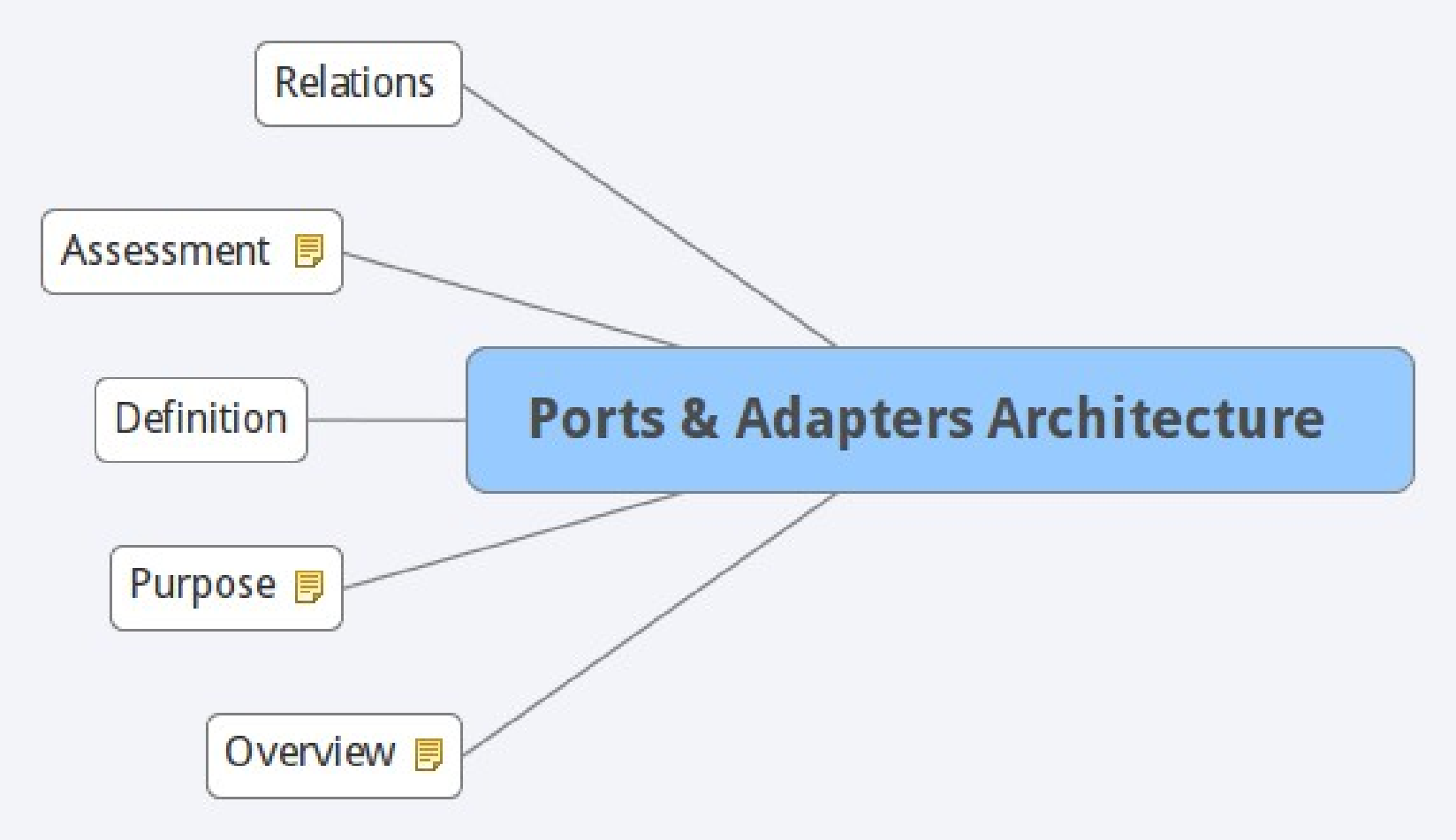
Strict Interaction: Each layer must interact with only the layer directly below. This rule will enforce strict separation of concerns where each layer knows only about the layer directly below. The benefit of this rule is that modifications to the interface of the layer will only affect the layer directly above.

Loose Interaction: Higher level layers can bypass layers to interact with lower level layers directly. This improves performance but will also increase dependencies, i.e. modification to a lower level layer can affect multiple layers above.

Assessment

Executability (Development, Testability)
Over-engineering of proxies propagating down requests to deeper layers. Team Specialization loosing track of the 'big picture', mixing up the business value of the solution with IT concerns, conflict of interest. Difficult to understand end to end, overwhelming. Handles complexity well overall. Style promotes good testability and handles complexity well.
Performance
Communication between layers can become costly over time because of abstraction and data exchange overhead.
Topology (Deployment / Allocation)
Requires entire redeployment of the entire application via an integrated release. Referred as "monolithic" because of coarse grain deployment.
Scalability
Can physically scale at a price, by adding more power to a single server, such as more memory, or by adding clusters.
Maintainability / Flexibility (Ease of change, Coupling)
Multiple teams are called into to perform one change, but layer isolation ensures proper coordination of efforts. If architecture well documented and layering isn't violated, change can be managed via modularity. However, ease of change killed by monolithic deployment of new releases.
Availability / Reliability (Resilience, Failures recovery)
Multitude of vendor and open sourced solutions for static libraries analysis, and dynamic monitoring of runtime modules.
Security
Relies on container-based security by default, and can be augmented by SSO/SAML.
Portability
In Java EE, can be migrated to different middleware platform with little reconfiguration, but overall it is Java EE or nothing: Can not employ diverse programming languages to answer specific specialized needs.

Ports & Adapters Architecture



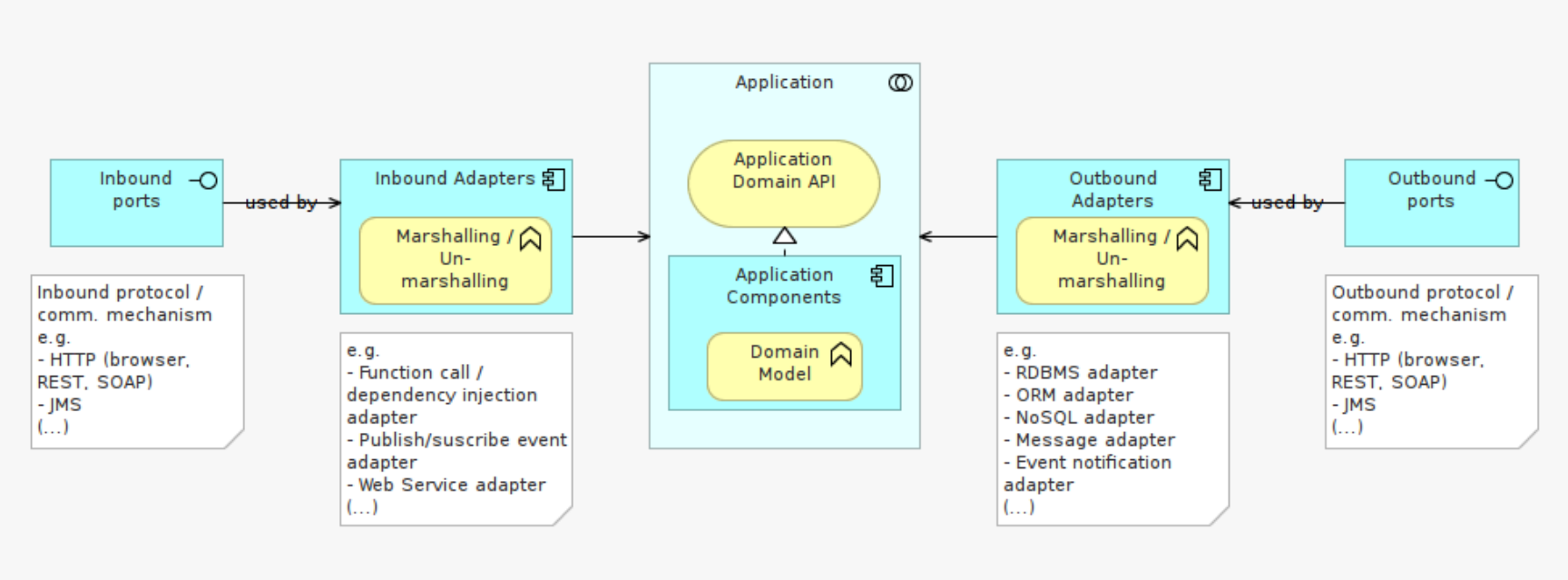
Overview

The purpose of the Ports & Adapters architectural style is to allow many disparate "clients" to interact with an application in consistent way (i.e. regardless if the client in question is a database, a user interface, another system).

Client inputs are transformed by inbound Adapters, so that the internal core application API can be invoked. Application core logic outputs are then transformed using outbound adapters.

Adapters provide a level of abstraction reducing coupling with the environment. Inbound and outbounds mechanisms can be replaced with minimum impact on core application components. For example, an output mechanism can be swapped from persistence to messaging without the application API to be changed.

What does it look like?



Purpose

This architecture style allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

To improve transparency

This style differs from layered architecture styles where business domain logic can too often hide behind several layers of abstraction. The Ports and Adapters style, while providing some level of TECHNICAL abstraction aims to provide ease of access to the core application and its domain model.

To minimize footprint

This style aims to simplify the overall design of complex application suites. Ports and Adapters can be used as a thin wrapper surrounding a legacy platform, or simply permits business-to-business service applications of the same architectural style to use each other, for example over the web in the context of a Service Oriented enterprise architecture landscape.

To separate concerns

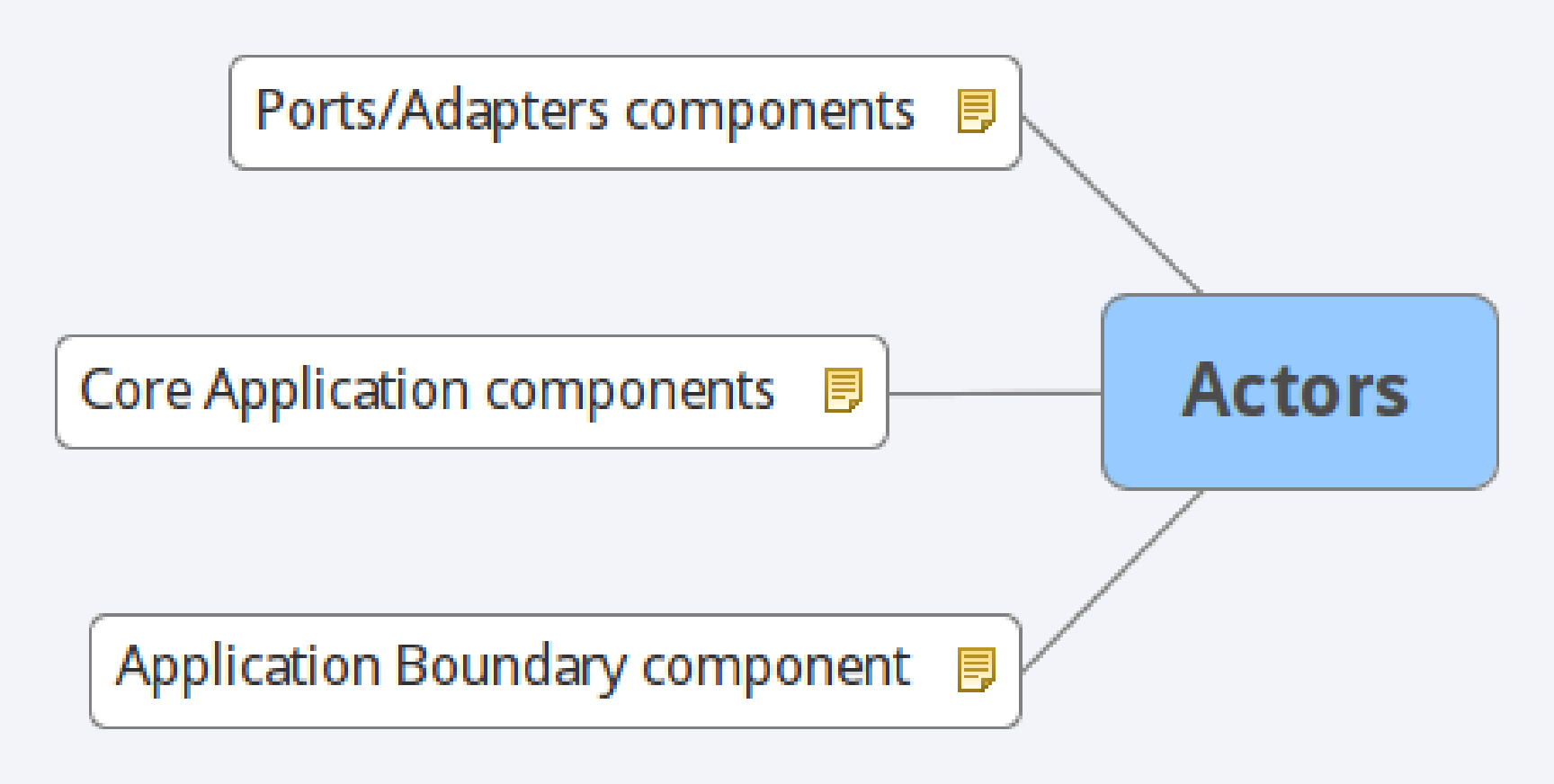
This architecture style primary goal is to keep business logic isolated in the application core. Ports and adapters subtract the need of inserting isolation layers within the architecture just to interface with external entities.

To isolate logic

The ultimate benefit of a ports and adapters implementation is the ability to run the application in a fully isolated mode. The Application connects with its surrounding via its API. Adapters provide means for an application to speak with its environment.

In any case, from the application’s perspective, if the environment changes from SQL database to Graph, the conversation across the API doesn't not change.

Actors



Actor: Application Boundary component

The application boundary is also referred as the use case (or user story) boundary. Use cases based on application functional requirements, not on the number of diverse clients or output mechanisms.

Functionality the application offers is available through an API (application programmed interface) or function call. Any number and type of clients may request through various Ports, but ultimately, each Adapter delegates to the application API.

The application may have multiple ports and adapters to communicate with its immediate environment. The application has a consistent way to interact with its adapters. The application doesn't know know about the nature of the things on the other side of the adapters.

Actor: Core Application components

The core an application consists of the algorithmic logic essential to its purpose. The core realize business use cases. The domain model is at heart of the application. When the application receives a request via its API, it uses the domain model to fulfill all requests involving the execution of business logic. The application API is published as a set of Application Services. Application Services are the direct client of the domain model, just as when using Layers. Changing the domain means changing the essence of the application, its services and has a ripple effect on ports and adapters.

Actor: Ports/Adapters components

A port identifies a purposeful conversation. There will typically be multiple adapters for any one port, for various technologies that may plug into that port. Typically, these might include a phone answering machine, a human voice, a touch-tone phone, a graphical human interface, a test harness, a batch driver, an http interface, a direct program-to-program interface, a mock (in-memory) database, a real database (perhaps different databases for development, test, and real use).

The number of ports isn't limited or set, however a number of generic categories have been identified over the years.

For each external device there is an “adapter” that converts the API definition to the signals needed by that device and vice versa.

Some of these adapters can be two-way dependencies.

The distinction between “primary “and” secondary “lies in who triggers or is in charge of the conversation.

Actor: Port

A port identifies a purposeful conversation. There will typically be multiple adapters for any one port, for various technologies that may plug into that port.

The number of ports isn't limited or set, however a number of generic categories have been identified over the years. For each external device there is an “adapter” that converts the API definition to the signals needed by that device and vice versa. Some of these adapters can be two-way dependencies.

Typically, these might include a phone answering machine, a human voice, a touch-tone phone, a graphical human interface, a test harness, a batch driver, an HTTP interface, a direct program-to-program interface, a mock (in-memory) database, a real database (perhaps different databases for development, test, and real use).

Actor: Adapter

A port is more or less the messaging mechanism, and the Adapter is the message listener, because it is the responsibility of the message listener to grab data from the message and translate it into parameters suitable to pass into the Application's API (the client of the domain model).

Variations

While in essence all built on the same mechanics, there are many variants of the Ports & Adapters architecture style, depending of the number and diversity of inbound and outbound adapters plugged to application boundary, and how operations are invoked and synchronized by the adapters (event-based, message, command, etc.).

Interactions

An instance of the application is created, as well as the adapters. The adapters are passed to the core logic (ideally via dependency injection). Adapters start to drive the application.

As events arrive from the outside world at a port, a technology-specific adapter converts it into a usable procedure call or message and passes it to the application. The application is ignorant of the nature of the input device.

When the application has something to send out, it sends it through an adapter/port, that the receiving technology (human or automated) has the appropriate means to consume information, hence feeding it back to the client / consumer.

Assessment

Executability (Development, Testability)
The size and scope of the shared application logic must be kept under control, otherwise at risk to become a monolithic architecture over time. Testability is high (mock up objects, dependency injection). The core logic can be tested independent of outside services. A big advantage of this architecture style is that Adapters are easily developed for test purposes. The entire application and domain model can be designed and tested before clients and storage mechanisms exist. Significant progress can be made on the core without the need for supplementary technical components.
Performance
Proximity of components within the application boundary and in-process call promote good performance. However uncontrolled amount conversation between ports/adapters + badly implement marshaling and UN-marshaling can impact performance.
Topology (Deployment / Allocation)
This architecture implies allocation on only one main node/component. Referred as "monolithic" because of coarse grain deployment. Adapters requires close version control. Requires entire redeployment of the entire application via an integrated release.
Scalability
Can physically scale at a price, by adding more power to a single server, such as more memory, or by adding clusters.
Maintainability / Flexibility (Ease of change, Coupling)
Not easy to replace application boundary / core domain components without affecting the whole application core. It takes a toll in terms of time and effort to plan the release and manage tightly coupled interdependent modular development. As the application grows in size, due to tight coupling between components, it becomes difficult to do easy and frequent releases. Release planning takes a lot of time of people from various groups.
Availability / Reliability (Resilience, Failures recovery)
Frequent release is discouraged for making sure the application should not break due to the newly released feature.
Security
From scratch.
Portability
Can not employ diverse programming languages to answer specific specialized needs.

Related Patterns

Mock Object

Adapter

Dependency Injection (encourages a way of producing an architecture that leans naturally toward the development of a Ports and Adapters style).

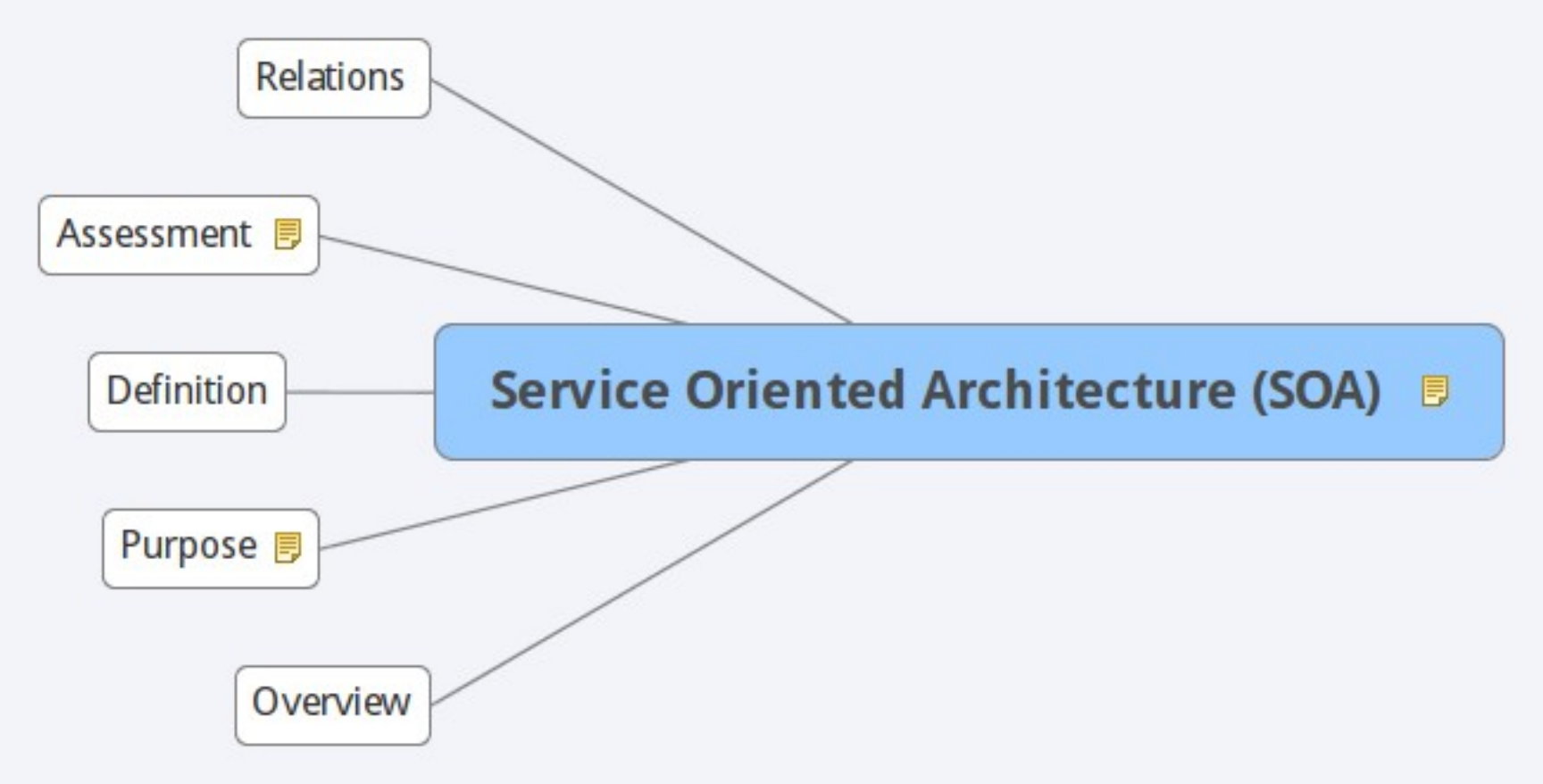
Related Implementation / Frameworks

Apache Isis (a Java framework for rapidly developing domain-driven applications).

Spring (dependency injection framework).

Clean Architecture (Android variant of Ports & Adapters).

Service Oriented Architecture (SOA)



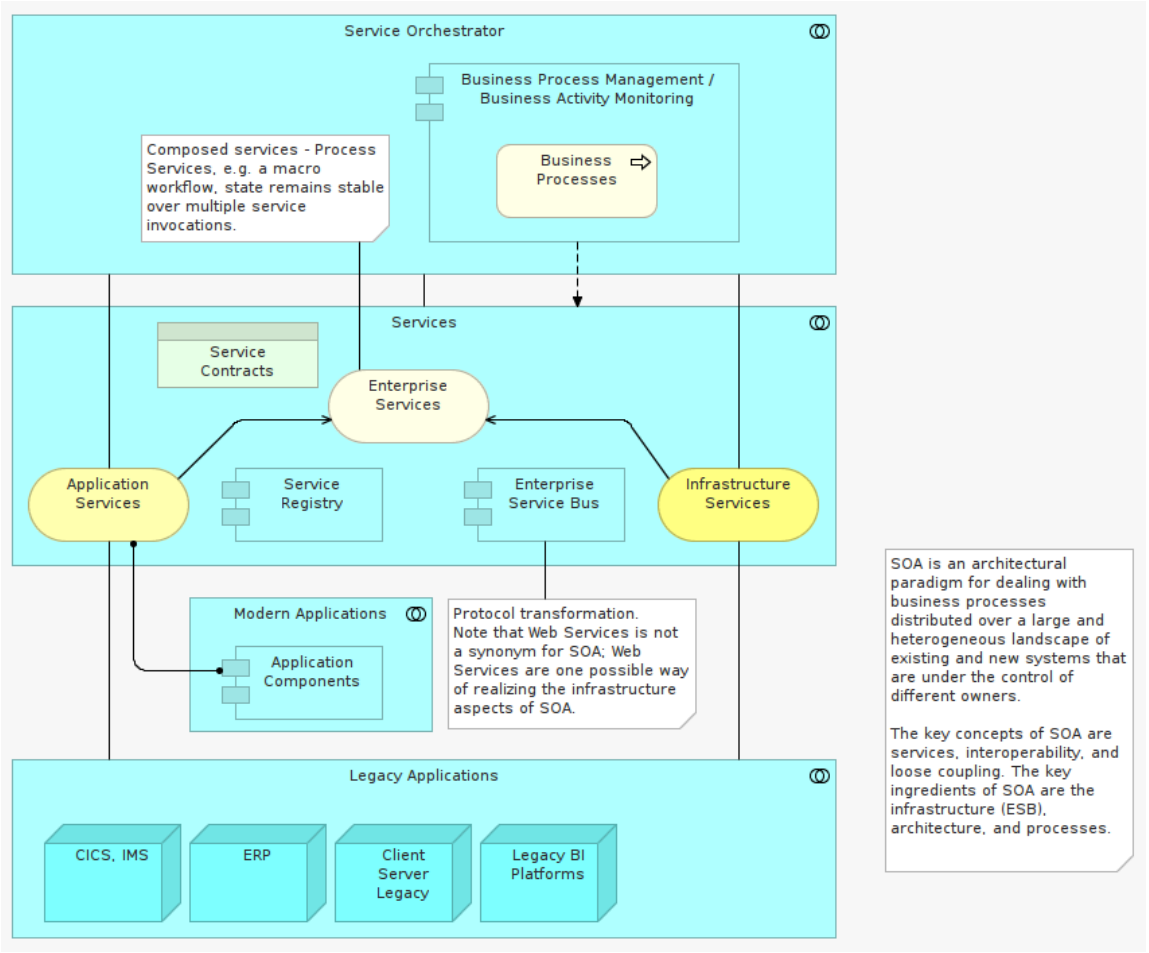
Service Oriented Architecture (SOA)

A Service Oriented Architecture style is a mean to force COARSER GRAIN coupling at BUSINESS PROCESS level (not service, not component) between distributed applications via standard, platform independent, well-defined interfaces. The primary driver for service orientation adoption is interoperability – i.e. protection of investments by supporting legacy system integration, also referred as Enterprise Application Integration (EAI).

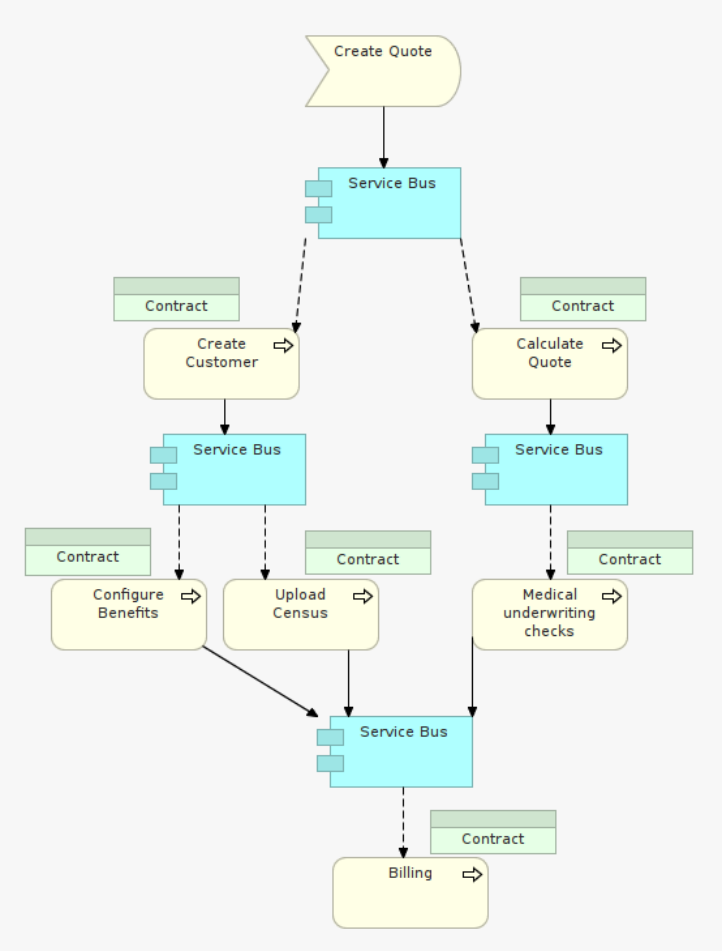
The goal is to architecture processes as a composition of processes (bricks of a SOA), via Remote Access or Messaging over the network.

This architectural style is not only rooted in open standards interchange, but also enabled by platforms providing a wide range of foundational infrastructure services to orchestrate data exchange between applications (Message Bus for example).

What does it look like?



How does it Work?



Purpose

Service Oriented Architecture (SOA) is an architectural paradigm for dealing with business processes distributed over a large and heterogeneous landscape of existing and new systems that are under the control of different owners.

This architectural style promotes the orchestration of “resources” in a single consolidated location, to compose coarse grain enterprise services, in turn to be orchestrated for other purposes.

The key ingredients of of this architectural style is interoperability and loose coupling via open standards for data contracts and message envelopes, and enterprise service bus infrastructure (ESB).

To promote Service Decoupling

Using a service registry.

Using protocol transformation.

Using message delivery.

Using data contracts.

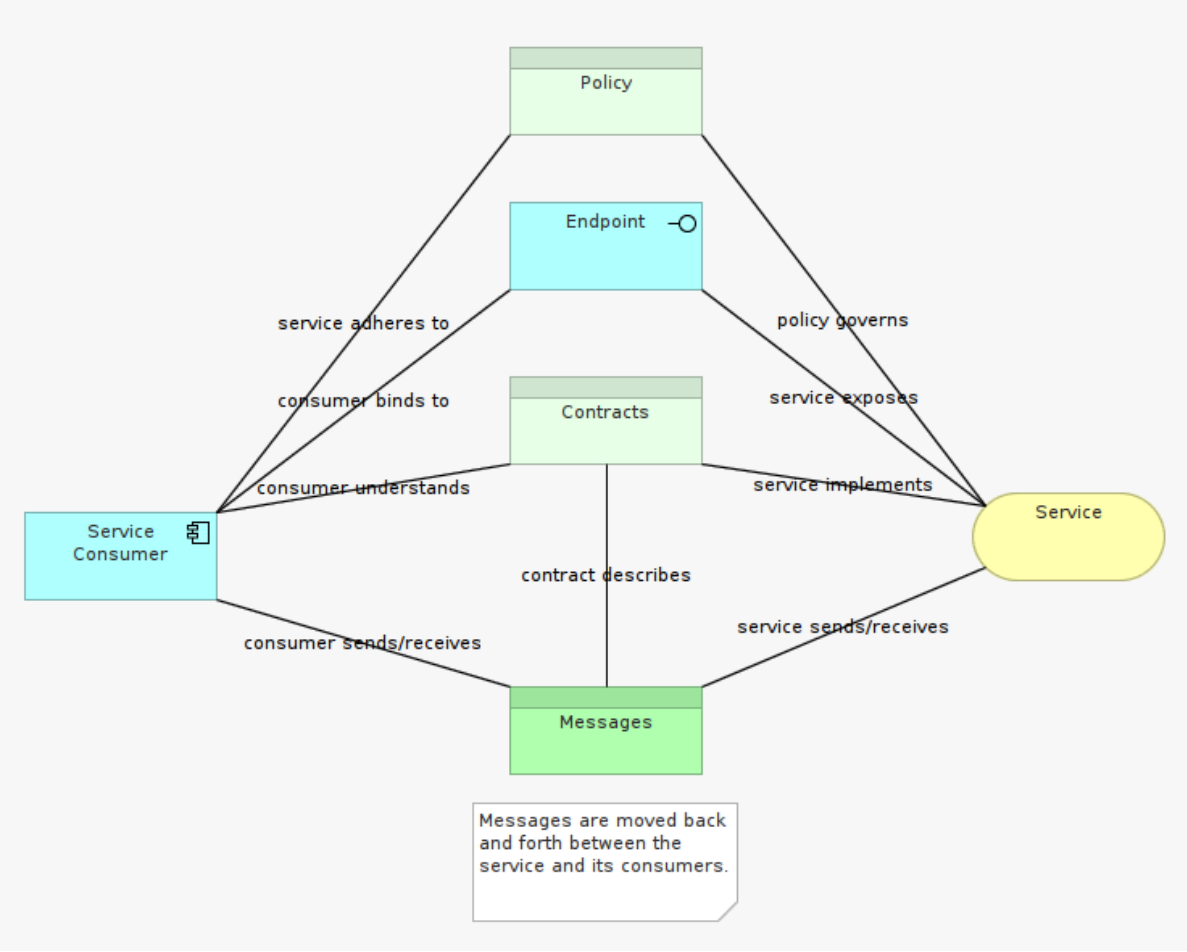
To maximize Re-use

Orchestration is a key concept of the SOA style aiming to aggregate services for future re-use.

It re-composes existing services to a new service that has central control over the whole process.

For example, BPEL is a WS standard for service orchestration, mapping service to business process, for which development tools and engines are available.

Actors



Actor: Basic Service

A basic service provides “basic” functionalities from a single backend application. These services are usually part of the first set of services that wraps or hides implementation details of a specific component. Basic services can be data-driven or logic-driven. They are the base for composing higher services such as composed services and process services.

Actor: Composed Service

A common term for services that are composed of basic services and/or other composed services.

Actor: Process Service

A process service represents a work flow or sequence of complex process steps. From a business point of view, this kind of service represents a macro flow, which is a long-running flow of activities (services) that is interruptible (by human intervention). Unlike basic services and composed services, these services usually have a state that remains stable over multiple service calls.

Variations

In the Service Oriented Architecture style, variations pertain to different ways to bind services.

Simple mediation and re-routing, where an inbound message is transformed and enhanced enhanced, The resulting output becomes the input of the next immediate target service.

Orchestration, where a message is received and is the trigger for the invocation of downstream services, to compose clusters of message requests that need to happen in sequence or all at the same time.

Choreography, where a message is received and is the trigger for a chain of invocation of downstream services, in turn each informing the other about next the next service to invoke.

Interactions

The fundamental rule of interaction in a SOA architecture style is that no service should accesses another service underlying component logic or data store. A corollary to this is that no component (a suer interface front-end, or a external system) should be allowed to bypass the services. Any form of component interaction requires to obtain a service from a service registry, understand its communication contract before invoking its endpoints, or sending a message.

About Web Services Protocols

Web Services are set of standards that serves as one possible way of realizing a SOA infrastructure. Initially started with the core standards XML, HTTP, WSDL, SOAP, and UDDI, it now contains over 60 standards and profiles developed and maintained by different standardization organizations, such as W3C, OASIS, and WS-I. SOAP is the basic protocol of Web Services. As an XML-based format, it defines the format of the header and body of a Web Services message. The SOAP protocol allows different types of message exchange. Web Services standards define all that is necessary to provide interoperability between systems without the need to buy some specific hardware or software, provided all exchanges are performed via SOAP.

But since the goal of an SOA architecture style is the connect heterogeneous application, there are many forms of data interchange protocols supported using a Message Bus component, allowing classic file transfer, asynchronous message delivery, and invocation of synchronous web services.

Interaction method: Point to Point RPC

Point to point RPC: An interaction mechanism for inter-application communication between components referred as SOAP that evolved from RMI and XML-RPC. The principle states that any component of an SOA exposes a web service interface that can be invoked by an external application. It simulates a function call, performed from a remote component.

Interaction method: Message Queues

Message Queues: An interaction mechanism for message exchange between components using “queues” providing a set of services: Guaranteed message delivery (fire and forget), Continuous message monitoring (confirmation of reception), Broadcast delivery (fire and forget to multiple locations), Transactional: Broadcast requiring confirmation before committing.

Interaction method: Message Bus

An Message Bus (ESB) is an EAI platform implementing all of the above interaction mechanisms. It is commonly accepted as the infrastructure of a SOA landscape that enables the interoperability of services between enterprise applications. Its core task is to provide connectivity, data transformations, and (intelligent) routing so that systems can communicate via services. An ESB also provides additional abilities that deal with security, reliability, service management, and service composition.

Assessment

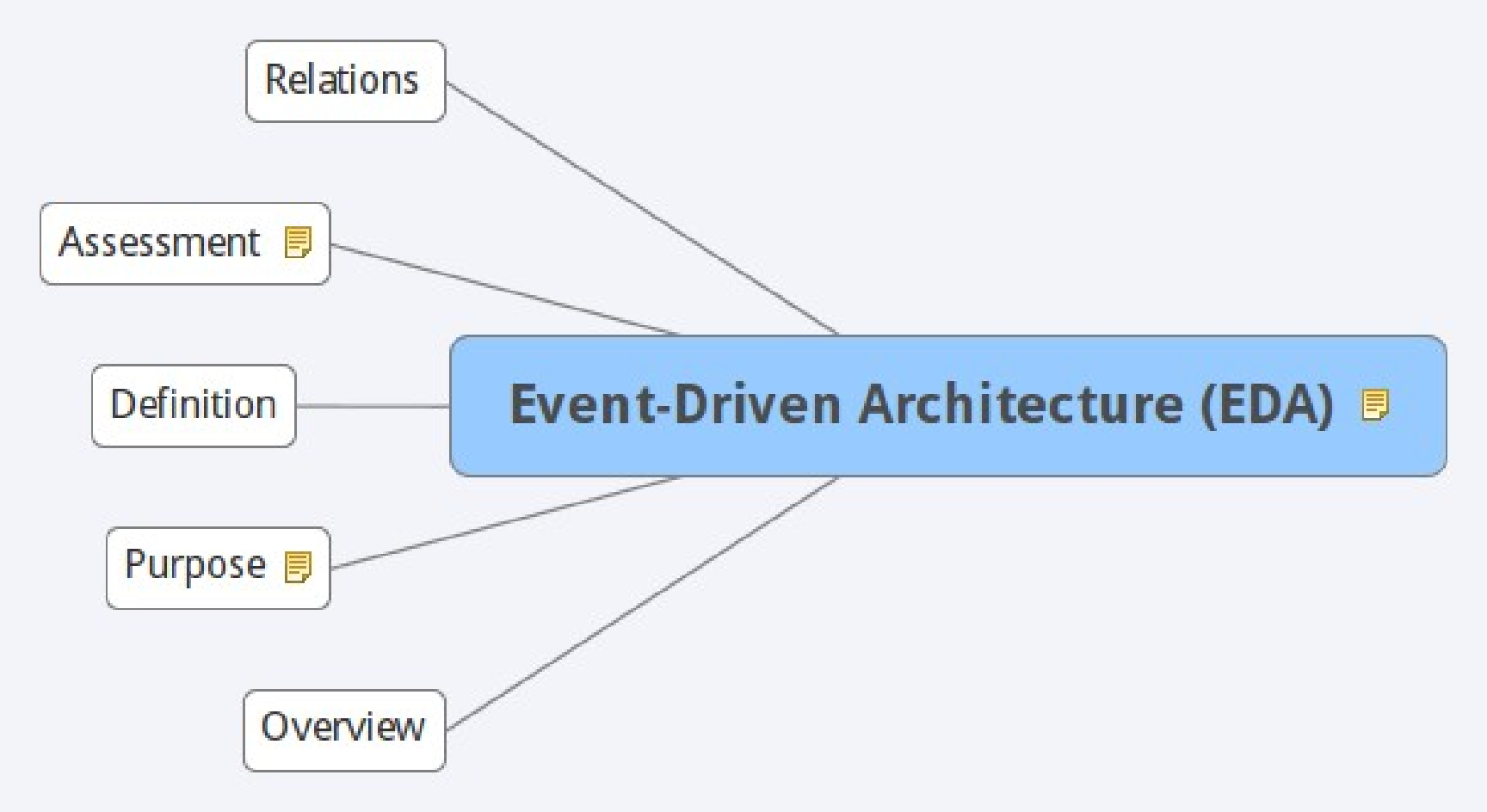
Executability (Development, Testability)
Facilitates the integration of different applications/platforms from different vendors. Easy to test and debug when managed as small, independent services. Distributed logging essential to monitor holistic view of a distributed transaction. Difficult to track service flow across applications. Complexity can explode, contracts can explode, requiring some external tooling to keep the event flow logic managed and under control using BPEL, BPM platforms.
Performance
Every time a service interacts with another service, complete validation of every input parameter takes place. This increases the response time and machine load, and thereby reduces the overall performance. A lot of conversations between components. badly implement marshaling and UN-marshaling can impact performance.
Topology (Deployment / Allocation)
Probably one of the best understood and documented enterprise deployment architecture style, leveraging Web standards firewall/proxy configurations.
Scalability
Leveraging Web standard protocols for load balancing. Multiple instances of a single service can run on different servers at the same time. This increases scalability and availability of the service.
Maintainability / Flexibility (Ease of change, Coupling)
Services are usually published to a directory where consumers can look them up. This approach allows a service to change its location at any time. Services can be reused in multiple applications independent of their interactions with other services. Services can be easily updated or maintained as long as contract interfaces do not change so large, complex applications can thus be managed easily. However contract interfaces often change, must be versioned and managed centrally (referred as SOA governance).
Availability / Reliability (Resilience, Failures recovery)
In case of failure, a log analysis must be performed to track down issues. Service Oriented Architecture are often seen as brittle – i.e. low tolerance for failure. This is mainly because common initial assumptions are that a network is reliable with low latency and no bottlenecks, remote servers are responsive and always available. SOA requires an understand of risk management.
Security
WS-I Security, an open standard protocol to sign SOAP messages using XML signature and XML encryption to provide end-to-end security.
Portability
Services can be developed in different languages and leverage different technologies.

Related Patterns

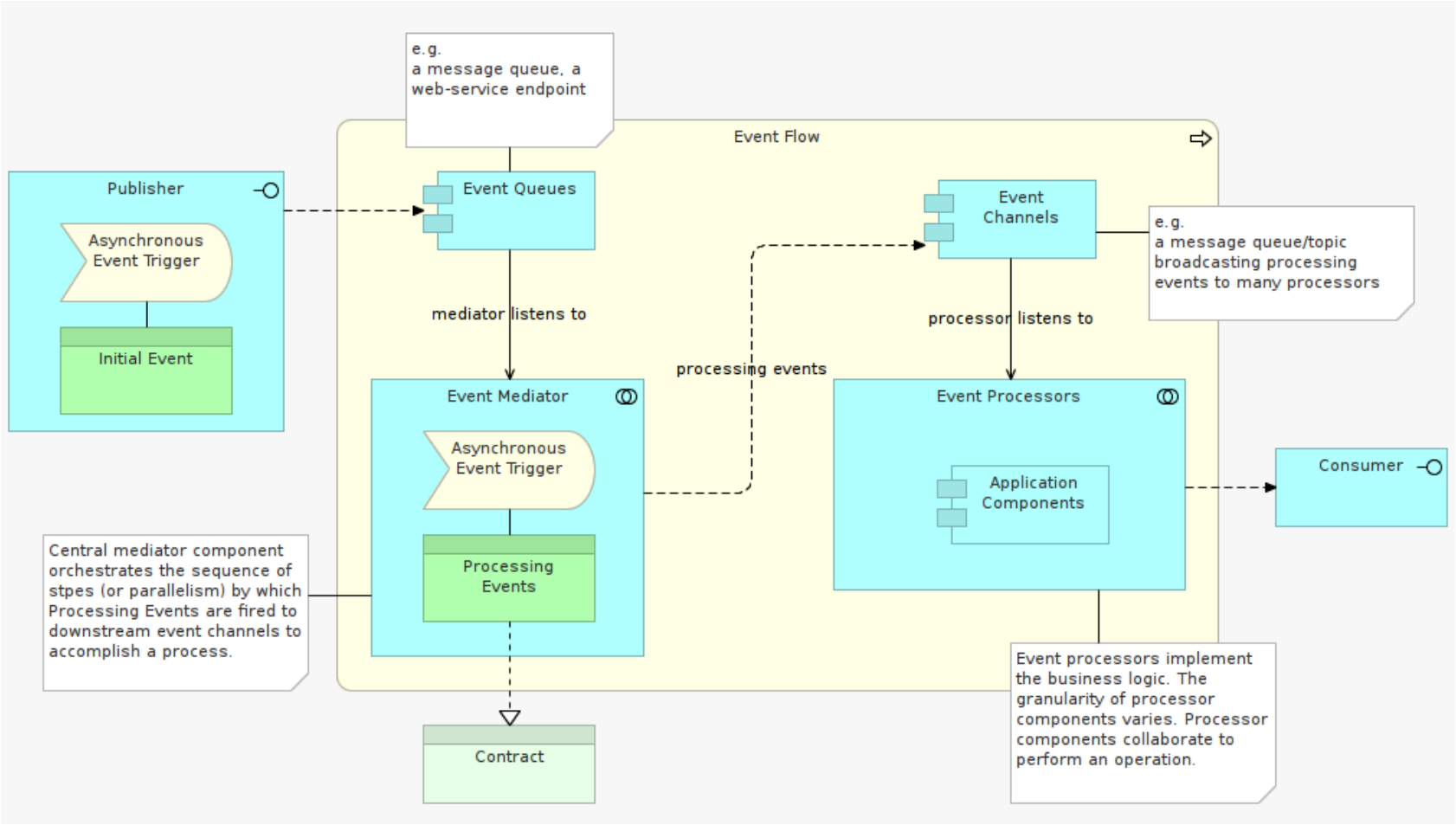
Related Patterns:
- “Document/literal wrapped" pattern
- Message Broker
...many enterprise integration patterns.

Related Frameworks:
- Apache ODE, JBPM
- JMS, Active MQ
- Mule ESB

Event-Driven Architecture (EDA)



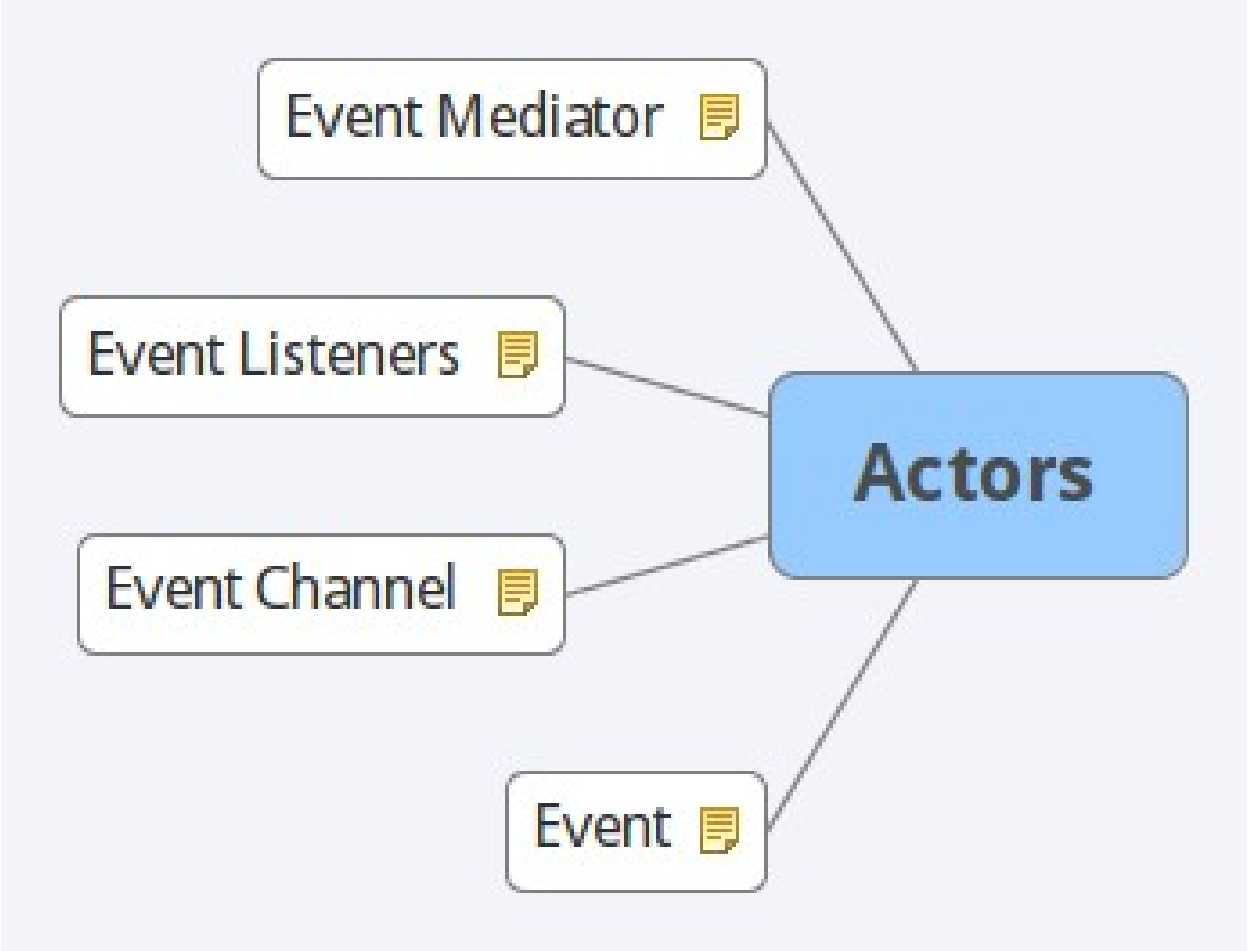
What does it look like?



Purpose

The EDA style is a push-based communication between publishers and consumers. It is used to communicate with a multitude of consumers with minimum or no delay and avoid bottlenecks. It is a style of application architecture that can be implemented in any language, which can improve responsiveness via process parallelism, throughput, and flexibility.

Actors



Actor: Event

A notification sent to a more or less well-known set of receivers (consumers). Usually, the receivers of an event have to subscribe for a certain type of event (sent by a certain system or component). Depending on the programming or system model, the systems sending the events (the providers) might or might not know and agree to send the events to the subscribing receivers.

You can consider events as part of the publish/subscribe message exchange pattern.

Actor: Event Channel

An event channel is a mechanism whereby the information from an event mediator is transferred to the event engine. This could be a TCP/IP connection or any type of input file (flat, XML format, e-mail, etc.).

Several event channels can be opened at the same time. Usually, because the event processing engine has to process them in near real time, the event channels will be read asynchronously

The events are stored in a queue, waiting to be processed later by the event processing engine.

An event queue is a type of event channel. Event Queue transports an event to the event mediator.

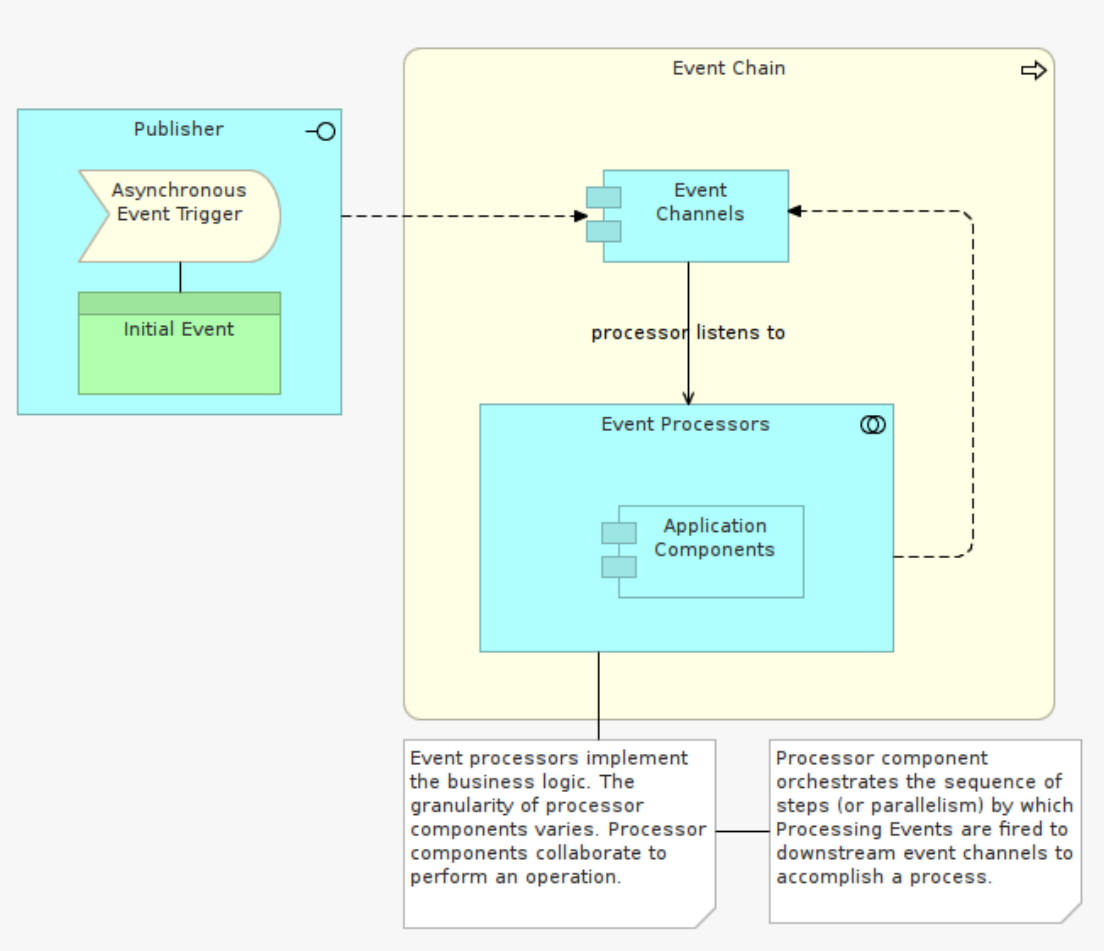
Actor: Event Listener

Event Listeners act as asynchronous receivers, which automatically receives messages as they're delivered on event channels. An event mediator is a type of event listener. An event processor is a type of event listener.

Actor: Event Mediator

Events are identified, and appropriate reaction are decided upon in the Event Mediator engine. Reactions are executed from the Event Mediator (i.e. orchestration of processing events to fire downstream in parallel to multiple event processors, or in sequence). For example, on reception of an initial event “product ID low in stock”, a mediator will trigger processing events such as, “Order product ID” and “Notify personnel”.

Event Chain Variant



Purpose

In this variant of the EDA architecture style is in fact the origins of event-driven architecture. A cascading chain of events processed by event processors implements the business logic of an application. In this variation, there is not Event Mediator, and no Contract Interfaces being managed. The flow isn't managed from a central location, making it difficult to trace. Flow of data is choreographed as it transits through processor components.

Interactions

The only ways publishers can communicate with listeners is via events.

The EDA architecture style is extremely loosely coupled because the publisher doesn't know the identity of the consumers.

There is no direct coupling between the publisher and the consumer i.e. loosely coupled.

Assessment

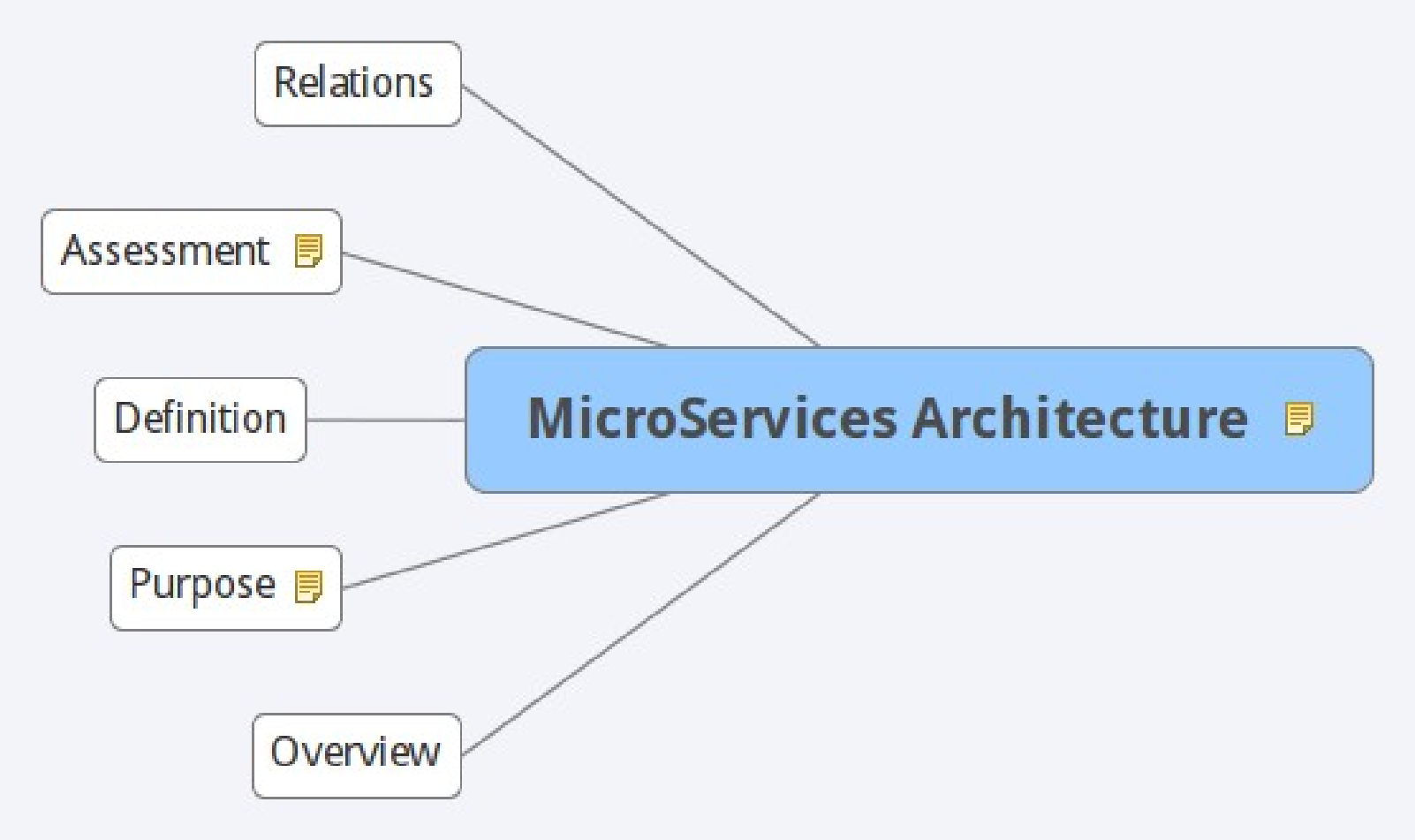
Executability (Development, Testability)
Development no traceability of logic, chain of responsibility. Complexity can explode, contracts can explode, requiring some external tooling to keep the event flow logic managed and under control using BPEL, BPM platforms.
Performance
Performance is high with parallel processing of many events possible in different processes.
Topology (Deployment / Allocation)
Deployment is fine grained.
Scalability
Allows for evolutionary design. Each event processor can reside on a dedicated server node. Scalability is high as long as the granularity of processors is managed.
Maintainability / Flexibility (Ease of change, Coupling)
Mediator and Processors are highly decoupled from one another, can be deployed independently from one another, can be changed without ripple effect on the overall architecture. However, the same issues regarding maintainability pertain to development and testability.
Availability / Reliability (Resilience, Failures recovery)
In case of failure, a log analysis must be performed to track down issues. Transactions cannot be reversed across event processors. Re-initializing the application based on an EDA architectural style may cause losses of transiting events, and event flow may not exactly resume from where it stopped.
Security
From scratch.
Portability
Event Processors can be developed in different languages and leverage different technologies. This is especially true for the Event Chain variant of the EDA architecture style. Event Mediators are often bundled as part of an EAI commercial platform.

Related

Related Patterns:
- Pipes & Filters
- Publish / Subscribe

Related Frameworks:
- Apache Camel
- Mule
- Oracle EDA Suite
- Apache ODE, JBPM

MicroServices Architecture



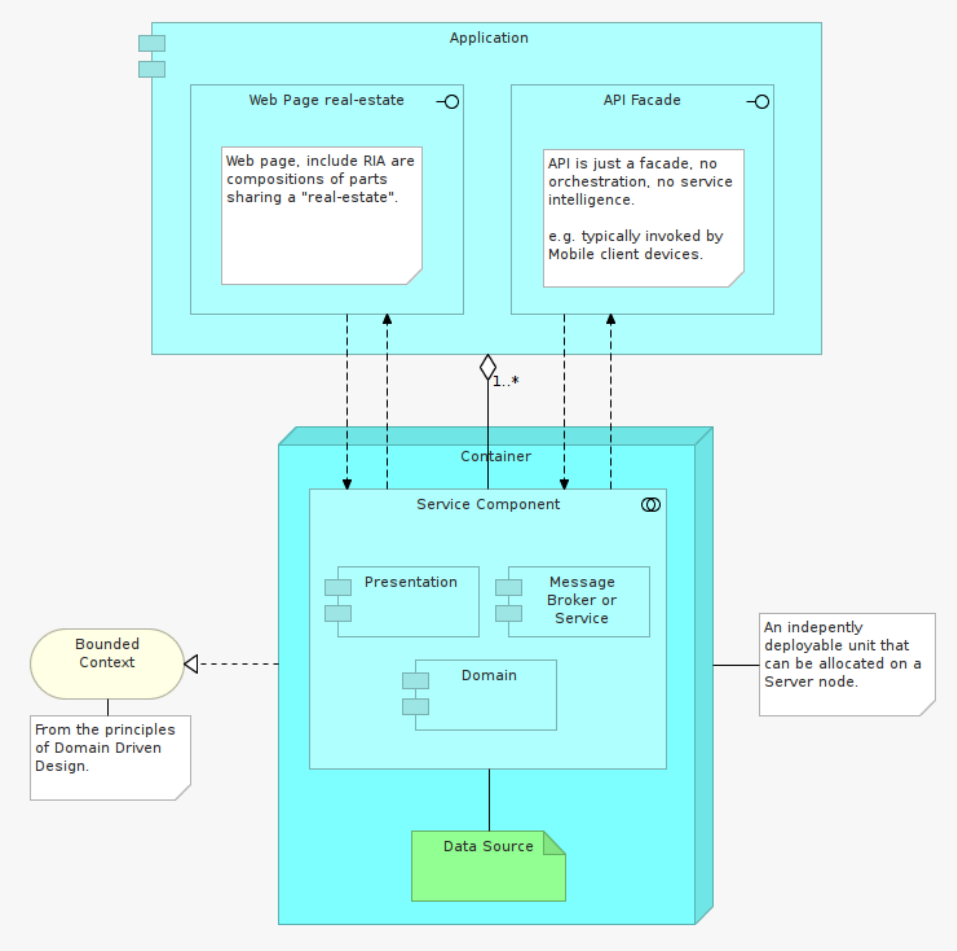
MicroServices Architecture

The Microservice Architecture style aims to partition the end-to-end logic of an application in domains concepts each encapsulating all layers of an application function (i.e. services, protocols, logic, data) into collection of highly decoupled and independently deployable units. The paradigm aims to build Evolutionary systems over time.

This style is born is born from experiencing the pros and cons of previous generations of architectural styles, domain-driven design, continuous delivery, on-demand virtualization, infrastructure automation, small autonomous teams, systems at scale.

It emerged as a pattern from small teams building applications in large organizations (born out of frustration and overkill) with desire to own to re-appropriate to themselves full life-cycle of their services to remain in control of systems complexity over time.

What does it look like?



Purpose

To isolate failure, design for failure.

To deploy independently, Eliminate the pains of deploying, maintaining and testing.

To hide implementation details, Keep it simple, granular service components that do one thing well with respect to their domain. “Gather together those things that change for the same reason, and separate those things that change for different reasons”.

To automate build and delivery

To scale, avoid introducing large and expensive enterprise message bus, and make of scalability a developer concern before it becoming a system administrator concern.

To decentralize governance, data governance, service governance. Accept building evolutionary architectures over time is a possibility (living system).

Actor: Service Components

A service component implement the notion of Bounded Context from Domain-Driven Design principles.

“No one outside of the bounded context knows what is inside”.

“You can connect to it via a service but it's a black box to you”.

The Service Components of a Microservice Architecture style ARE the system.

Variations

Variations of the Microservice integration style pertain to the style of integration of service components from Clients and Consumers systems.

Service components can be access via REST or message or remote access protocols.

Components are small, granularity doesn't matter.

Microservice uses choreography to allow service components to interact. Unlike orchestration, choreography does not compose services to a new service that has central control over the whole process. Instead, it defines rules and policies that enable different services to collaborate to form a business process. Each service involved in the process sees and contributes only a part of it.

Message brokers can be embedded in service components to facilitate distribution and decoupling. Doing so a service component can behave like an event listener if needs be.

Interactions

The architecture must first and foremost plan for the size (i.e. granularity) of Service Components. The more components, the more transactions. The bigger the components, the less transactions, but you loose the advantages and move towards the monolithic architecture again.

Rules of interactions within a Microservice Architecture style are best expressed in terms of what NOT to do.

Binding on the database schema is allowed and recommended (not data abstraction layer). If the database schema changes, the service components need to change and be redeployed transparently. Sharing databases kills the purpose of the Microservice architectural style.

Avoid service dependencies between service component or there will be a need to redeploy everything.

Avoid any form of orchestration dependencies between service components, keep micro concepts or you will destroy the Pattern and run into real troubles.

Assessment

Executability (Development, Testability)
Can become complex if the scope of Service Components increase beyond "micro". Service-to-service components interactions must be managed and not become a substitute for Service Orchestration. Contract maintenance can become difficult if left unmanaged or not version controlled. Regression testing can be scoped down to a reduced number of service components and immediate clients/consumers that are re-deployed.
Performance
Remote protocols (like REST) as based on Web standards and as such can benefit from optimisation techniques like caching.
Topology (Deployment / Allocation)
Increased uptime during deployment.
Scalability
Each deployable unit can be scaled individually. Granularity give options for scaling that monolithic applications cannot provide. Remote protocols (like REST) as based on Web standards and as such can benefit from load balancing techniques like caching.
Maintainability / Flexibility (Ease of change, Coupling)
Service Components are highly decoupled from one another, can be deployed independently from one another, can be changed without ripple effect on the overall architecture.
Availability / Reliability (Resilience, Failures recovery)
In case of failure, a log analysis must be performed to track down issues. It is still early days but activity monitoring and automated recovery are starting to emerge.
Security
Surface area of threats is increased by the multiplication of service components. Remote connections must be secured. This style can leverage implementations for SAML, WS-I Security, OAuth 2.x. Web standards and Remote protocols are well understood from previous architecture styles,. The Microservice Architecture Style doesn't introduce new protocols. Deployment containers are extremely secure.
Portability
Service Components can be developed in different languages and leverage different technologies.

Related

Related Patterns:

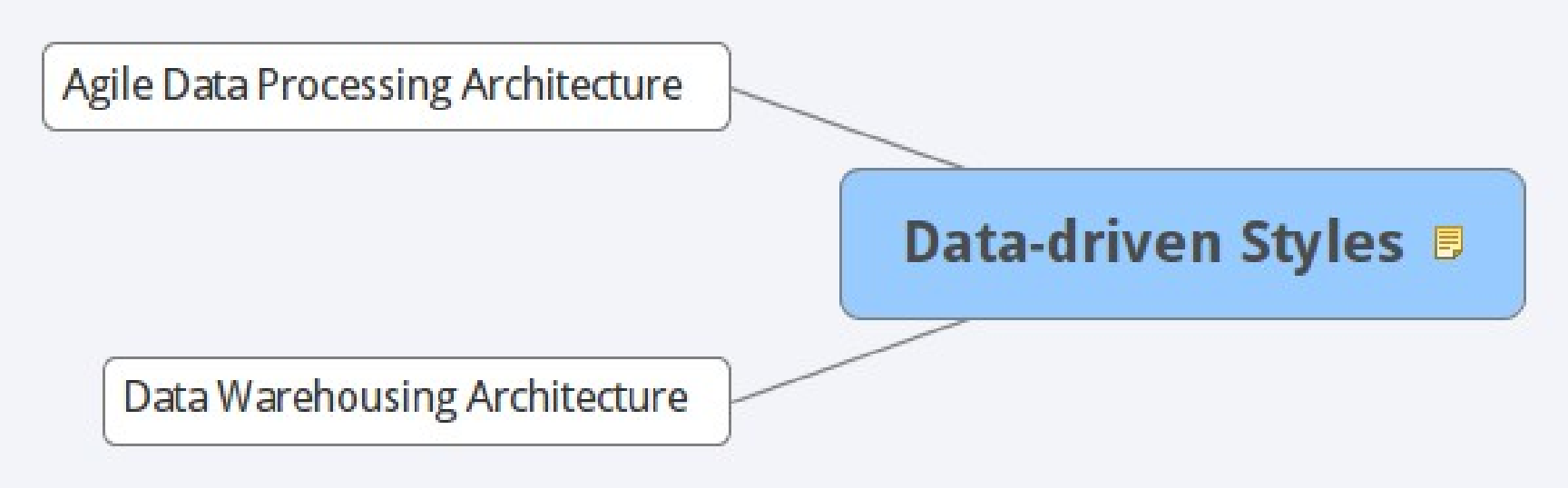
- Deferred to Service Components → a wide range of choices.

- Inherits many patterns from older architectural styles as long as these do not violate the interaction rules of the Microservice architecture style.

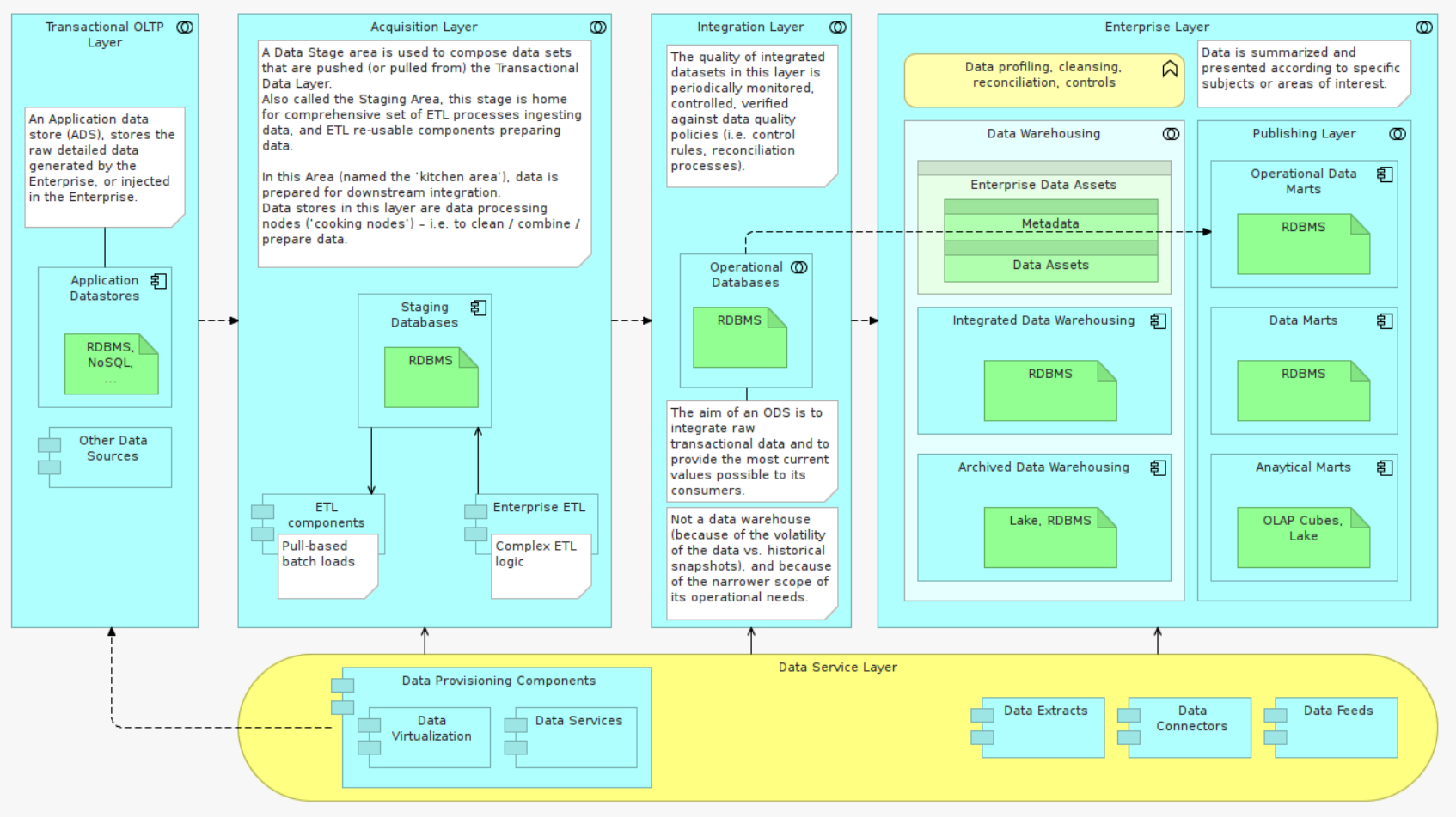
Related Frameworks:

- Axway for API Layer

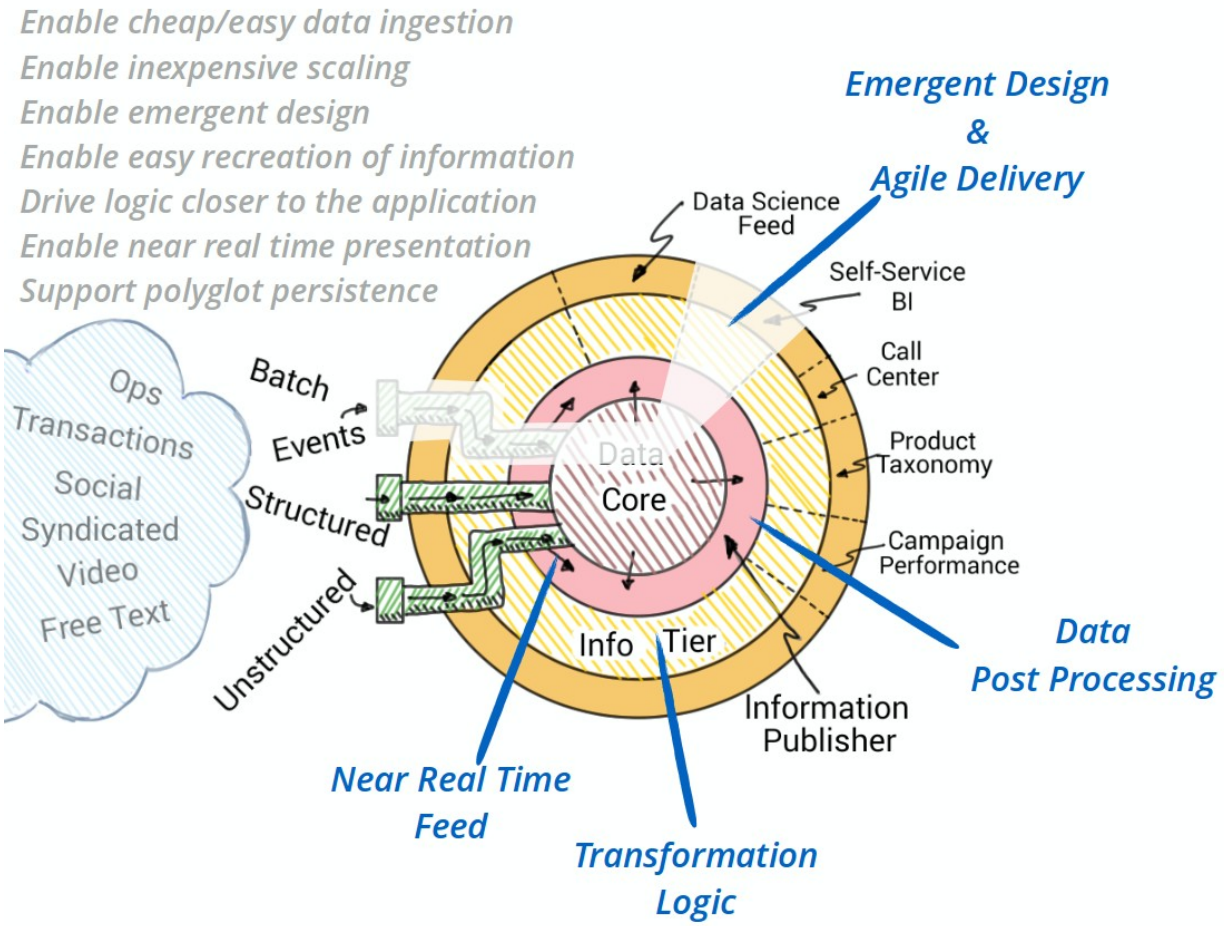
Data-driven Styles



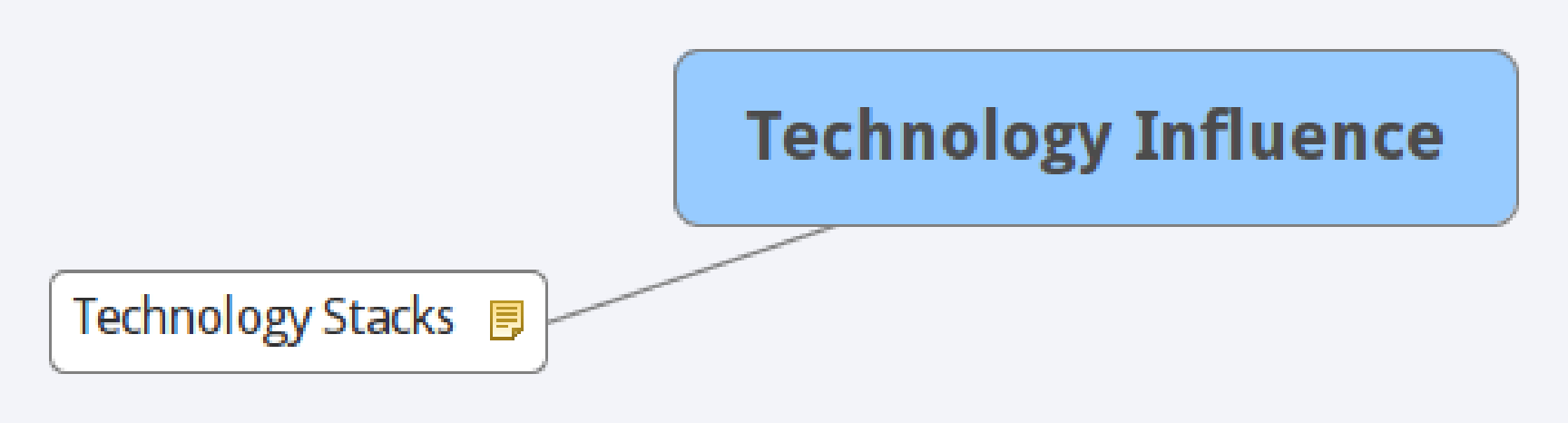
Datawarehousing Architecture



Agile Data Processing Architecture



Technology Influence



Technology Stacks

Making patterns and Styles irrelevant: concluding remarks on ORM, SQL, RIA/Flash.

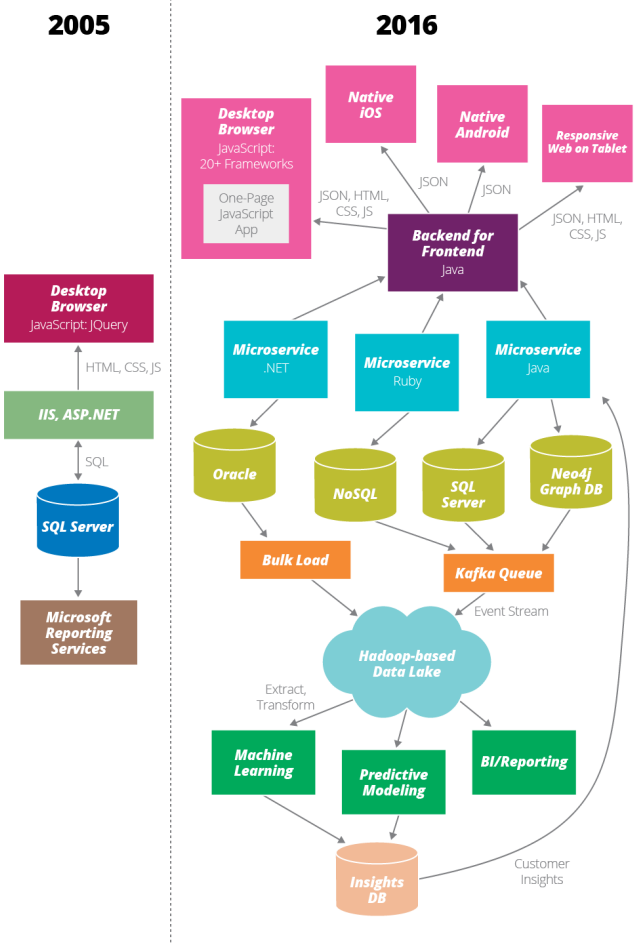
Other discussion points:

- Technology force and the Pendulum effect
- Technology Debt and refactoring.

New Technologies are the best friend of IT Architecture.

Architects swing the momentum injected by NT to "re-think" and "simplify" the TECHNICAL DEBT that crept in within the enterprise over time.

Application Technologies (2005 / 2016)



Data Technologies (2016 - 2025)