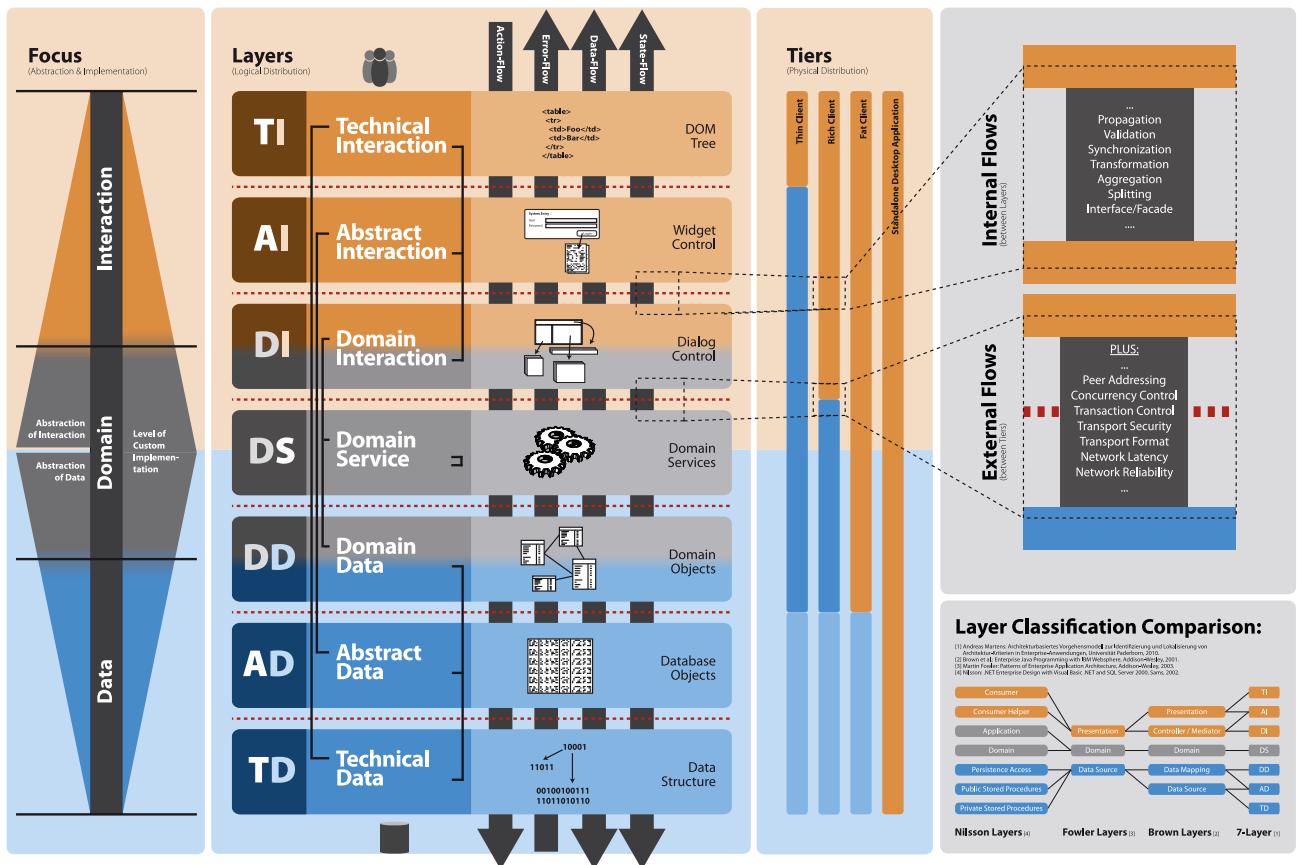




Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall



In an application, it is possible to distinguish 7 logical layers, each grouped in two ways: on the one hand, there are the three sequential layer groups **Technical/Abstract/Domain Interaction**, **Domain Service** and **Domain/Abstract/Technical Data**, on the other hand, there are the three nested layer groups **Technical Interaction/Data**, **Abstract Interaction/Data** and **Domain Interaction/Service/Data**.

In addition, one can distinguish 4 primary flows in an application: the **Action Flow** consequently runs from top to bottom only because all actions at the top are triggered by the user (or neighboring systems); the **Error Flow** consistently runs only in the opposite direction, i.e., from the bottom to the top, because errors, in the worst case, must be reported to the user; the (domain-specific) **Data Flow** and the (technical) **State Flow** run in both directions because data and states have to be persisted as well as displayed.

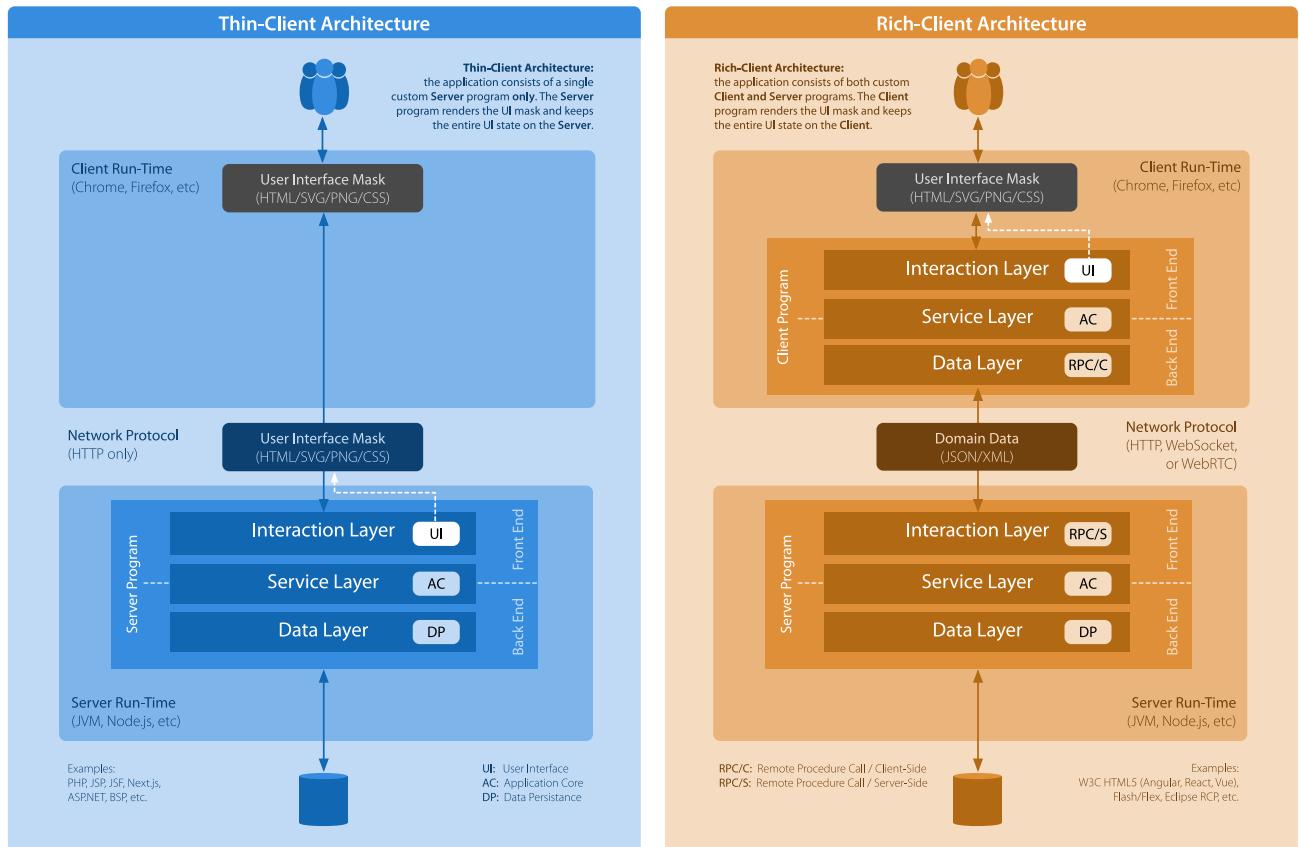
The abstraction of Interaction/Data in the layers increases from the top/bottom towards the middle, so most of the functional code of an application is written there. For the upper/lower layers, one usually massively relies on Open Source libraries/frameworks.

If instead of a logical cut (resulting in an **Internal Flow** between the layers), one makes a physical cut (which then results in an **External Flow**), i.e., one distributes the application into single programs on different computers, then the resulting architecture is called according to the scope and responsibility of the client.

With **Thin Client**, only the **Technical Interaction** is offloaded to the client, while with **Rich Client** the entire user interface (i.e., all three layers **Technical/Abstract/Domain Interaction**) is autonomously offloaded to the client (usually as a so-called "HTML5 Single-Page-Application"), with **Fat Client** there is no more associated server at all, and with the **Standalone** application, there is only one single program.

Questions

- ?
- What is the name of the application architecture in which the entire user interface runs autonomously on the client, while the server only provides purely functional services?
- ?
- What are the web applications named that implement a **Rich Client** architecture?



In the **Thin-Client Architecture**, the application consists of a single custom Server program only. This Server program renders the User Interface Mask and keeps the entire state of the User Interface on the Server.

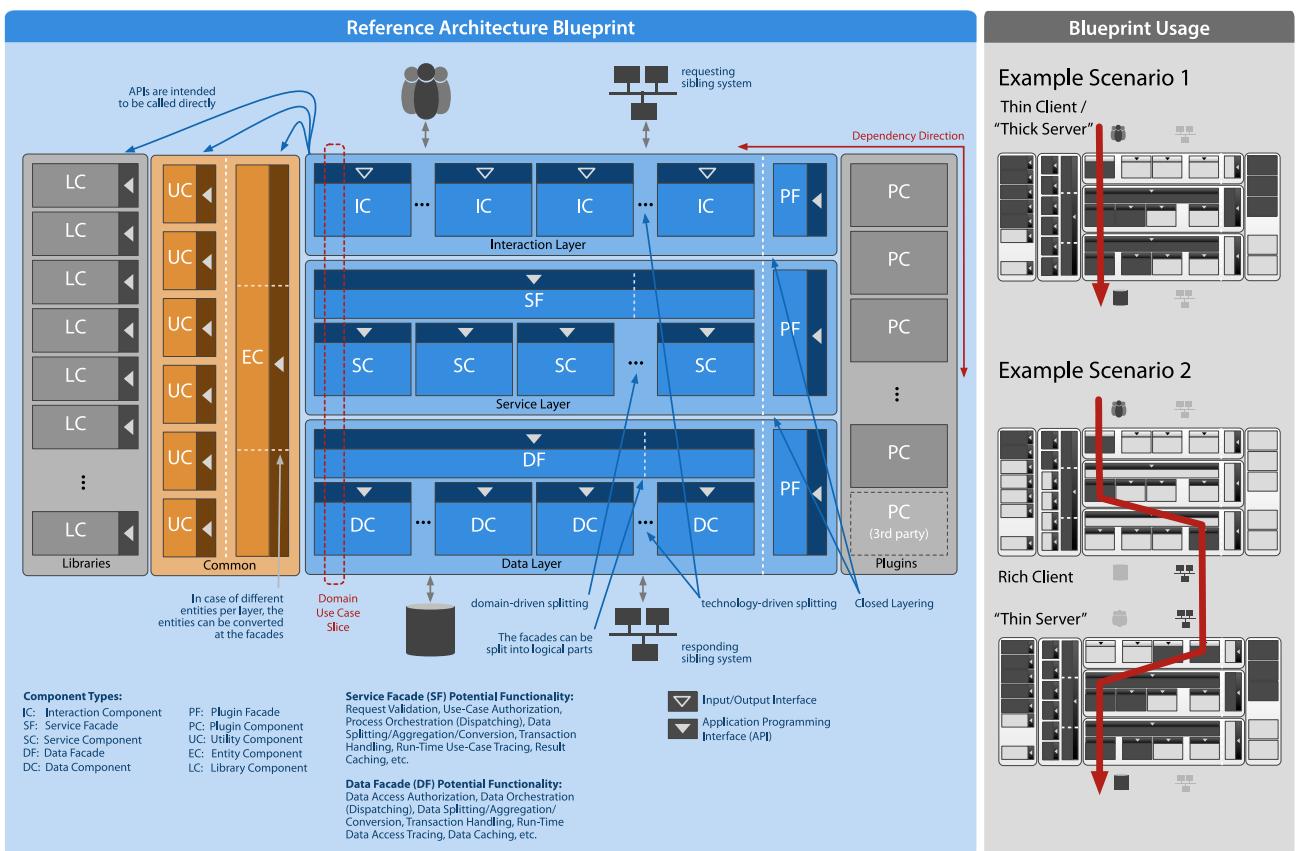
The advantage of this architecture is that the application can be updated very easily. The disadvantage of this architecture is that the user interface reacts sluggishly, and the state of the user interfaces of all clients must be kept on the server, which can make the server a bottleneck.

In the **Rich-Client Architecture**, the application consists of both custom Client and Server programs. The Client program renders the User Interface mask and keeps the entire state of the User Interface on the Client.

The advantage of this architecture is that the user interface is highly responsive, only domain-specific data has to be exchanged between the client and the server and the server becomes less of a bottleneck. The disadvantage of this architecture is that, if necessary, the client has to be updated explicitly via an installation procedure.

Questions

- ❓ With which Client Architecture does the User Interface offer the higher responsiveness?



A (business) Information System usually follows a stringent component-based reference architecture. This is represented “full blown” and can be arbitrarily “slimmed down.”

First, this reference architecture consists of 3 substantial Layers: the **Interaction Layer** with the (technically cut) components, which provide the I/O-based interfaces to the user (User Interface) and/or requesting neighboring systems (via Network interface), the **Service Layer** with the (domain-specifically cut) service components (also called Application Core) and the **Data Layer** with the (technically cut) components that provide the connection to the own database and/or neighboring systems to be queried.

Note that the “docking position” of a neighboring system depends on its roles: if it requests, it docks to the Interaction Layer; if it is queried, it docks at the Data Layer. If it happens to have both roles, it docks twice. The other view is that both the user and the database can be understood as special “neighbor systems.”

To connect the **N Interaction Components (IC)** with **M Service Components (SC)** a decoupling **Service Facade (SF)** is usually inserted. For the same reason, there is usually also a **Data Facade**.

The Data Model is offloaded to common **Entity Components (EC)**. Together with possibly shared code, both live in one **Common Slice**. Libraries and Plugins are also offloaded to separate slices, but there are two major differences: Libraries are passive and provide their functionality to the application via their interfaces. Plugins are active and control the application in that they hook into the application via Service Provider Interfaces (SPI) of the **Plugin Facades**.

In the application, there has to be only exactly one **Dependency Direction** so that the application (in the opposite direction of the dependencies) can be built cleanly. The reference architecture is usually also instantiated twice in order to design both a **Rich Client** and an associated “**Thin Server**” from it.

Questions

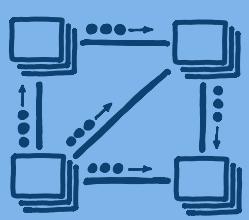
- ?
- With which layer pattern can in an Information System the **Interaction Components** be decoupled from the **Service Components**?
- ?
- In which order are the components of an application built?

Architecture & Systems

DEF Definition

Reactive System Architecture enables the realization of Reactive Systems.

Reactive Systems are in **subordinated interaction** with their **dominating environment**. They **continuously process endless data streams as small messages**, react at **any time** and respond within **tight time limits**. For this, they **continuously observe their environment** and adapt their **behaviour** to the current situation.



Demand & Deliverables

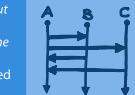
CTX Context

Real-time communication in the context of Digitization, Internet, Internet of Things (IoT), Systems of Engagement, Media and Analytics.



VAL Values

Non-blocking input/output data processing, fast responses within tight time limits, and continuous availability of the provided services.



REQ Requirements

Services are elastic and provide high scalability, and are **resilient** and provide high fault tolerance.



PRP Properties

Services run fully **autonomously**, monitor themselves, and automatically adapt to changes in the environment.



Principles

Stay Responsive

Always respond in a timely manner.

Accept Uncertainty

Build reliability despite unreliable foundations.

Embrace Failure

Expect things to go wrong and design for resilience.

Assert Autonomy

Design components that act independently and interact collaboratively.

Tailor Consistency

Individualize consistency per component to balance availability and performance.

Decouple Time

Process asynchronously to avoid coordination and waiting.

Decouple Space

Create flexibility by embracing the network.

Handle Dynamics

Continuously adapt to varying demand and resources.

Patterns & Paradigms

ARC Architecture

Microservices, Cloud-Native Architecture (CNA), Event-Driven Architecture (EDA).



EXE Execution

Parallelization, Concurrency, Actors, Threads, Thread-Pool, Event-Loop.



COM Communication

Asynchronous Communication, Non-Blocking I/O, Sequence, Push, Backpressure, Quality of Service (QoS).



INF Infrastructure

Message Queue (MQ), Load Balancer, Reverse Proxy, Service Mesh, Virtual Private Network (VPN).



DAT Data

Semantical Event, Small Message, Endless Stream.



PRC Processing

Complex Event Processing (CEP), EAI Patterns, Stream Processing (map, flatMap, filter, reduce), Event Sourcing.



STY Style

Functional Programming, Asynchronous Programming.



ASY Asynchronism

Callback, Promise/Future, Observable, Publish & Subscribe.

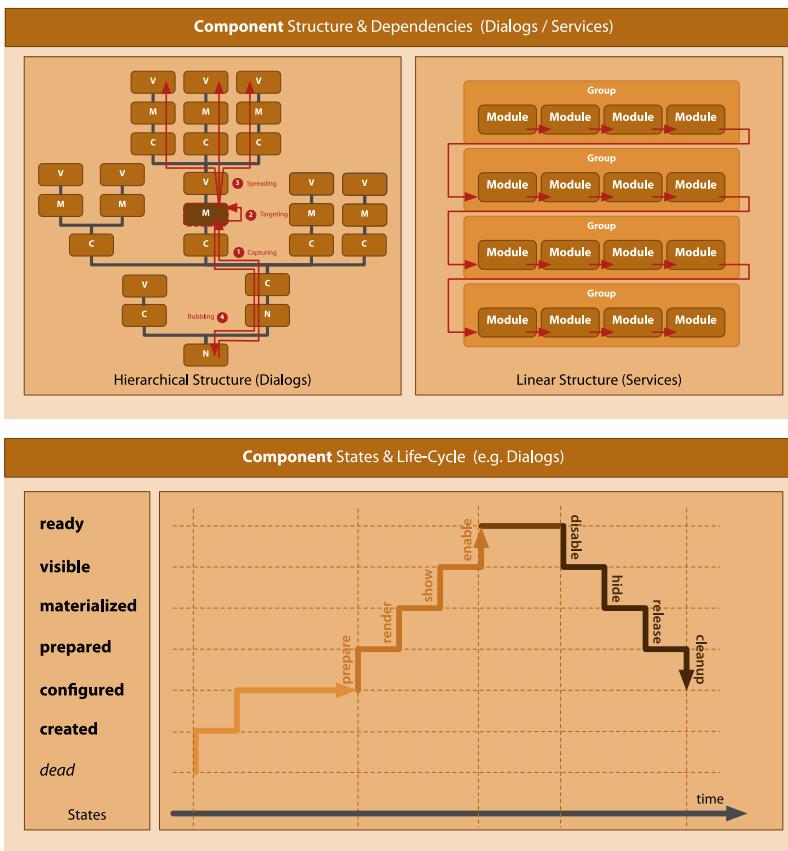


Reactive System Architecture enables the realization of **Reactive Systems**. Reactive Systems are in **subordinated interaction** with their **dominating environment**. They **continuously process endless data streams as small messages**, react at **any time** and respond within **tight time limits**. For this, they **continuously observe their environment** and adapt their **behaviour** to the current situation.

Reactive Systems are primarily used in the context of real-time communication where services are provided, which have to be **elastic** and provide high scalability, and which have to be **reliable** and provide high fault tolerance.

Questions

- ?
- Which two essential requirements do Reactive Systems fulfill?
- ?
- What characterizes Reactive Systems in respect to their data processing?



In order to implement the maxim “Component Orientation” meaningfully in practice, a so-called **Component System** is usually used. Its functionality allows the management and communication of a large number of components and thus the mastering of the overall complexity of an application.

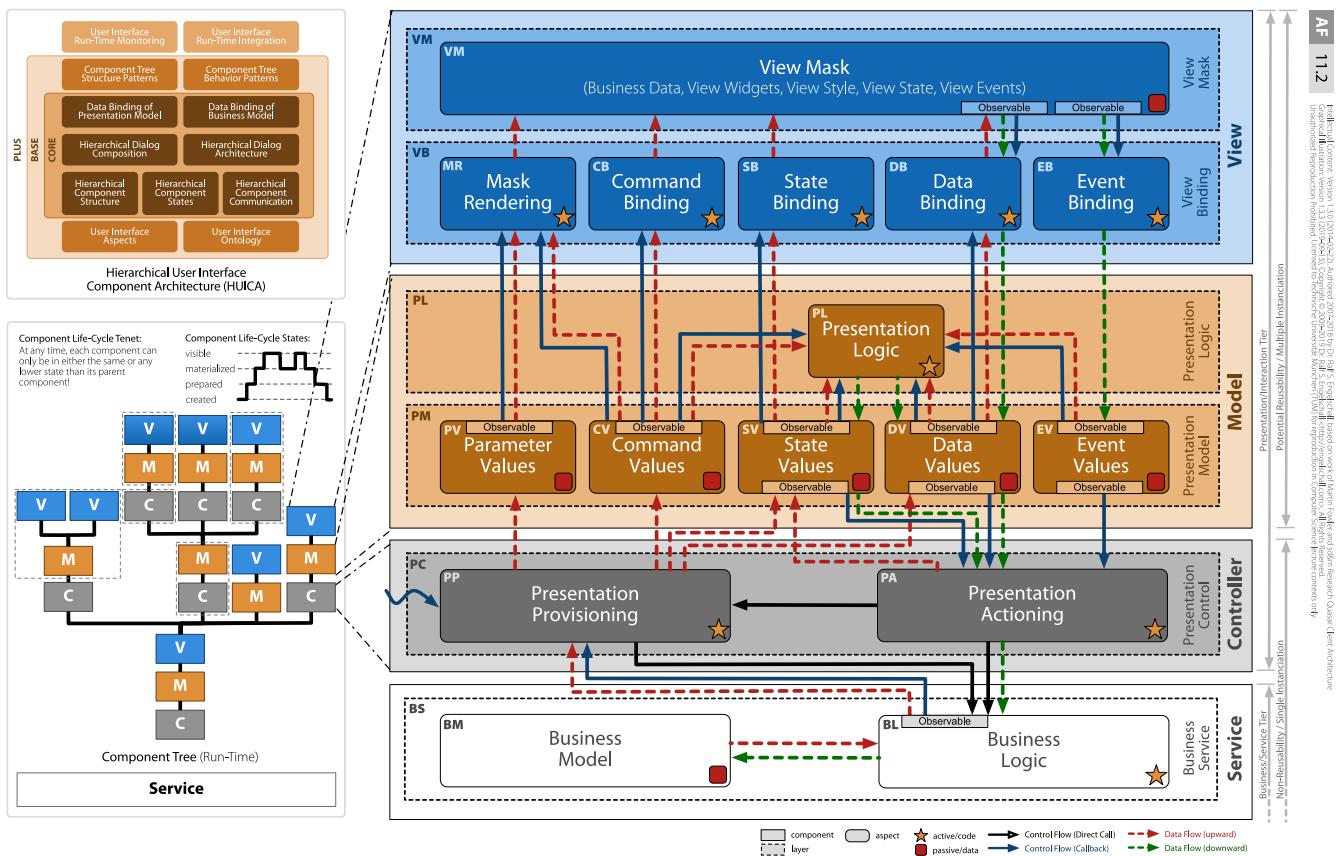
The functionality of a **Component System** usually includes the following essential aspects: **Component Creation & Lookup** for structuring the components, **State Definition & Transition** for the life-cycle of the components, **Property Observation & Modification** for the data of the components, **[Result] Socket Definition & Plugging** for the results of the components, **Event Subscription & Publishing** for the events of the components, **Hook Latching & Execution** for the connection between components and **Service Registration & Calling** for the calls between components.

There are primarily two types of component structures and dependencies that a **Component System** supports: a **Hierarchical Structure** for clients (since user interfaces are inherently hierarchical) and a **Linear Structure** for servers (since the request/response processing is inherently linear).

For the life-cycle of components, a **Component System** usually allows the definition of arbitrary conditions which components can have over the time and which are centrally administered by the **Component System**.

Questions

- ?
- How can you master the overall complexity of managing and communicating of a multitude of components at the maxim **Component Orientation**?
- ?
- Which two types of component structures are most commonly supported by **Component Systems**?

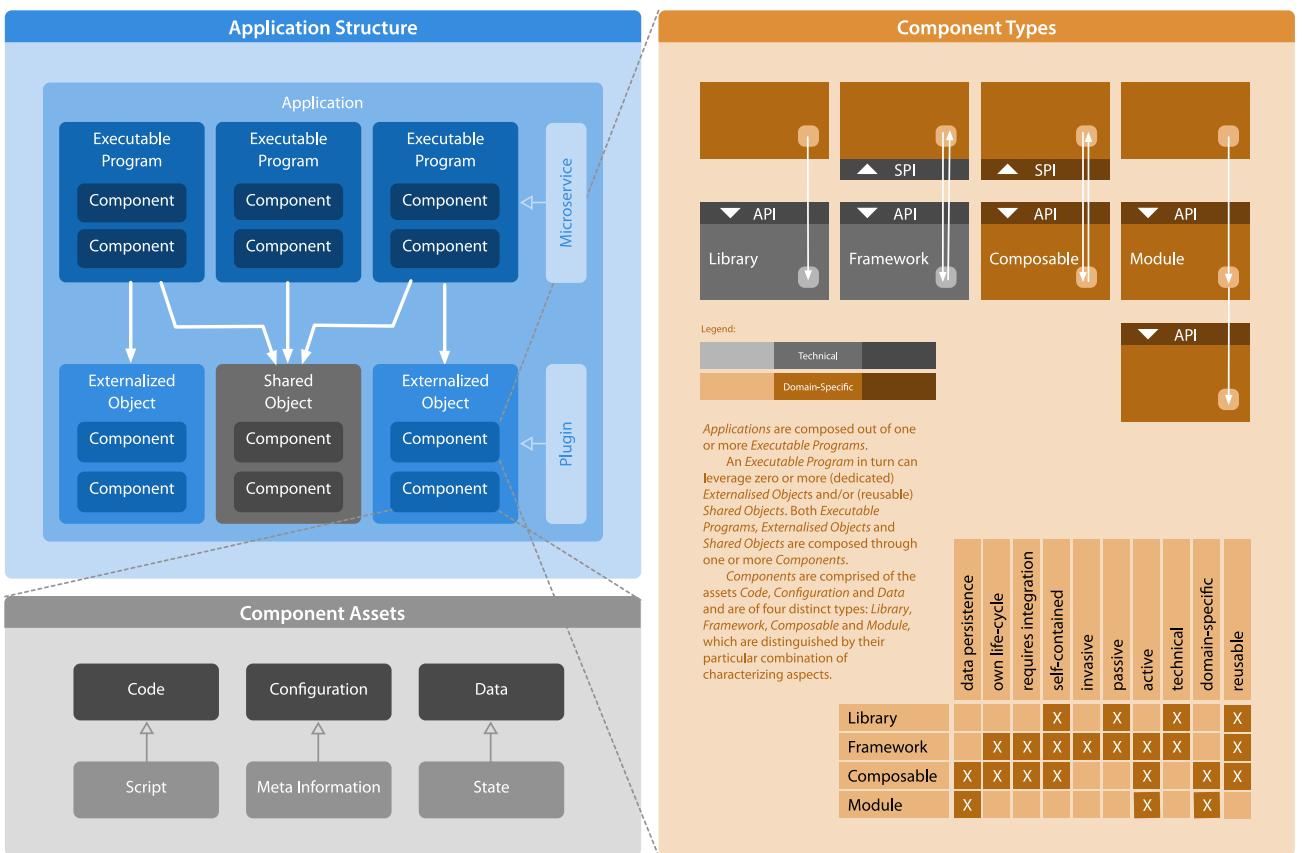


FIXME

Questions

?

FIXME



Applications are composed out of one or more Executable Programs. An Executable Program in turn can leverage zero or more (dedicated) Externalised Objects and/or (reusable) Shared Objects. Both Executable Programs, Externalised Objects and Shared Objects are composed through one or more Components. In a Microservice Architecture, the Executable Programs are called Microservices. In a Plugin Architecture, the Externalised Objects are called Plugins.

There are four distinct types of Components: Library, Framework, Composable and Module. They can be distinguished by their particular combination of characterizing aspects. Most prominently, whether they provide an Application Programming Interface (API) to the consumer of the Component and/or whether they require the consumer of the Component to fulfill some sort of Service Provider Interface (SPI).

Questions

- ?
- What is the main difference between a Library and a Framework?