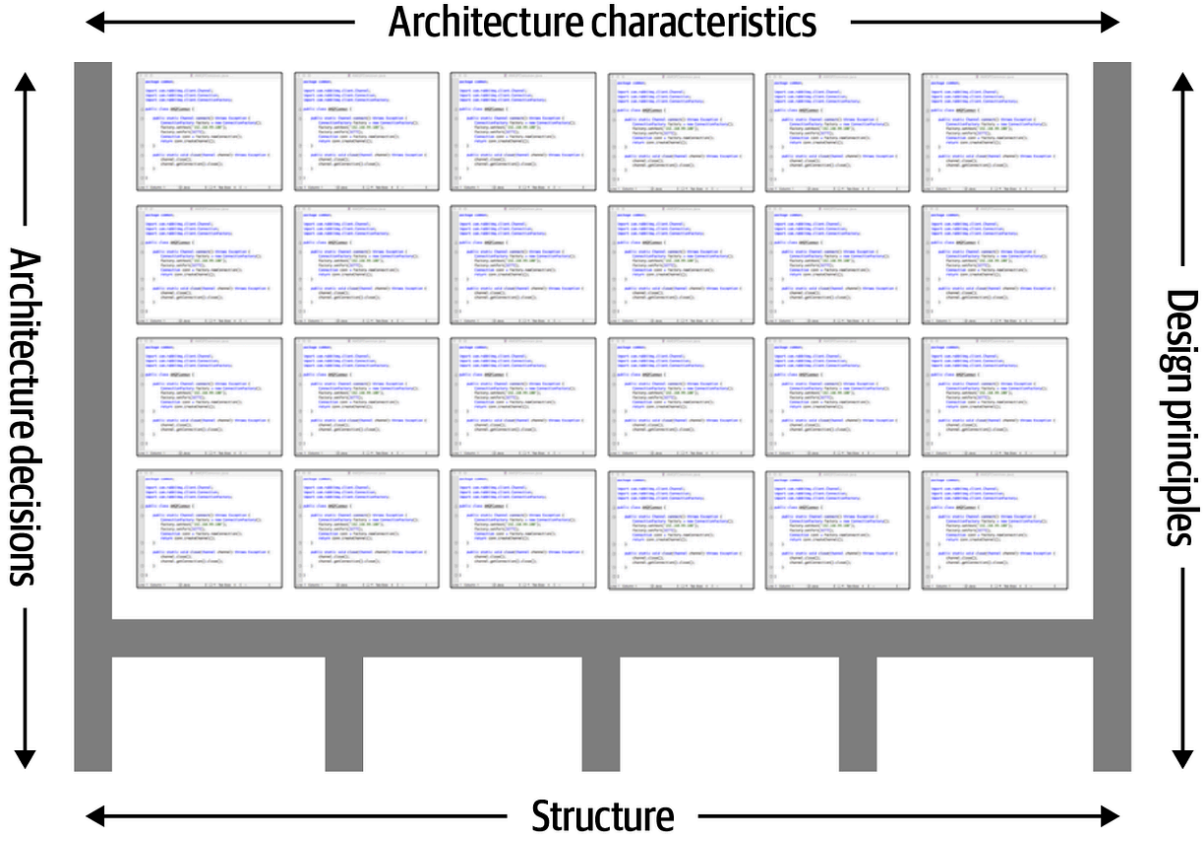
Fundamentals of software architecture: Part I

Architecture

Architecture consists of the **structure** (the type of architecture styles used in the system: microservices, microkernel, etc.) combined with **architecture characteristics** (the "-ilities" that the system must support: availability, scalability, etc.), **architecture decisions** (the constraints of the system: what is and what isn't allowed), and **design principles** (guidelines rather than a hard-and-fast rule).



Architect

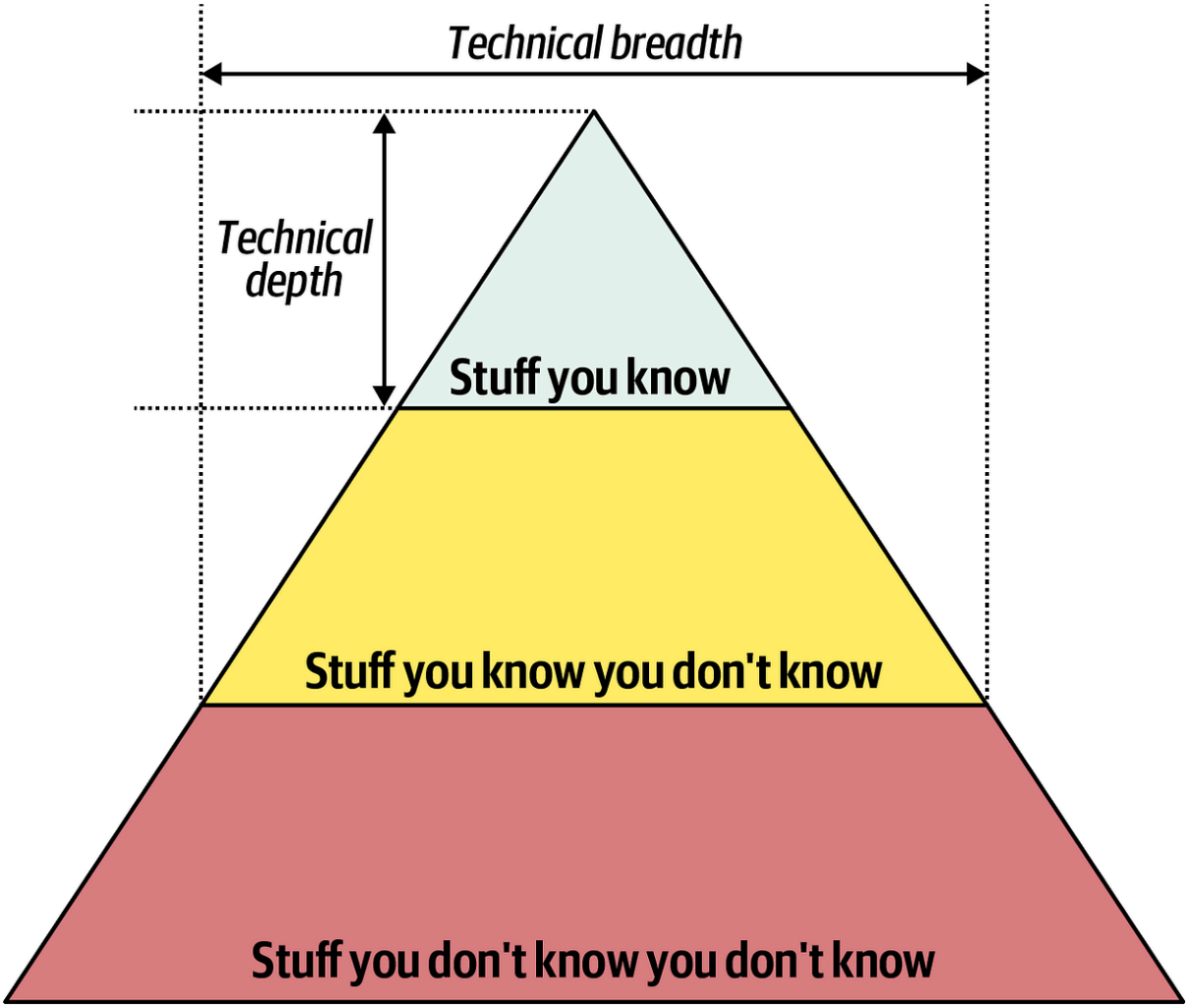
A software architect should always guide rather than specify technology choices. He is expected to:

- **Keep current** with latest trends and be familiar with a variety of technologies.
- Continually **analyse** the architecture, **understanding trade-offs** on each decision (as everything in software

architecture has an advantage and disadvantage, and there are no right or wrong answers)

- **Verify** that development teams are following the architecture decisions and design principles defined, because if not, the architecture will not meet the required architectural characteristics, and the application or system will not work as expected.
- Have a certain level of business **domain expertise**; if not, he cannot communicate with stakeholders and business users and will quickly lose credibility.
- Possess **interpersonal skills**, including teamwork, facilitation, and leadership.
- Understand the political climate of the enterprise and be able to **navigate the politics**, applying basic **negotiation** skills to get most decisions approved. Despite that most decisions made as a developer did not require approval or even a review. almost every decision an architect makes will be challenged, due to increased costs or increased effort (time) involved.

Unlike a **developer**, who must have a significant amount of **technical depth** to perform their job, a **software architect** must have a significant amount of **technical breadth**.



The architect and developer must be on the same virtual team to make this work, as they are both part of the circle of life within a

software project. They must always be kept in **synchronisation with each other** in order to succeed.

Architects should balance architecture stuff with coding, trying to **remain hands-on**, delegating the critical path but doing frequent proof-of-concepts or POCs, tackling some of the technical debt stories, architecture stories, bug fixing, leveraging automation by creating simple command-line tools and analysers to help the development team with their day-to-day, doing frequent code reviews to seek out mentoring and coaching opportunities on the team, etc.

Architecture characteristics

They are also known as nonfunctional requirements or quality attributes. All of them specify a **non-domain design consideration** (they aren't directly related to the domain functionality), **influence some structural aspect of the design** and are **critical** to application success.

An important thing here is that most applications **can only support a few** of all the architecture characteristics, as each of the supported characteristics requires design effort and often has a negative impact on others. So, **never shoot for the best** architecture, but rather the least worst architecture; design it to **be as iterative as possible** and easy to try & change (so you can stress less to find the best on the first attempt).

Some architecture characteristics come from explicit statements in requirements documents, others come from inherent domain knowledge by architects. Have the domain stakeholders select the top three most important characteristics and keep the final list as short as possible.

Operational Characteristics

- **Availability:** how long the system will need to be available.
- **Continuity:** the ability to disaster recovery.

- **Performance:** includes capacity required, and response times expected.
- **Recoverability:** business continuity requirements.
- **Reliability/safety:** assess if the system needs to be fail-safe or it's critical to save lives.
- **Robustness**: the ability to handle error and boundary conditions.
- **Scalability**: the ability to handle a large number of concurrent users without serious performance degradation
- **Elasticity**: the ability to handle bursts of requests.

Structural Characteristics

- **Configurability**: ability for the end users to easily change aspects of the software's configuration.
- **Extensibility**: the ability to plug new pieces of functionality in.
- **Installability**: ease of system installation on all necessary platforms

- **Leverageability/reuse:** ability to leverage common components across multiple products.
- **Localization**: support for multiple languages.
- **Maintainability:** how easy it is to apply changes and enhance the system.
- **Portability:** does the system need to run on more than one platform?
- **Supportability:** what level of technical support (logging and other facilities) is needed by the application?
- **Upgradeability:** ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

Cross-Cutting Characteristics

- **Accessibility**: Access to all your users, including those with disabilities like color blindness or hearing loss.
- **Archivability**: Will the data need to be archived or deleted after a period of time?

- **Authentication**: Security requirements to ensure users are who they say they are.
- **Authorization**: Security requirements to ensure users can access only certain functions within the application.
- **Legal**: What legislative constraints is the system operating in?
- **Privacy**: Ability to hide transactions from internal company employees.
- **Security**: Does the data need to be encrypted in the database/network communications?
- **Usability/achievability**: Level of training required for users to achieve their goals with the application/solution.

Measuring

As you can see, the previous list of architecture characteristics isn't physics, it's partial, many of them have vague meanings, and have widely varying definitions on the industry, so it's important to **agree on an ubiquitous language** and **encourage objective definitions** on your team/s.

Many of the operational characteristics may have obvious measurements, as **response time** for performance. However, measures for structural characteristics are not so obvious. One metric that helps to detect complex code and therefore, bad maintainability, testability, and so on, is **cyclomatic complexity**. It is computed by applying graph theory to code, specifically decision points. As a general rule, a value under 10 is acceptable, but it is preferable to fall under 5 (always check if the function is complex because of poor coding or the domain itself!). Another metric that helps to measure some characteristics, as deployability, is **code coverage.**

Once the architecture characteristics were established, it is important to respect their priorities among the development process; one way to ensure that is using **fitness functions**, as a mechanism to provide an objective integrity assessment and automatically verify important architecture principles (these functions may overlap other mechanisms as metrics, monitors, testing libraries, chaos engineering, etc.). A fitness function may, for

example, check the distance from the main sequence (connascence) and fail if is not acceptable. Another example is Netflix's Simian Army, that ensures that each service responds usefully to all RESTful verbs.

Modularity

Modularity is an organising principle that helps us to better organise complex systems. A **module** is a **logical group of related code**, as it doesn't imply a physical separation (it can be a group of classes in OOP or functions in structured/functional paradigms).

Cohesion

It is a concept of intra-module. It's **how "related" the parts are** to one another. Dividing a high cohesive module would result in increased coupling and poor readability.

Given the subjectiveness of cohesion, computer scientists have developed a structural **metric** to determine cohesion (The

Chidamber and Kemerer Lack of Cohesion in Methods or **LCOM**); however all this metric can find is structural lack of cohesion, as it has no way to determine logically if particular pieces are related.

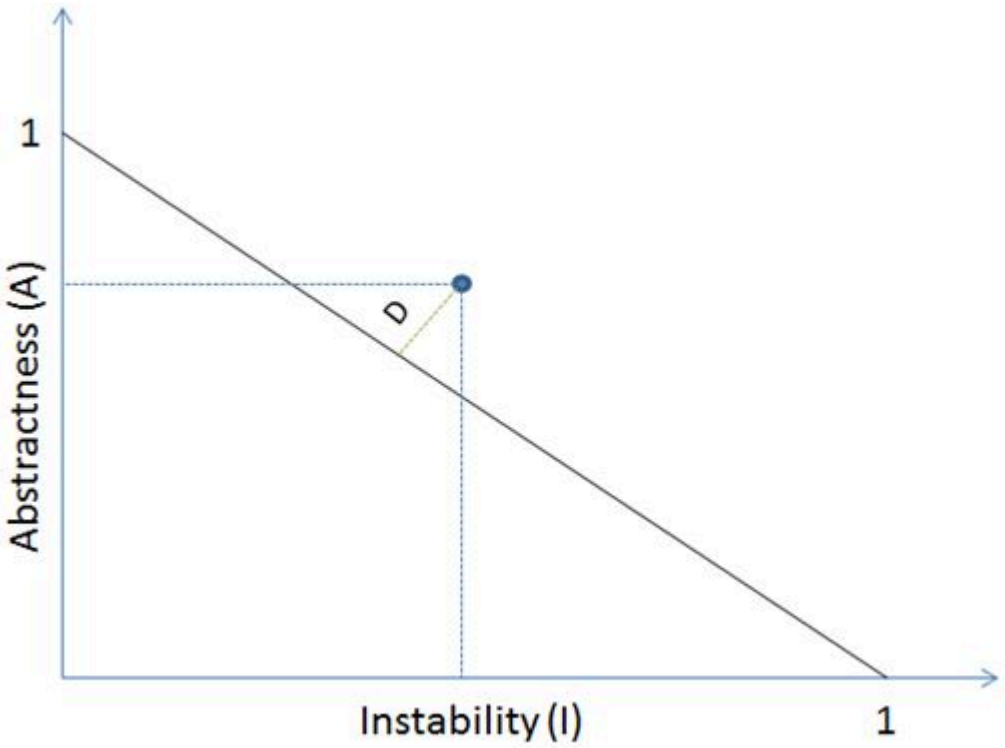
Coupling

It is a concept of inter-module. It is the **degree of interdependence** between software modules. It is based on graph theory, and considers the number of incoming (afferent coupling) and outgoing (efferent coupling) connections to a code artifact. There are some metrics derived from this concept:

- **Instability**: the ratio of efferent coupling to the sum of both (efferent and afferent): [Ce/Ci+Ce]. It determines the **volatility** of a code base; code with higher degrees breaks more easily when changed because of high coupling.
- **Abstractness**: the ratio of the sum of abstract artifacts to the sum of concrete ones. It is used to measure the **degree of abstraction** of the package: code that is too abstract becomes difficult to use and code with too much

implementation and not enough abstraction becomes brittle and hard to maintain.

- **Distance from the Main Sequence:** This metric is used to measure the **balance between stability and abstractness;** classes that fall near this line exhibit a healthy mixture of these two competing concerns.



Ideal packages are either completely abstract and stable (I=0, A=1) or completely concrete and unstable (I=1, A=0). On the other hand,

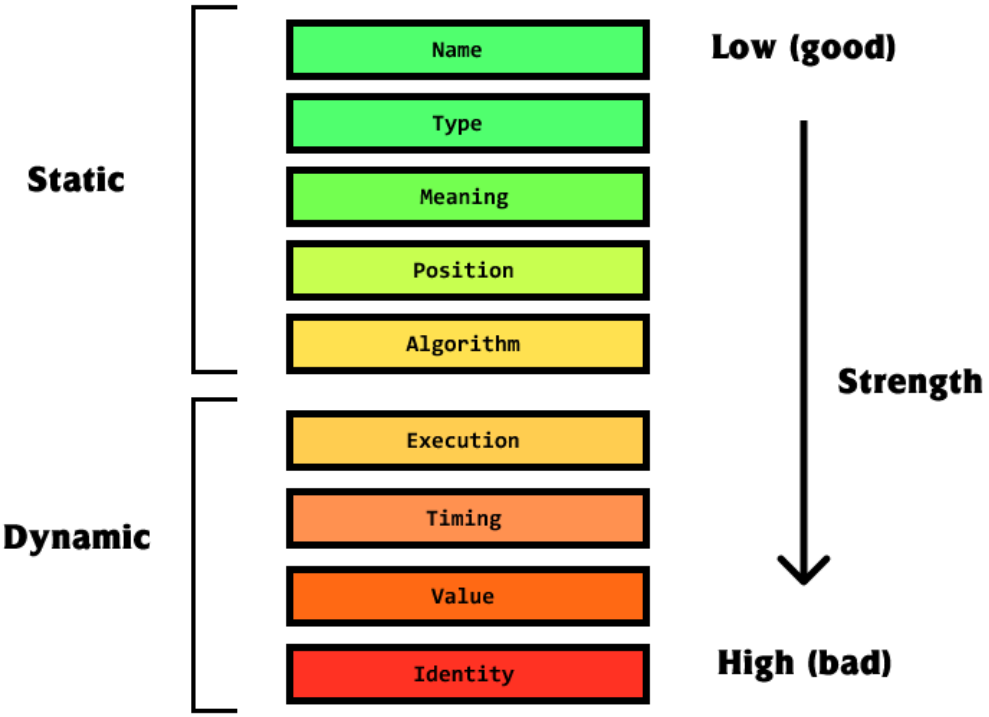
packages with high abstractness and high instability (I=1, A=1) are in the zone of uselessness, and packages with low abstractness and low instability (I=0, A=0) are in the zone of pain as they are difficult to change and cannot be extended because they aren't abstract. This means ideally you should **depend on abstractions and not on implementations**.

Connascence

It provides a **classification of forms of coupling** in a system, and allows reasoning complexity caused by dependency relationships. Each instance of connascence in a codebase must be considered on three separate **axes**:

- **Strength**: the ease with which a developer can refactor that type of coupling.
- **Locality**: how proximal the modules are to each other.
- **Degree**: relates to the size of its impact — does it impact a few classes or many.

There are the following forms, one weaker than others:



- **Name**: when multiple components must agree on the name of an entity.
- **Type**: when multiple components must agree on the type of an entity.
- **Meaning**: when multiple components must agree on the meaning of specific values.

- **Position**: when multiple components must agree on the order of values.
- **Algorithm**: when multiple components must agree on a particular algorithm.
- **Execution** (order): when the order of execution of multiple components is important.
- **Timing**: when the timing of the execution of multiple components is important.
- **Value**: when several values must change together.
- **Identity**: when multiple components must reference the entity.

The idea is simple: the more remote the connection between two clusters of code (locality), the weaker the connascence between them should be. More detail and some examples on https://connascence.io/.

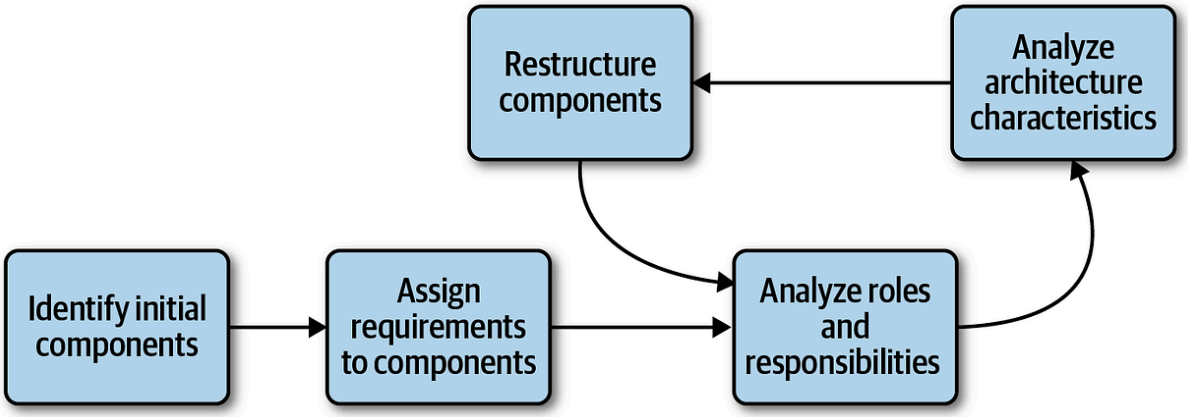
Components

Components are the **physical manifestation of a module** (collection of related code) and form the fundamental modular **building block in architecture**. Too fine-grained a component design leads to too much communication between components to achieve results but too coarse-grained components encourage high internal coupling, which leads to difficulties in deployability and testability, as well as modularity-related negative side effects.

The simplest component are **libraries** (that wraps code at a higher level of modularity than classes); they also appear as **subsystems or layers**, as the deployable unit of work for many event processors, or as **services** (that tends to run in its own address space and communicates via low-level networking protocols like TCP/IP or higher-level formats like REST or message queues).

Generally the component is the lowest level of the software system an architect interacts directly with. Developers typically take components, jointly designed with the architect role, and further subdivide them into classes, functions, or subcomponents.

The component **identification flow** may be something like this:



There are some useful techniques for doing it, as:

- **ACTOR/ACTIONS APPROACH**: identify actors who perform activities with the application and the actions those actors may perform.
- **EVENT STORMING**: the team tries to determine which events occur in the system based on requirements and identified roles, and build components around those event and message handlers (inspired by DDD).
- **WORKFLOW APPROACH**: models the components around workflows, much like event storming, but without

the explicit constraints of building a message-based system.

based on Fundamentals of Software Architecture by Richards, Mark; Ford, Neal (2020-01-28).