Clean architecture [3]

An architecture that allows for replacement details and is easy to verify.

Simple architecture

An architecture that is easy to understand. Simplicity is, however, subjective.

- Consistent design decisions
  One problem has one solution. Similar problems are solved similarly.

- Number of concepts/technologies
  Simple solutions make use of only a few different concepts and technologies.

- Number of interactions
  The less interactions the simpler the design.

- Size
  Small systems/components are easier to grasp than big ones. Build large systems out of small parts.

- Modularity
  Build your system by connecting independent modules with a clearly defined interface (e.g. with adapters).

Flexible architecture

An architecture that supports change.

Separation of concerns

Divide your system into distinct features with as little overlap in functionality as possible so that they can be combined freely.

Software reflects user's mental model

When the structure and interactions inside the software match the user's mental model, changes to the real world can more easily be applied in software.

Abstraction

Separating ideas from specific implementations provides the flexibility to change the implementation. But beware of ‘over abstraction’.

Interface slimness

Fat interfaces between components lead to strong coupling. Design the interfaces to be as slim as possible. But beware of ‘ambiguous interfaces’.

Prefer composition over inheritance

Inheritance increases coupling between parent and child, thereby limiting reuse.

Tangle-free cycle-free dependencies

The dependency graph of the elements of the architecture has no cycles, thus allowing localized changes.

Evolvable architecture

An architecture that is easy to adapt step by step to keep up with changes.

Matches current needs, not the future

The architecture of the current system should match the current needs (functional and non-functional) - not some future ones. This results in simpler, easier to understand solutions. Otherwise, the risk of waste is very high.

No dead-ends, architecture can be extended/adapted

The current architecture should be extensible and adaptable so that future needs can be addressed. When evaluating different alternatives, choose one that is open for change.

Architecture agnostic components

When components don’t care about which architecture they run in, the architecture can be changed without having to rewrite the components.

Sacrificial architecture [4]

When the software has outlived its architecture, throw the architecture away and start over. This mindset can be used to build a first version with a very simple architecture, then start over for the next.

Rolling refactoring [5]

When a necessity of a concept is introduced, then the old one is refactored out step by step. There can be at most two versions of a concept in an application (and it should be temporary).

Agile architecture

An architecture that supports agile software development by enabling the principles of the Agile Manifesto [6].

- Allow change quickly
  The architecture allows quick changes through flexibility and evolvability.

- Verifiable at any time
  The architecture can be verified (fulfills all quality aspects) at any time (e.g. every Sprint).

- Rapid deployment
  The architecture supports continuous and rapid deployment so that stakeholders can give feedback continuously.

- Always working
  The system is always working (probably with limited functionality) so that it is potentially shipped any time/at end of Sprint. Use assumptions, simplifications, simulations, shortcuts, hard-coding to build a walking skeleton.

Workshop

Use a top-down approach to find the architecture.

1. Context
   What belongs to your system and what does not? Which external services will you use?

2. Break down into parts
   Split the whole into parts by applying separation of concerns and the single-principle principle.

3. Communication
   Which data flows through which call, message or event from one part to another? What are the properties of the channels (sync/async, reliability…)?

4. Repeat for each part
   Repeat the above-mentioned three steps for each part as if it were your system. A part is a bounded context, subsystem or component.

Defeat decisions

Decide only things you have enough knowledge about. Otherwise find a way to defer the decision and build up more knowledge. A good architecture allows you to defer most decisions.

Decision delegation

Build the (part of a) system in a way that doesn’t require any decision, by making some other (part of the) system responsible that can be implemented later. E.g. instead of deciding how to persist data, make the code calling your code responsible for passing all needed data to your code. This allows you to build your whole business logic and decide about persistence when implementing the host that runs the business logic.
Clean, simple flexible evolvable agile Architecture

Priorities

Simplicity before generality [7]
Concrete implementations are easier to understand than generalised concepts.

Hard-coded before configurable
Configurability leads to false constructs or polymorphism inside the code, resulting in more complicated code.

Don’t use reuse [7]
Don’t design for reuse before the code has never actually been used. This leads to overgeneralisation, innap interfaces and increased complexity.

Working on optimising
First, make it work, then optimise. Premature optimisation leads to more complex solutions or to local instead of global optimisations.

Quality attributes before functional requirements
Use quality scenarios to guide your architectural decisions because most of the times, quality attributes have more impact than functional requirements.

Combined small systems over a single big system
Big systems are more complicated to comprehend than a combination of small systems. But beware of complexity hidden in the communication between the systems.

Principles [8]

The teams that code the system, design the system.
Teams themselves are empowered to define, develop, and deliver software, and they are held accountable for the results.

Build the simplest architecture that can possibly work.
Simplicity leads to comprehensibility, changeability, low defect introduction.

When in doubt, code it out.
Get real feedback from running code, then decide.

Testing is an integral part of building software, not an afterthought.

System architecture as a role collaboration.
The whole team participates in architecture decisions.

There is no monopoly on innovation.
Every team member has time to innovate (spikes, hackathons, pet projects).

Tips and tricks

Start with concepts, not with technologies.
Don’t think in technologies, think in concepts. Then choose technologies matching the concepts and adapt concepts to technological limitations.

Think about your envisioned architecture, but also lay a way from here to there.

Break your architecture work into steps. Use assumptions and simplifications in early steps. Always make sure that there is a path from the current architecture to the envisioned architecture.

Most of the time, persistence is a secondary thought.
You always have some data. But that is no reason to start your design with the database. Business logic and workflows are more important.

Decouple from environment
Design everything so that it has to know nothing about its environment.

Prototypes, proof of concepts, feasibility studies
Use architecture patterns as inspiration, not as solutions. Architecture patterns are good examples of solutions to specific problems. Use them to find solutions for your problems and do not apply them to your problems.

Architectural aspects

Persistence
Form of data (document-based, relational, graph, key-value), backup, transactions, size of data, throughput, replication, availability, concurrency.

Translation (UI and data)
Static (e.g. resources) vs. dynamic, switchable during implementation/installation/start-up/runtime.

Communication between parts
Asynchronous/synchronous, un/reliable, latency, throughput, availability of connection, method call/events/messages.

Simplicity
Run on multiple threads/processes/machines, availability, consistency, redundancy.

Security
Authentication, authorisation, threats, encryption (of communication and data). See [9].

Journaling, auditing
Operations, granularity, access to journal, tampering, regulatory.

Reporting
Access to data (production/dedicated database/data warehouse), delivery mechanism (synchronous/asynchronous), formats (Web, PDF, ...).

Data migration, data import
Available time frame for migration/import, data quality, default values for missing values, value merging/splitting.

Reliability
Release as one, per service or per component (e.g. plug-ins). Automatic or manual release.

Versioning
One product vs. a product family, technical/marketing version, manually or automatically generated, releases/service packs/hot fixes, SemVer. [10]

Backward compatibility
APIs, data (input/output/persisted), environment (e.g. old OS).

Response times
Service time (actually performing the work) + wait time + transmission time.

Archiving data
Data growth rate, access to archived data, split relations in relational data.

Distribution
 Beware of the fallacies of distributed computing: the network is reliable, latency is zero, bandwidth is infinite, the network is secure, topology doesn’t change, there is one administrator, transport cost is zero, the network is homogenous.

Public interfaces
Version, immutability and stability of contracts and schemas.

Documentation
Questions to ask yourself [11]
Who is the consumer? What do they need? How do you deliver the documentation to them? How do you know when they are ready for it? How do you produce it? What input do you need to produce it?

Manual and automatic production
Manual: someone writes the documentation, high risk of being out-of-date, very flexible
Automatic [12]: generated from code, can be regenerated anytime and is therefore never out of date, finding right level of abstraction is hard. Works good for state machines, bootstrapping mechanics, and structural breakdown.

About now, not the future
Only document what you did, not what you want to do.

Shared
The whole team participates in producing the documentation.

Architecture smells

Causes: applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviours.

Overlayered architecture
When there are layers on layers on layers ... in your application. Not providing abstraction, lots of boilerplate code.

Overabstraction
Too abstract to be understandable. Concrete designs are easier to understand.

Overconfigurability
Everything is configurable because no decisions were made how the software should behave.

Overkill architecture
A simple problem with a complex (however technically interesting) solution.

Futureistic architecture
The architecture wants to anticipate a lot of future possible changes. This adds complexity and most likely also waste.

Technology enthusiastic architecture
Lots of new cool technology is introduced just for the sake of it.

Paper tiger architecture
The architecture exists only on paper (UML diagrams) with no connection to the reality.

Connector envy [13]
A component doing the job that should be delegated to a connector: communication (transfer of data), coordination (transfer of control), conversion (bridge different data formats, types, protocols), and facilitation (load-balancing, monitoring, fault tolerance).

Scattered parasitic functionality [13]
A single concern is scattered across multiple components and at least one component addresses multiple orthogonal concerns.

Ambiguous interfaces [13]
Ambiguous interfaces are interfaces that offer only a single general entry point into a component (e.g. pass an object, or general purpose events over an event bus). They are not exploitable.

Extraneous adjacent connector [13]
Two connectors of different types are used to link a pair of components. E.g. event (asynchronous) and service call (synchronous).

Event: loosely coupled \(\rightarrow\) availability, replicability.

Method call: easy to understand.

Both: neither.

Bibliography


Legend:

<table>
<thead>
<tr>
<th></th>
<th>DO</th>
<th>DON'T</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work by Urs Enzler is licensed under a Creative Commons Attribution 4.0 International License.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Urs Enzler www.bbv.ch June 2015 V1.0