Clean simple flexible evolvable agile Architecture

Why is a clean, simple, flexible, evolvable, and agile architecture important?

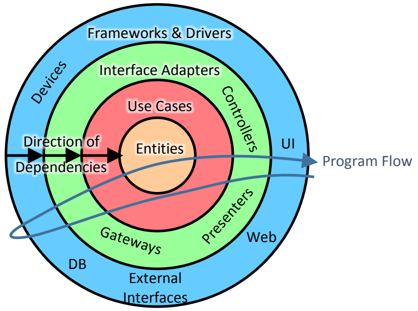
Software architecture is the high level structure of a software system, the discipline of creating such structures, and the documentation of these structures. [1]
It is the set of structures needed to reason about the software system, and comprises the software elements, the relations between them, and the properties of both elements and relations. [2]

In today's software development world, requirements change, environments change, team members change, technologies change, and so should the architecture of our systems.

The architecture defines the parts of a system that are hard and costly to change. Therefore we are in need of a clean, simple, flexible, evolvable, and agile architecture to be able to keep up with all the changes surrounding us.

Clean architecture [3]

An architecture that allows to replace details and is easy to verify.



Entities: Entities encapsulate enterprise-wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions.

Use cases: Use cases orchestrate the flow of data to and from the entities, and direct those entities to use their enterprise-wide business rules to achieve the goals of the use cases.

Interface adapters: Adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as a database or the Web.

Frameworks and drivers: Glue code to connect UI, databases, devices etc. to the inner circles.

Program Flow: Starts on the outside and ends on the outside, but can go through several layers (user clicks a button, use case loads some entities from DB, entities decide something that is presented on the UI)

Dependency management

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

Source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle. Use dependency inversion to build up the system (classes in an outer circle implement interfaces of an inner circle or listen to events from inner circles).

Independent of frameworks

The architecture does not depend on the existence of some library of feature-laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their technical constraints.

Testable

The business rules and use cases can be tested without UI, database, Web server, or any other external element.

Independent of system boundaries (UI, database, …)

The UI, database, or any other external element can easily change without any impact on use cases and business rules.

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.

A reasonable amount of components with only efferent coupling and most of the others with preferably only afferent coupling.

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 in 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-/cycle-free dependencies

The dependency graph of the elements of the architecture has no cycles, thus allowing locally bounded 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 extendable 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 new version 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 (fulfils 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 shippable any time/at end of Sprint. Use assumptions, simplifications, simulators, shortcuts, hard-coding to build a walking skeleton.

Workflow

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-responsibility principle.

3. Communication

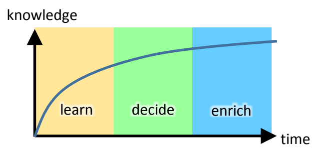
Which data flows through which call, message or event from one part to another? What are the properties of the channels (sync/async, reliability, …)

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.

Defer 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.



Abstraction

Use an abstraction to hide details so that you don't have to decide about the details, but can use a simulation/fake at first to build up more knowledge.

Simplification

Simplify the problem so that a decision can be made and work can progress. Use this to break free from a blocking state, but be aware of the risks a wrong decision could have.

Wilful ignorance

Refuse to decide and wait until more knowledge about the problem and its potential solutions is built up.

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.

Architecture influencing forces

Quality attributes

The needed quality attributes (functionality, reliability, usability, efficiency, maintainability, portability, …) are the primary drivers for architectural decisions.

Team know-how and skills

The whole team understands and supports architecture and can make design decisions according to the architecture.

Easiness of implementation

How easy an envisioned architecture can be implemented is a quality attribute.

Cost of operations

Most costs of a software system accrue during operations, not implementation.

Risks

Every technology, library, and design decision has its risks.

Inherent opportunities

Things the architecture would allow us to do (but without investing any additional effort because we may never need it).

Technology churn

Availability of new (better) technologies, resulting in a need for architecture change.

Trade-offs

Designing an architecture comprises making trade-offs between conflicting goals. Trade-offs must reflect the priorities of quality attributes set by the stakeholders. Trade-offs should be documented and communicated to all stakeholders.

Architecture degrading forces

Architectural drift

Introduction of design decisions into a system's actual architecture that are not included in, encompassed by, or implied by the planned architecture.

Architectural erosion

Introduction of design decisions into a system's actual architecture that violate its planned architecture.

Architecture killers

Split brain

Different parts of the system claim ownership of the same data or their interpretation resulting in inconsistencies and difficult synchronisation.

Coupling in space and time

E.g. shared code to remove duplication hinders independent advancements, a service that needs other services to be up and running, an `initialise` method that has to be called prior to any other method on the class (better use constructor injection or a factory).

Dead-end

A design decision that prevents further adaptability without a major refactoring or rewrite.

bbv

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 if/else constructs or polymorphism inside the code, resulting in more complicated code.

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

Working before optimised
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 building 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.

They build it, they test it.
Testing is an integral part of building software, not an afterthought.

System architecture is 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 project).

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
Break risks and grow knowledge fast, then decide.

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 calls/events/messages.

Scaling
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.

Releasability
Release as one, per service or per component (e.g. plug-in). 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 homogeneous.

Public interfaces
Versioning, 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 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.

Futuristic 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 explorable.

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 → availability, replicability.
Method call: easy to understand.
Both: neither.

Bibliography

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford, Documenting Software Architectures: Views and Beyond, 2nd ed., Boston: Addison-Wesley, 2010.

[2] Wikipedia, "Software architecture," [Online]. Available: http://en.wikipedia.org/wiki/Software_architecture. [Accessed 2015].

[3] R. C. Martin, "The Clean Architecture," [Online]. Available: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html. [Accessed 2015].

[4] M. Fowler, "Sacrificial Architecture," [Online]. Available: http://martinfowler.com/bliki/SacrificialArchitecture.html.

[5] [Online]. Available: https://lostechies.com/jimmybogard/2015/01/15/combating-the-lava-layer-anti-pattern-with-rolling-refactoring/.

[6] "Manifesto for Agile Software Development," [Online]. Available: http://agilemanifesto.org/principles.html. [Accessed 2015].

[7] K. Henney. [Online]. Available: http://www.artima.com/weblogs/viewpost.jsp?thread=351149.

[8] D. Leffingwell, "Principles of Agile Architecture," [Online]. Available: http://scalingsoftwareagilityblog.com/wp-content/uploads/2008/08/principles_agile_architecture.pdf. [Accessed 2015].

[9] [Online]. Available: https://www.owasp.org.

[10] [Online]. Available: http://semver.org/.

[11] [Online]. Available: http://thinkrelevance.com/blog/2013/10/07/begin-with-the-end-in-mind.

[12] [Online]. Available: http://www.planetgeek.ch/2014/06/17/effective-teams-know-your-code/.

[13] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, "Toward a Catalogue of Architectural Bad Smells," [Online]. Available: http://softarch.usc.edu/~josh/pubs/qosa_2009.pdf. [Accessed 2015].

Legend:
- DO
- DON'T

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