SOLID principles

Program slicing:

- originaly the program is like a bunch of lines of code executed sequentially
- extract functions and the program is like a composition of functions
- group fundions into modules or classes and the program has structure, so you can find easily the function you are interested in

(years)
After a while of the program evolving, responsabilities settles and a kind of domain structure emerges.

- modular monolith architecture
  - (micro)services from each module
    distributed architecture

distributed is always ~~may~~ more complex than monolith,
but sometimes this added complexity is worth for the (horizontal) scalability it gives to us.

Open for extension
but closed for modification

Liskov substitution
(design by contract)
(interface)

Single responsability
for each function or module

SoC Separation of Concerns

well-defined dependencies between fuctions / modules
(data transfers)

Low coupling
High cohesion

Interface segregation

Dependency inversion

Trade-off between complexity (of slicing) and benefits (maintability) (scalability)

- reduce duplication (DRY - Don't Repeat Yourself) — idealy → only one point to change for each responsability
- flexibility to evolve — idealy → you can change inside each part | module | class without affecting any other
- allows interface creation — idealy → all dependencies are well stated
- reduces cognitive load — when combined with good naming, you can focus at different levels of abstraction
  (the names of fncIias | modules | parameters convey clearly what each of them does)

---

monolith → all modules/services in one single binary → you build deploy all toghether

(micro) services → each service in it's own binary → you build deploy each one independently

↓

if you have realized each service right, you can deploy more of one or another service as you need to cope with the demand of each one.

you can also add redundancy to cope with failures in a system

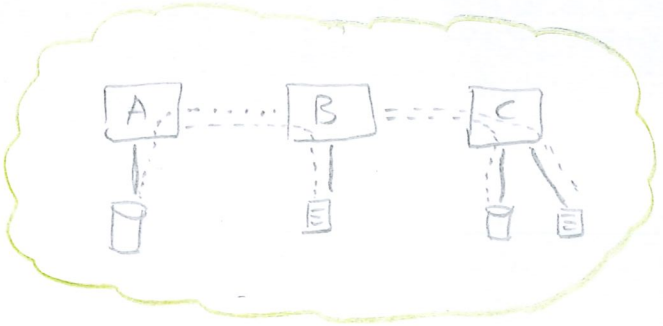
↓

complexity ad coordination needs are bigger

Decoupling

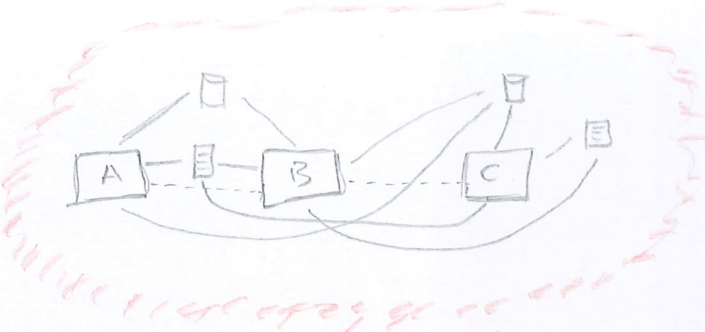
- In monoliths you must care about interfaces,
  but all modules (service) share build, deployment, libraries and repositories
- In microservices, each service (module) has it's own
  - build
  - deployment
  - libraries
  - repositories

(and you must care about relationships (APIs) (contracts))



Hidden Coupling

There is a thing worse than a higly coupled monolith (Big Ball of Mud)
and it is a distributed monolith! (higly coupled microservices)



Invocation Consequences → pay attention to network invocation burden (async / latency delays / errors / timeouts)
→ pay attention to reliability or congestion issues

Addressing Consequences → pay attention to (DNS) hosts, (IP) ports, load balancers, proxies, ...
- where is B?
- is B alive?

more complicated as more services we have to coordinate

Dependency Isolation Consequences → pay attention to versioning of each (API) service

8 Fallacies of Distributed Computing

observability is vital!