CLEAN ARCHITECTURE

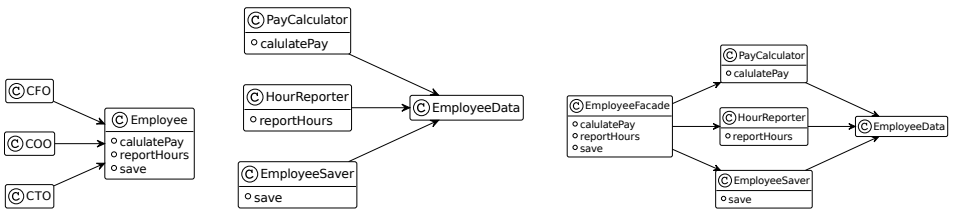
Authors Robert C. Martin, James Grenning, Simon Brown

ARCHITECTURE

From Grady Booch, architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

SINGLE RESPONSIBILITY PRINCIPLE

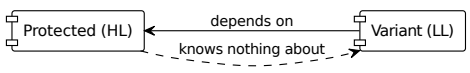
A module should **be responsible to one, and only one actor**, so that actors are not coupled.



(a) Bad design (b) Good design (c) Using a facade

OPEN CLOSE PRINCIPLE

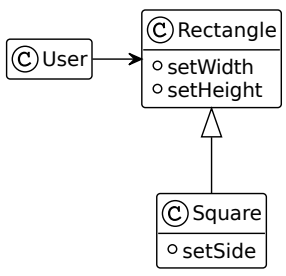
A software artifact should be open for extension but close for modification: arrange the components into a **dependency hierarchy that protects higher-level components** from changes in lower-level components.



This can be implemented by following the *Dependency Inversion Principle*

LISKOV SUBSTITUTION PRINCIPLE

All the subtypes S of a type T should be fully substituable by T: **interfaces of T and S should be exactly the same**.



This illustrates the square/rectangle problem, which violates the principle, as `Square` and `Rectangle` interfaces and behaviours are not compatible. It forces to add extra mechanisms to distinguish each types during runtime, which lowers the software maintainability.

INTERFACE SEGRAGATION PRINCIPLE

The client should not depend on something that it does not use. Consider a class whose responsibility is persisting data on the harddrive. Splitting the class into a read- and a write part would not make practical sense. But some clients should only use the class to read data, some clients only to write data, and some to do both. Applying ISP here with three different interfaces would be a nice solution.

DEPENDENCY INVERSION PRINCIPLE

The most flexible systems are those which source **dependencies refer only to abstractions**, not to concretions. It is possible to invert dependencies using Object-Oriented techniques:

(a) Both sides dependencies (b) All dependencies from B to A

- Do not refer to concrete classes,
- Do not derive a concrete class,
- Do not redefine concrete methods.

PRINCIPLES OF COMPONENTS COHESION

- Reuse/Release Equivalence: the granule of reuse is the granule of release;
- Common Closure: a component should not have mutliple reasons to change (*SRP*);
- Common Reuse: classes in a component are inseparable (*ISP*).

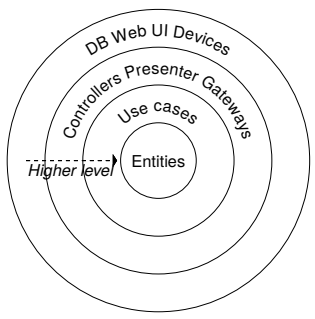
STABLE & ABSTRACT DEPENDENCY PRINCIPLES

These 2 rules can be applied at the component-level and state that a component should **depend in the direction of stability** and **be as abstract as it is stable**.

- Let $I(C)$ be a measure of the instability of a component C: $I(C) = \frac{dep_{out}(C)}{dep_{out}(C)+dep_{in}(C)}$. The *Stable Dependency Principle* (*OCP* applied to component) states that if B depends on A, we should have $C(A) < C(B)$.
- Let $A(C)$ be a measure of the abstractness of a component C: $A(C) = \frac{N_{interfaces}(C)}{N_{class}(C)}$. The *Abstract Dependency Principle* states that component should be on the curve represented by : $A(C) = 1 - I(C)$.

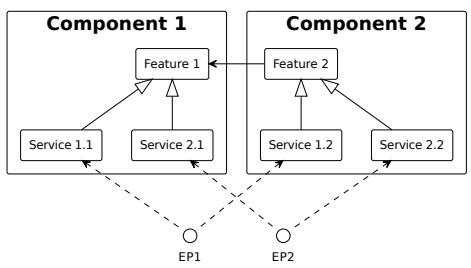
DEPENDENCY RULE

The following rules can be applied in the whole project:



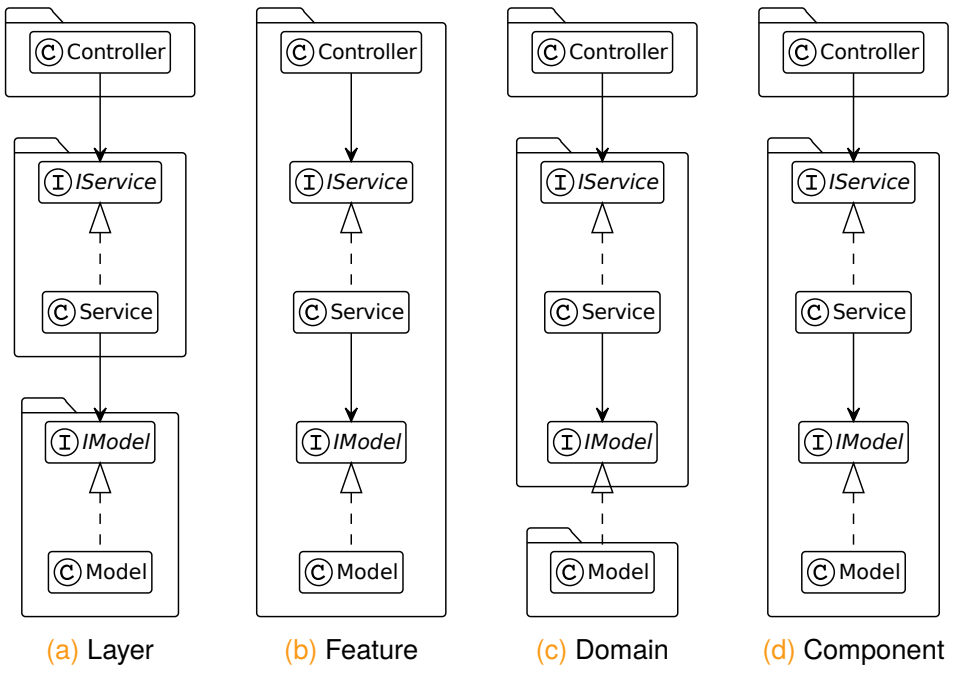
- inner layers know nothing about outer layers;
- source code dependencies point inwards to higher-level policies;
- **isolated data structures** are passed across boundaries.

SERVICES ARCHITECTURE



Services in themselves are not architecturally significant elements. Architectural boundaries do not fall between services, but they run through the services, dividing them in components.

DESIGN & CODE ORGANIZATION



(a) Layer (b) Feature (c) Domain (d) Component

(a) is easy and quick to get started, (b) is easy but removes all boundaries, (c) maintains boundaries but bypass is still possible via visibility incorrectness and (d) has a good separation and architecture principles can be statically verified.