University of Oviedo

Universidad de Oviedo

EN
English

School of Computer Science

Modularity

SOFTWARE ARCHITECTURE

School of Computer Science

2023-24

Jose E. Labra Gayo

Modularity

Decomposing the project in modules at development time
Modules can be developed independently

Modularity

- Big Ball of Mud
- Modularity definitions
- Modularity recommendations
  - SOLID, Cohesion, Coupling, Connascence, Robustness, Demeter, Fluid interfaces
- Modularity styles
  - Layers
  - Aspect Oriented decomposition
  - Domain based decomposition

Big Ball of Mud

Big Ball of Mud

- Described by Foote & Yoder, 1997

Elements

- Lots of entities intertwined

Constraints

- None

Big Ball of Mud

- Quality attributes (?)
  - Time-to-market
    - Quick start
      - It is possible to start without defining an architecture
      - Incremental piecemeal methodology
    - Solve problems on demand
  - Cost
    - Cheap solution for short-term projects

Big Ball of Mud

Problems

- High Maintenance costs
- Low flexibility at some given point
  - At the beginning, it can be very flexible
  - After some time, a change can be dramatic
- Inertia
  - When the system becomes a *Big Ball of Mud it* is very difficult to convert it to another thing
    - A few *prestigious* developers know where to touch
    - *Clean* developers run away from these systems

Big Ball of Mud

Some reasons

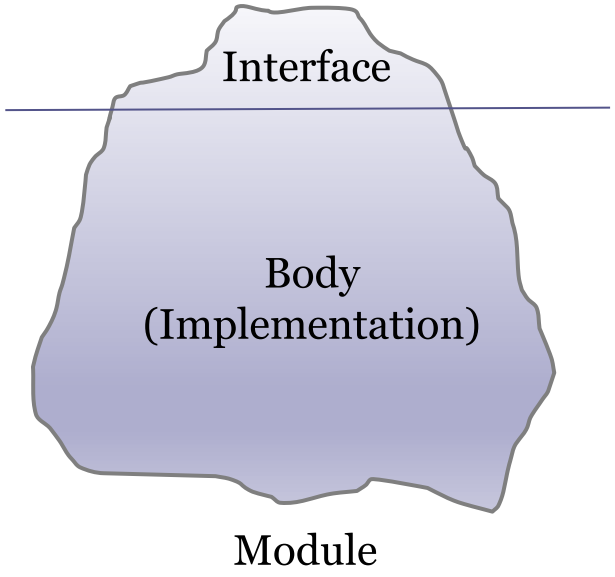
- Throwaway code:
  - You need an immediate fix for a small problem, a quick prototype or proof of concept
  - When it is good enough, you ship it
- Piecemeal growth
- Cut/Paste reuse
  - Bad code reproduced in lots of places
- Anti-patterns and technical debt
  - Bad smells
  - Not following clean code/architecture

Definitions of modules

Module:
- Piece of software the offers a set of responsibilities
- It makes sense at building time (not at runtime)
- Separates interface from body

Interface
- Describes what is a module
- How to use it ≈ Contract

Body
- How it is implemented



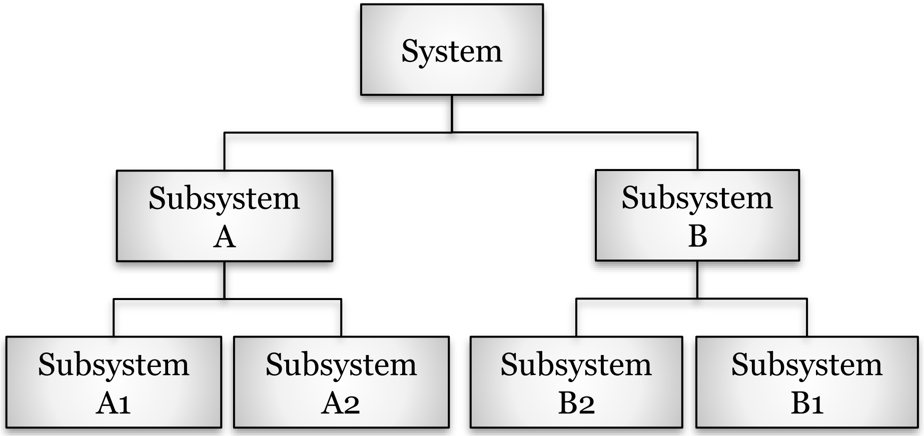
Modular decomposition

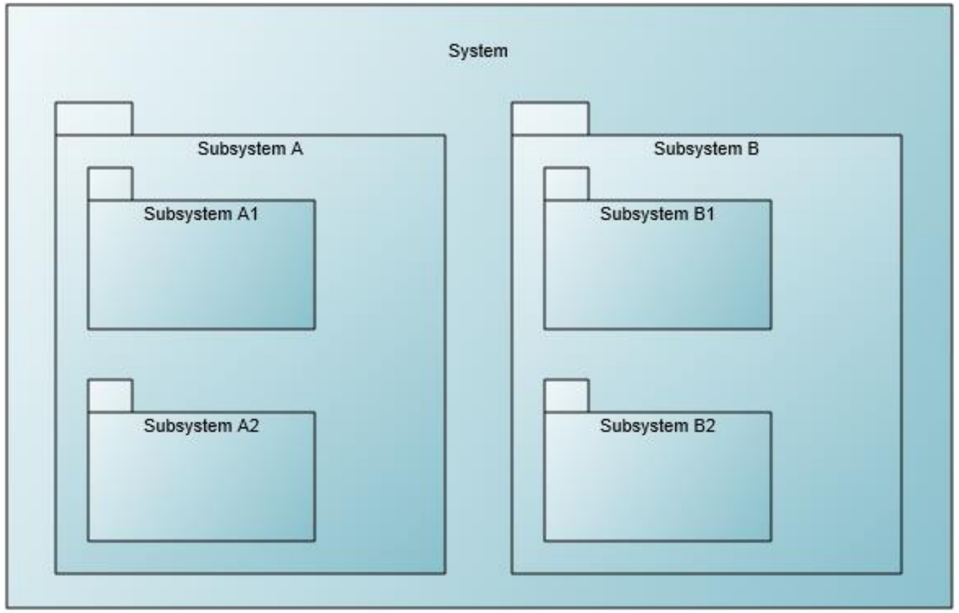
Relationship: *is-part-of*

Constraints

- No cycles are allowed
- Usually, a module can only have one parent

Several representations





Modularity Quality attributes

- Communication
  - Communicate the general aspect of the system
- Maintainability
  - Facilitates changes and extensions
  - Localized functionality
- Simplicity
  - A module only exposes an interface - less complexity
- Reusability
  - Modules can be used in other contexts
  - Product lines
- Independence
  - Modules can be developed by different teams

Modularity challenges

- Bad decomposition can augment complexity
- Dependency management
  - Third parties modules can affect evolution
- Team organization
  - Modules decomposition affects team organization
- Decision: Develop vs buy
  - COTS/FOSS modules

Modularity recommendations

- SOLID design principles
- Cohesion
- Coupling
- Connascence
- Robustness: Postel’s law
- Demeter's Law
- Fluid interfaces

SOLID design principles

SOLID principles can be applied to clases and modules

- SRP (Single Responsability Principle)
- OCP (Open-Closed Principle)
- LSP (Liskov Substitution Principle)
- ISP (Interface Seggregation Principle)
- DIP (Dependency Injection Principle)

Robert C. Martin

Source: Wikipedia

(S)ingle Responsibility

- A module must have one responsibility
  - Responsibility = A reason to change
  - No more than one reason to change a module
    - Otherwise, responsibilities are mixed and coupling increases

VS

(O)pen/Closed principle

- Open for extension
  - The module must adapt to new changes
  - Change/adapt the behavior of a module
- Closed for modification
  - Changes can be done without changing the module
    - Without modifying source code, binaries, etc

> It should be easy to change the behaviour of a module without changing the source code of that module

http://blog.8thlight.com/uncle-bob/2013/03/08/AnOpenAndClosedCase.html

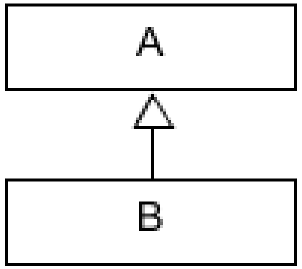
(L)iskov Substitution

Subtypes must follow supertypes contract

B is a subtype of A when:

∀x∈A, if there is a property Q such that Q(x)

then ∀y∈B, Q(y)



"Derived types must be completely substitutable by their base types"

Common mistakes:

Inherit and modify behaviour of base class

Add functionality to supertypes that subtypes don't follow

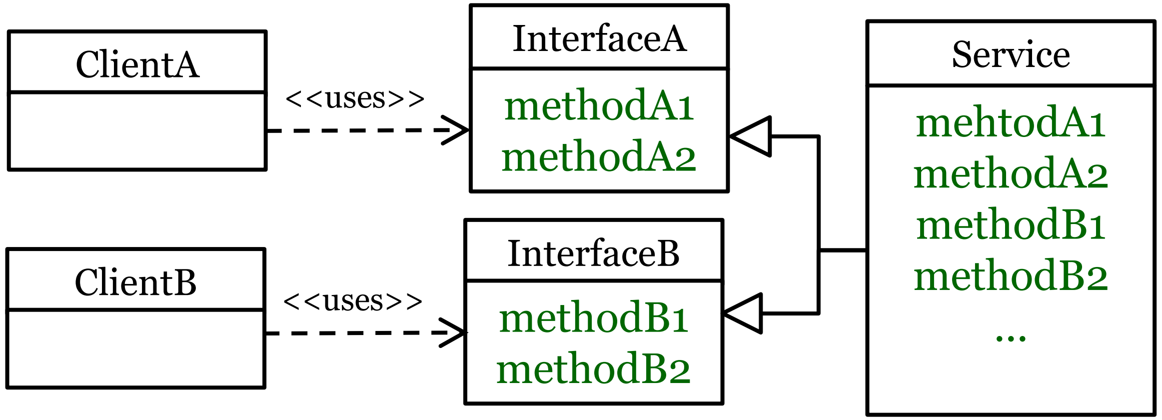
(I)nterface Segregation

Clients must not depend on unused methods

Better to have small and cohesive interfaces

Otherwise $\Rightarrow$ non desired dependencies

- If a module depends on non-used functionalities and these functionalities change, it can be effected



(D)ependency Inversion

- Invert conventional dependencies
  - *High-level modules should not depend on low-level modules*
    - *Both should depend on abstractions*
  - *Abstractions should not depend upon details.*
    - *Details should depend upon abstractions*
- Can be accomplished using dependency injection or several patterns like plugin, service locator, etc.

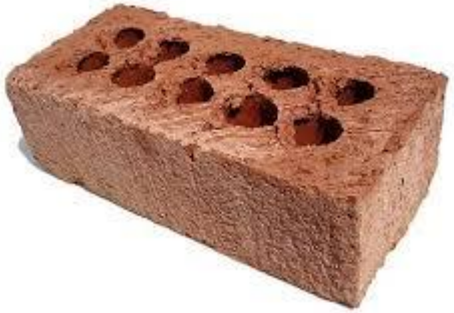
http://www.objectmentor.com/resources/articles/dip.pdf
http://martinfowler.com/articles/dipInTheWild.html

(D)ependency Inversion

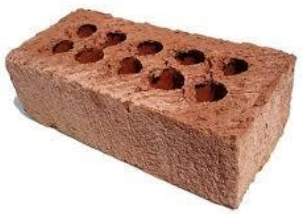
- Lowers coupling
- Facilitates unit testing
  - Substituting low level modules by test doubles
- Related with:
  - Dependency injection and Inversion of Control
    - Frameworks: Spring, Guice, etc.

Cohesion

- Cohesion = Degree to which the elements of a module work together
- It is recommended to have high cohesion
  - Each module must solve one functionality
  - Granularity
    - Modules must be released and reused independently
  - It should be possible to test each module separately

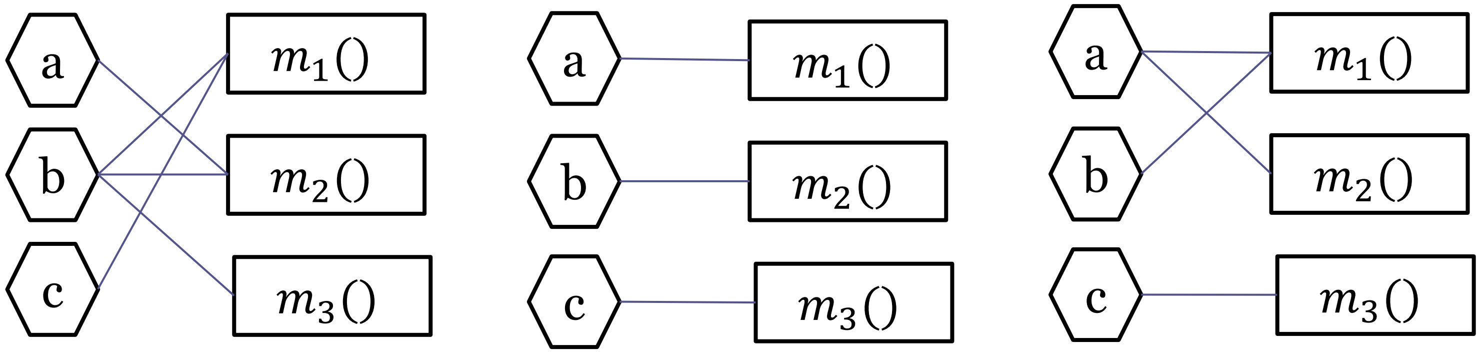


Cohesion metric LCOM



- LCOM (Lack of cohesion of methods), Chidamber and Kemerer
  - Measure degree of similarity of methods in a class
  - Several variants have been proposed LCOM 1-5
  - $$\text{LCOM} = \begin{cases} |P| - |Q| & si \quad |P| - |Q| > 0 \\ 0 & en\ caso\ contrario \end{cases}$$
    - $|P|$= Number of methods without common attributes
    - $|Q|$= Number of methods with common attributes

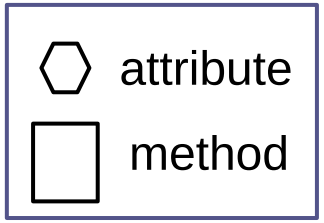
Bigger LCOM ⇒ less cohesion



$|P|$= 0, $|Q|$= 3
LCOM=0

$|P|$= 3, $|Q|$= 0
LCOM=3

$|P|$= 2, $|Q|$= 1
LCOM=1



Cohesion principles

REP - Reuse/Release Equivalence Principle

CCP - Common Closure Principle

CRP - Common Reuse Principle

Robert C. Martin

Source: Wikipedia

REP
Reuse/Release Equivalence Principle

- The granule of reuse is the granule of release
  - In order to reuse an element in practice, it is necessary to publish it in a release system of some kind
    - Release version management: numbers/names
- All related entities must be released together
  - Group entities for reuse

CCP
Common Closure Principle

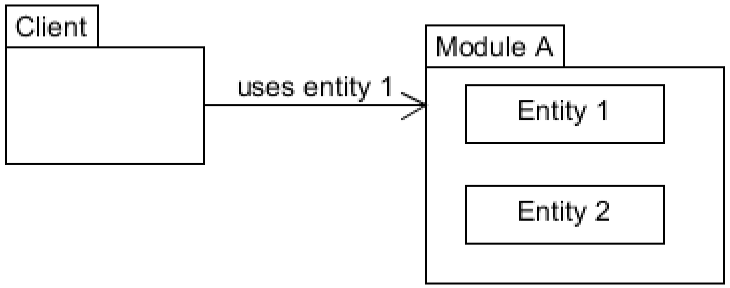
- Gather in a module entities that change for the same reasons and at the same time
  - Entities that change together belong together
    - Goal: limit the dispersion of changes among release modules
      - Changes must affect the smallest number of released modules
  - Entities within a module must be cohesive
    - Group entities for maintenance

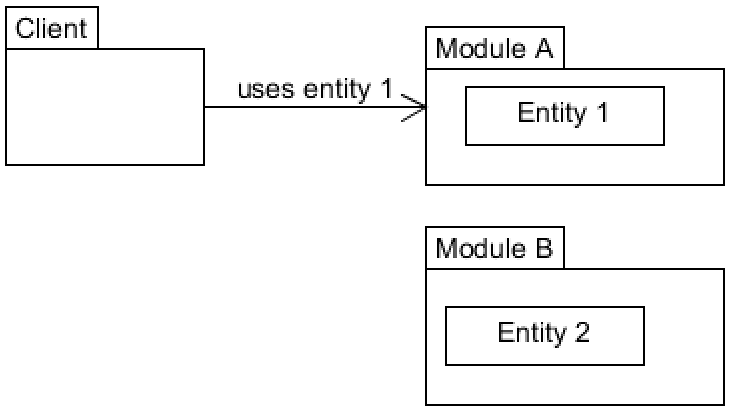
Note: imilar to SRP (Single Responsibility Principle), but for modules

CRP
Common Reuse Principle

Modules should only depend on entities they need

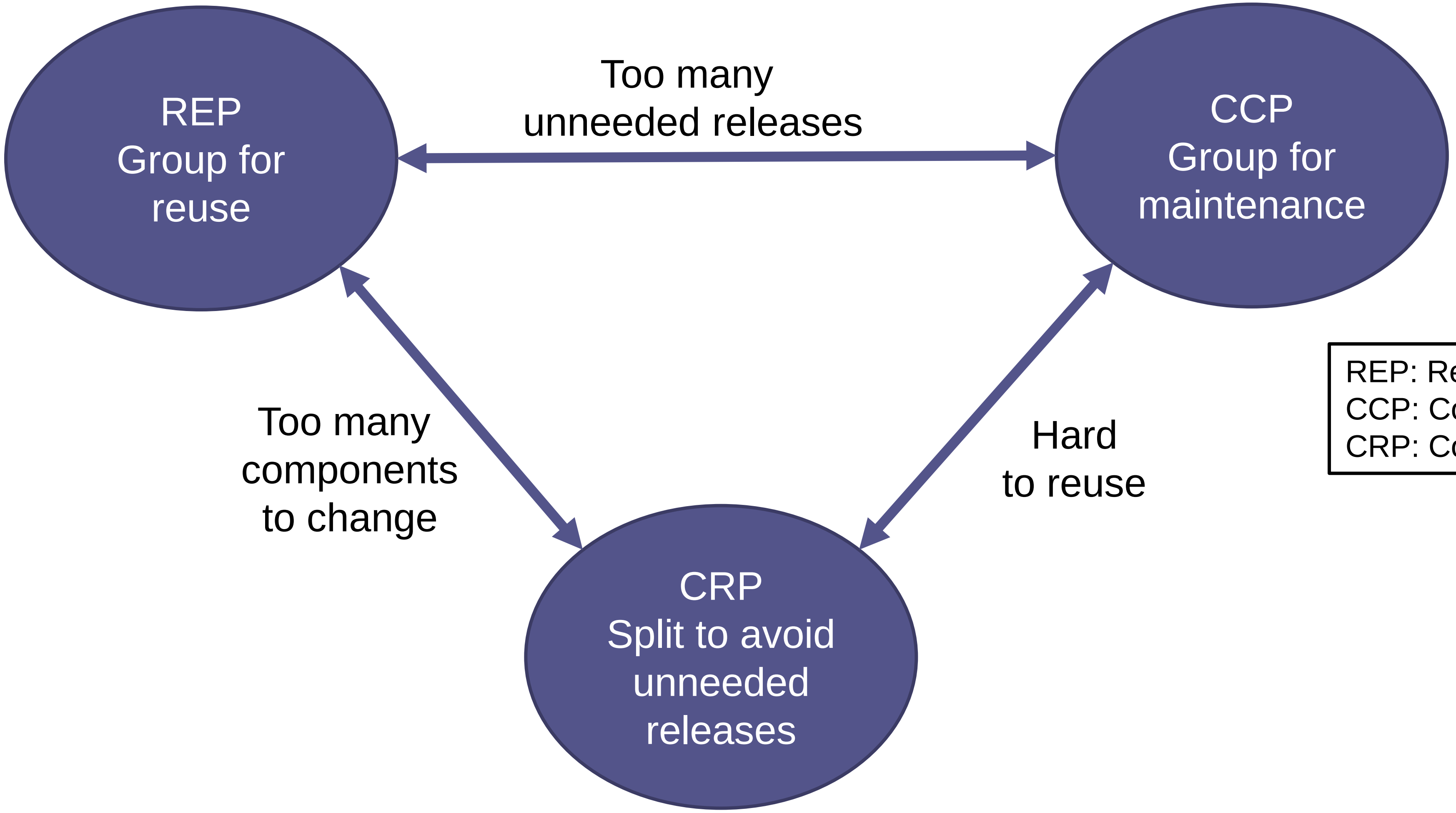
- *They shouldn't depend on things they don't need*
- Otherwise, a consumer may be affected by changes on entities that is not using
- Split entities in modules to avoid unneeded releases





Note: This principle is related with the ISP (Interface Seggregation Principle)

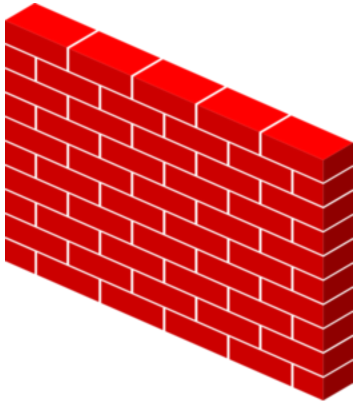
Tension diagram between component cohesion



REP: Reuse/Release Equivalence Principle
CCP: Common Closure Principle
CRP: Common Reuse Principle

Coupling

- Coupling = Degree of interdependence between software modules
  - Low coupling ⇒ Improves modifiability
- Independent deployment of each module
  - Stability against changes in other modules



Coupling principles

ADP - Acyclic dependencies principle

SDP - Stable dependencies principle

SAP - Stable abstractions principle

Robert C. Martin

Source: Wikipedia

ADP - Acyclic Dependencies Principle

- The dependency structure for released modules must be a Directed Acyclic Graph (DAG)
  - Avoid cycles
  - A cycle can make a single change very difficult
    - Lots of modules are affected
    - Problem to work-out the building order

NOTE: Cycles can be avoided using the DIP (Dependency Inversion Principle)

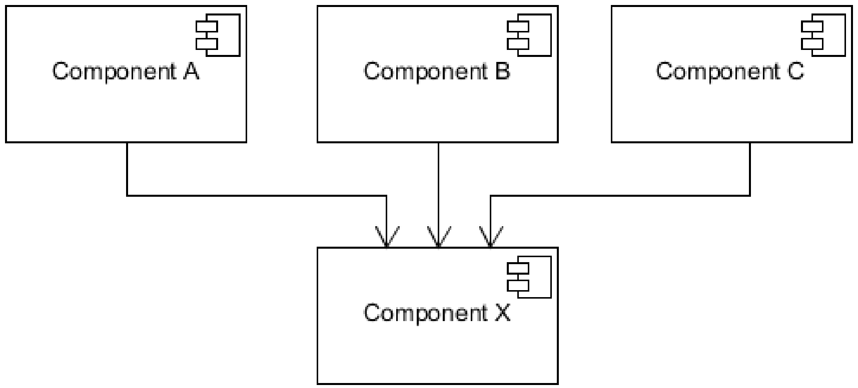
http://wiki.c2.com/?AcyclicDependenciesPrinciple

SDP Stable Dependencies Principle

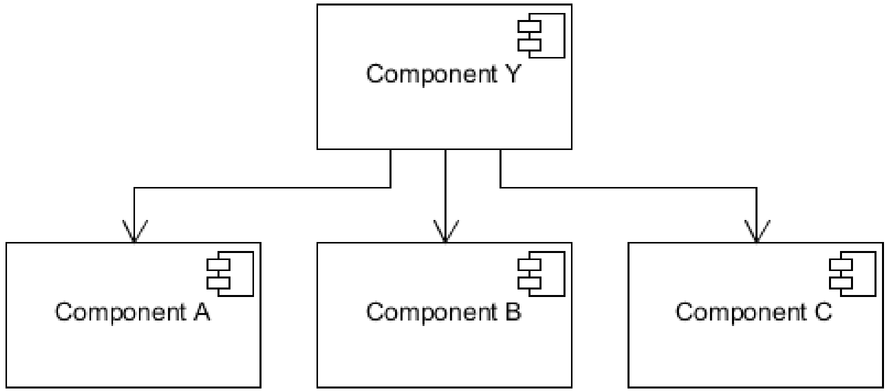
The dependencies between components in a design should be in the direction of stability

A component should only depend upon components that are more stable than it is

Stability = fewer reasons to change



Component X is stable
Only depends on itself



Component Y is less stable
It has at least 3 reasons to change

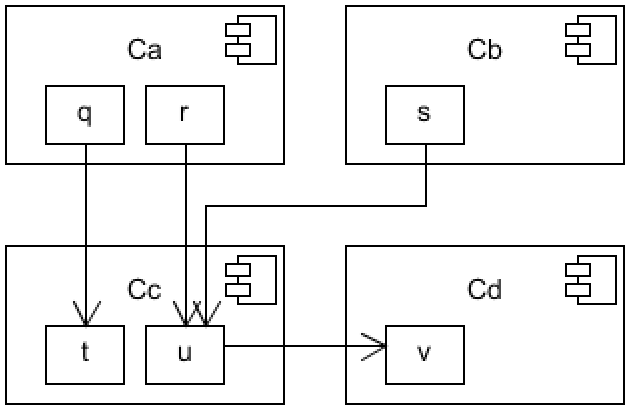
Stability metrics

Fan-in: incoming dependencies

Fan-out: outgoing dependencies

Instability $I = \frac{Fan{-}out}{Fan{-}in + Fan{-}out}$

Value between 0 (stable) and 1 (instable)



$I(Ca)=\frac{2}{0+2} = 1$

$I(Cb)=\frac{1}{0+1} = 1$

$I(Cc)=\frac{1}{3+1} = \frac{1}{4}$

$I(Cd)=\frac{0}{1+0} = 0$

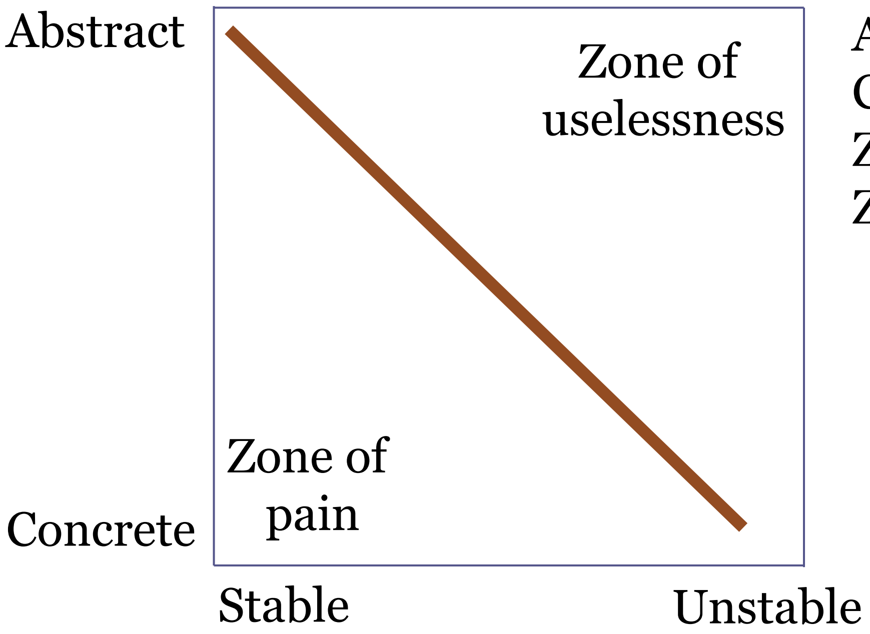
Stable Dependencies Principle states that the dependencies should be from higher instability to lower

http://wiki.c2.com/?StableDependenciesPrinciple

SAP - Stable Abstractions Principle

A module should be as abstract as it is stable

- Packages that are maximally stable should be maximally abstract.
- Instable packages should be concrete



Abstract/stable = Interfaces with lots of dependant modules
Concrete/Unstable = Implementations without dependant modules
Zone of pain = DB schema
Zone of uselessness = interfaces without implementation

http://wiki.c2.com/?StableAbstractionsPrinciple

Connascence

- Things that are born and grow together
- A change in one requires others to be modified to maintain the system correct
  - Indicates problems to change - affects modifiability
  - A vocabulary to talk about coupling
  - Combines coupling and cohesion
- Several types of connascence
  - Static = can be detected with static analysis
  - Dynamic = detected at runtime

More info: https://connascence.io/

Static connascence

Of name

Several components must agree on the same name

Of type

Several components must agree on the same type

Of meaning

Several components must agree on a meaning

Example: magical constants

Of position

Several components must agree on a position

Example: arguments with same type

Of algorithm

Several components must agree on an algorithm

Example: Same hash function to encrypt/decrypt

```
public class Time {
 int _hour; int _min; int _sec;

 public Time(int hour, int min, int sec) {
  _hour = hour ;
  _minute = minute ;
  _second = second ;
 }

 public String display() {
  return _hour + ":" + _min + ":" + _sec ;
 }
}

public class Client {
 val noon = Time(12,0,0);
 . . .
}
```

Dynamic connascence

- Of execution
  - The order of execution is important
- Of timing
  - When the timing is important
  - Example: race conditions
- Of values
  - Several values must change together
- Of identity
  - Multiple components must reference the same entity

```
Email email = new Email();
email.setRecipient("foo@example.comp");
email.setSender("me@mydomain.com");
email.send();
email.setSubject("Hello World");
```

3 properties of connascence

- Degree
  - Number of elements affected by connascence
- Locality
  - Distance between those elements
    - Same function?, same class?, same package? ...
- Strength
  - Easy with which it can refactored

Types of connascence

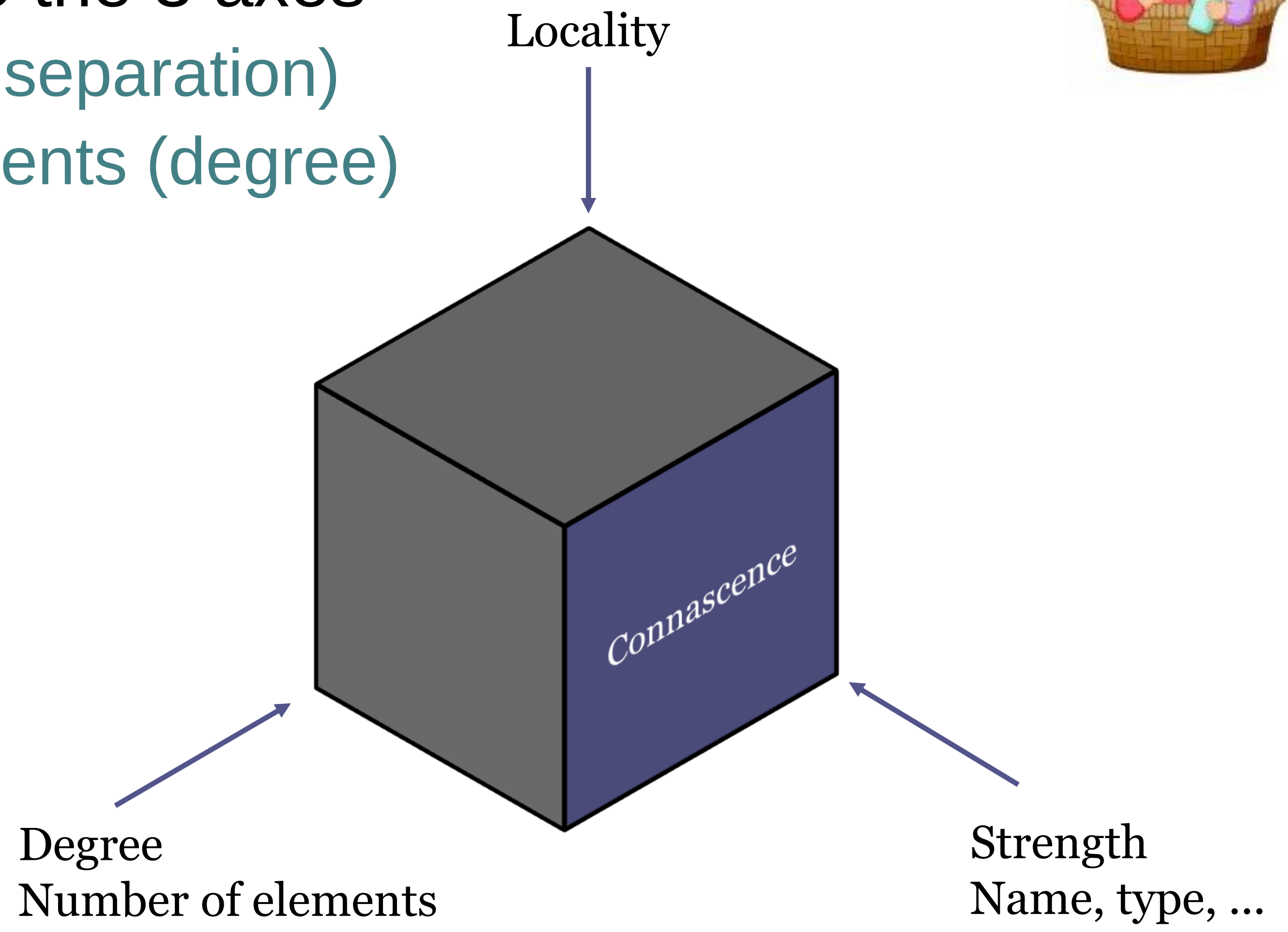
Strength: - (top) → + (bottom)

Static Can be detected with static analysis	Of name
	Of Type
	Of meaning
	Of position
	Of algorithm
Dynamic Detected at runtime	Of execution
	Of timing
	Of value
	Of identity

Reducing connascence

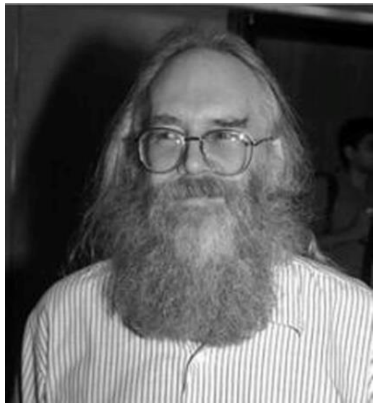
Refactor code according to the 3 axes

- Minimize locality (reduce separation)
- Minimize number of elements (degree)
- Minimize strength



Robustness Principle, Postel's law

- Postel's law (1980), defined for TCP/IP
  - Be liberal in what you accept and conservative in what you send
- Improve interoperability
  - Send well formed messages
  - Accept incorrect messages
- Applications to API design
  - Process fields of interest ignoring the rest
  - Allows APIs to evolve later



Jon Postel

https://en.wikipedia.org/wiki/Robustness_principle
https://devopedia.org/postel-s-law

Demeter's Law

Also known as Principle of less knowledge

Named after the Demeter System (1988)

- Units should have limited knowledge about other units
  - Only units “closely” related to the current unit.
- Each unit should only talk to its friends
  - "Don’t talk to strangers"
- Symptoms of bad design
  - Using more than one dot...

```
a.b.method(...)
a.method(...)
```

The Law of Demeter improves loosely coupled modules
It is not always possible to follow

Fluent APIs

Improve readability and usability of interfaces

Advantages

- Can lead to domain specific languages
- Auto-complete facilities by IDEs

```
Product p = new Product().setName("Pepe").setPrice(23);
```

Trick: Methods that modify, return the same object

```
class Product {
 ...
  public Product setPrice(double price) {
   this.price = price;
   return this;
  };
```

It does not contradict Demeter's Law because it acts on the same object

Other modularity recommendations

- Facilitate external configuration of a module
  - Create an external configuration module
- Create a default implementation
- GRASP Principles
  - General Responsibility Assignment Software Patterns
- DRY (Don't repeat yourself)
  - Intent is declared in one place
- YAGNI (You ain't gonna need it) and
- KISS (Keep it simple stupid)
  - Do the Simplest Thing That Could Possibly Work”

Modularity styles

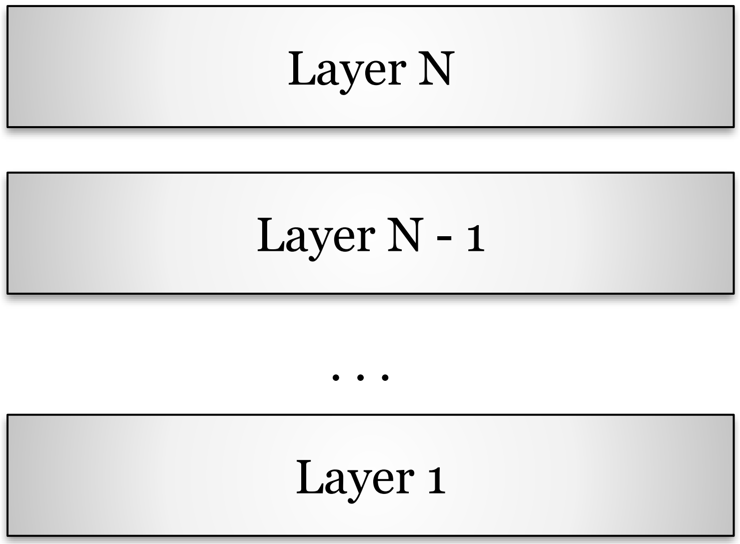
Layers

Layers

Divide software modules in layers

Layers are ordered

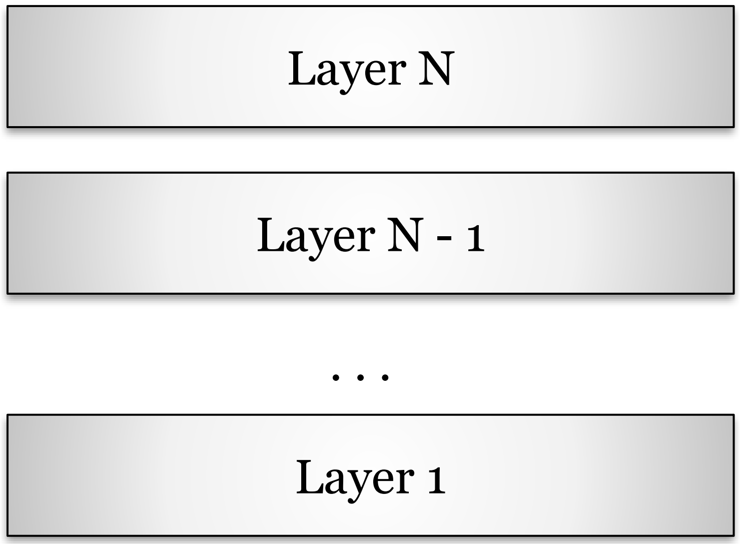
Each layer exposes an interface that can be used by higher layers



Layers

Elements

- Layer: set of functionalities exposed through an interface at a level N
- Order relationship between layers



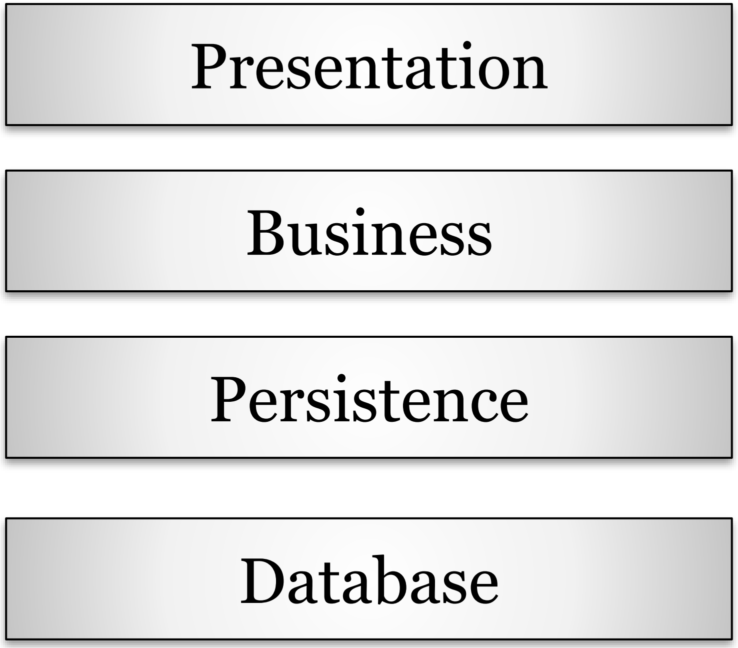
Layers

Constraints

- Each software block belongs to one layer
- There are at least 2 layers
- A layer can be:
  - Client: consumes services from below layers
  - Server: provides services to upper layers
- 2 variants:
  - Strict: Layer N uses only functionality from layer N-1
  - Lax: Layer N uses functionalities from layers N - 1 a 1
- No cycles

Layers

Example



Layers

Layers ≠ Modules

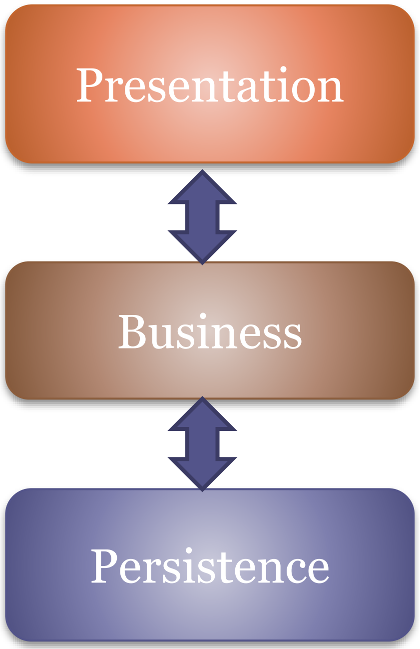
- A layer can be a module...
- ...but modules can be decomposed in other modules (layers can't)
- Dividing a layer, it is possible to obtain modules

University of Oviedo

Layers

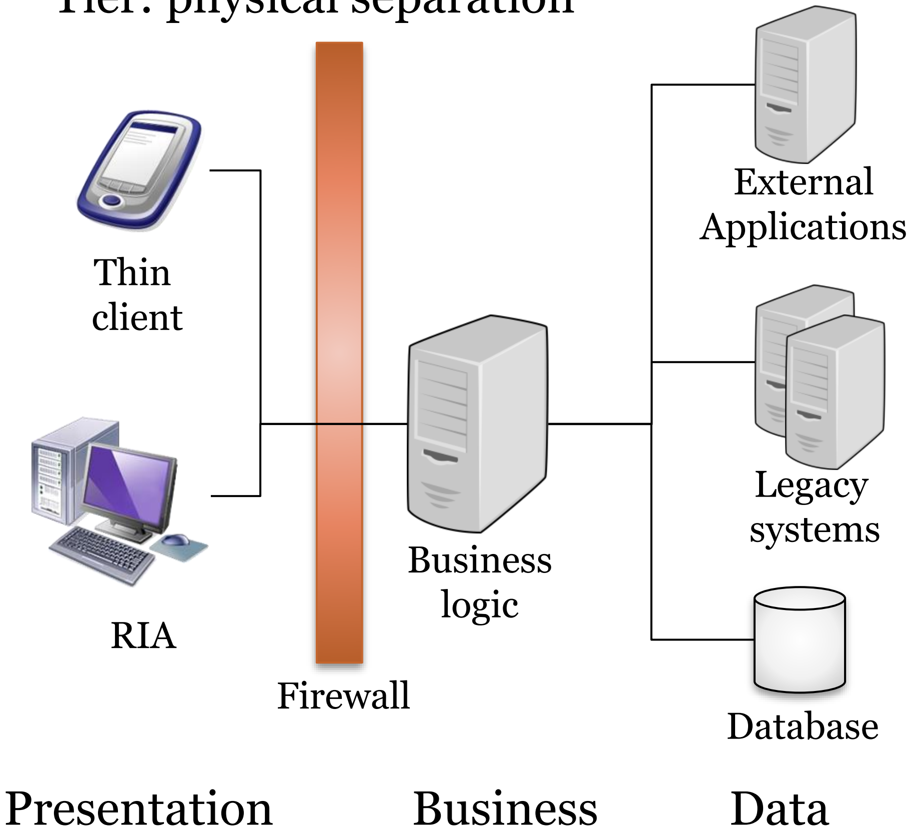
Layers ≠ Tiers

Layer: conceptual separation



3-Layers

Tier: physical separation



3-tiers

School of Computer Science

Layers

Advantages

- Separates different abstraction levels
- Loose coupling: independent evolution of each layer
  - It is possible to offer different implementations of a layer that keep the same interface
- Reusability
  - Changes in a layer affects only to the layer that is above or below it.
  - It is possible to create standard interfaces as libraries or application frameworks
- Testability

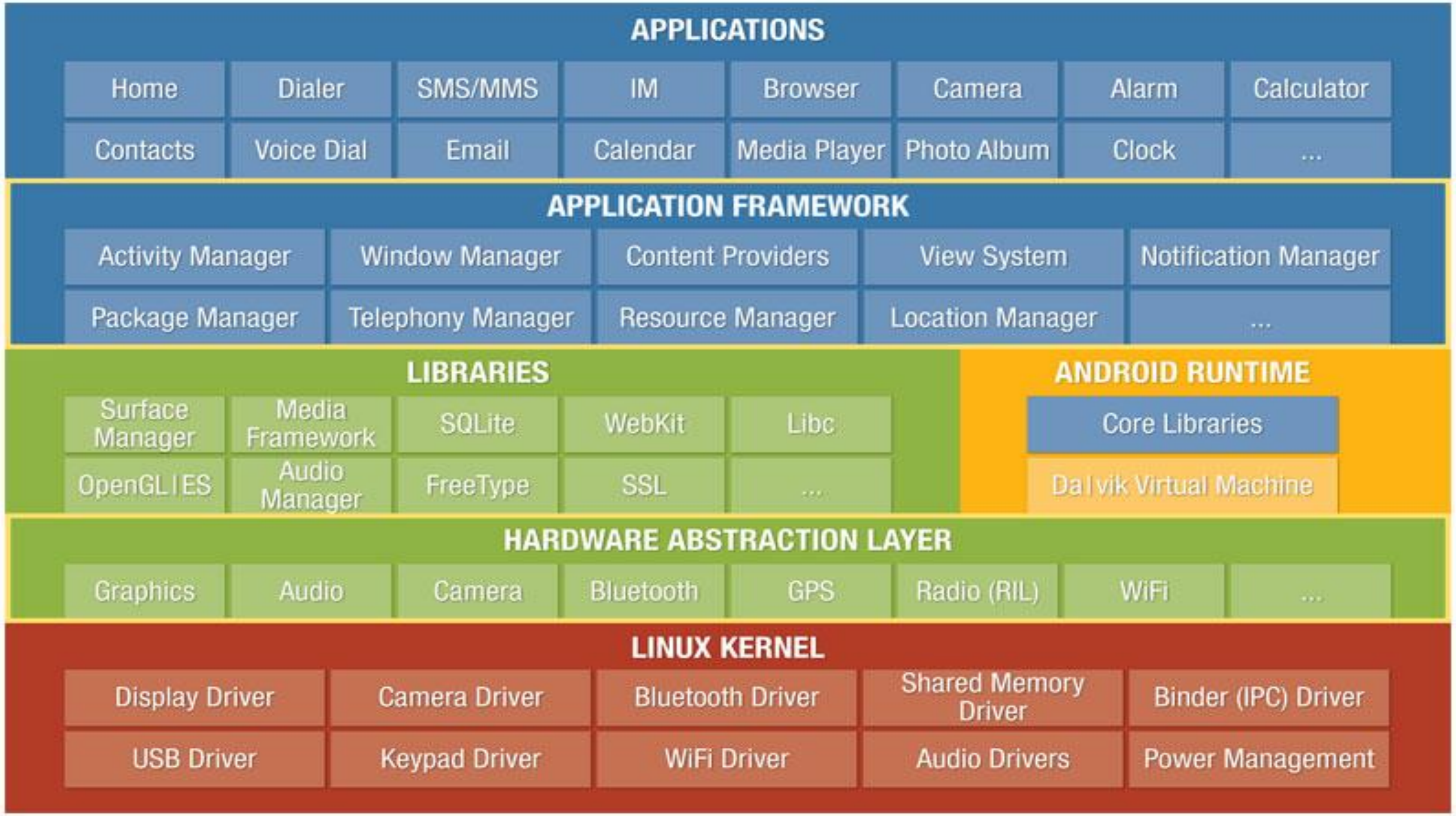
Layers

Challenges

- It is not always possible to apply it
  - We don't always have different abstraction levels
- Performance
  - Access through layers can slow the system
- Shortcuts
  - Sometimes, it may be necessary to skip some layers
- It can lend to monolithic applications
  - Issues in terms of deployment, reliability, scalability
- Sinkhole antipattern
  - Requests flow through layers without processing

Layers

Example: Android



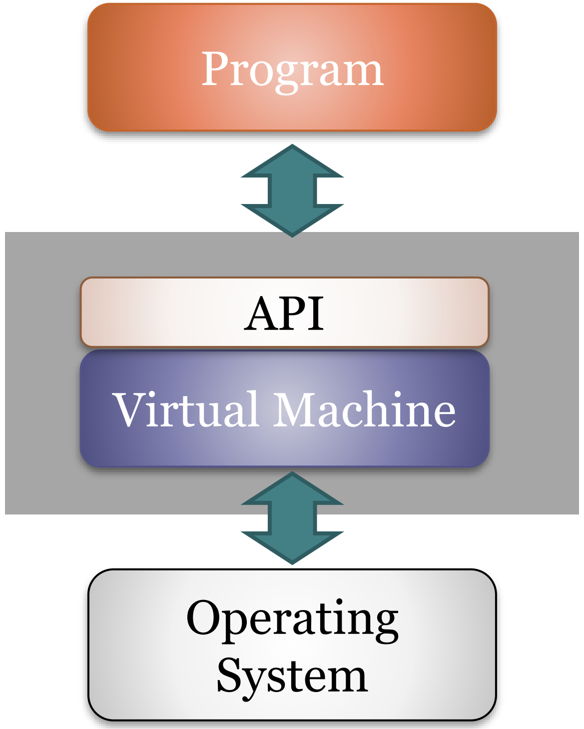
Layers

Variants:

- Virtual machines, APIs
- 3-layers, N-layers

Virtual machines

- Virtual machine = Opaque layer
  - Hides a specific OS implementation
- One can only get Access through the public API



Virtual machines

- Advantages
  - Portability
  - Simplifies software development
    - Higher-level programming
  - Facilitates emulation
- Challenges
  - Performance
    - JIT techniques
  - Computational overload generated by the new layer

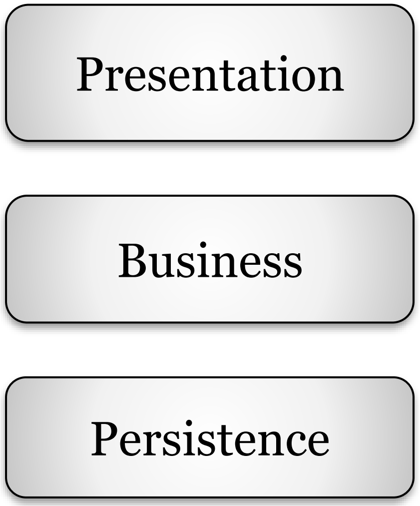
Virtual machines

- Applications
  - Programming languages
    - JVM: Java Virtual Machine
    - CLR .Net
  - Emulation software

3-layers (N-layers)

Technical partitioning

Each layer requires different technical capabilities

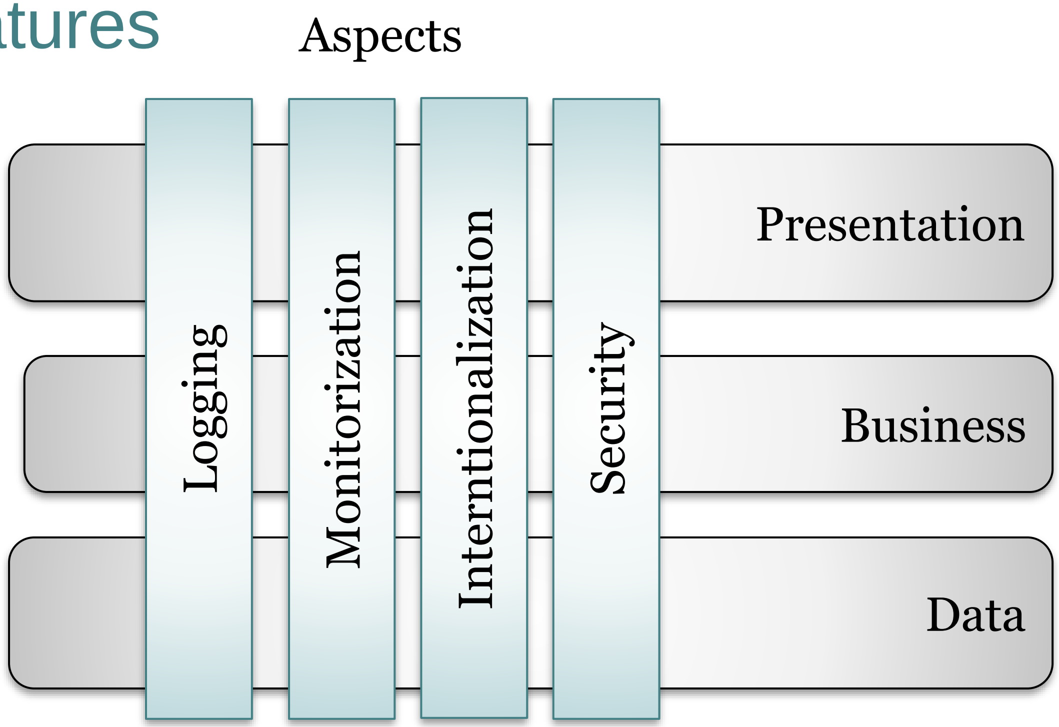


Aspect Oriented

Aspect Oriented

Aspects:

- Modules that implement crosscutting features



Aspect Oriented

Elements:

- *Crosscutting concern*
  - Functionality that is required in several places of an application
  - Examples: logging, monitoring, i18n, security,...
  - Generate *tangling* code
- Aspect. Captures a *crosscutting-concern* in a module

Aspect Oriented

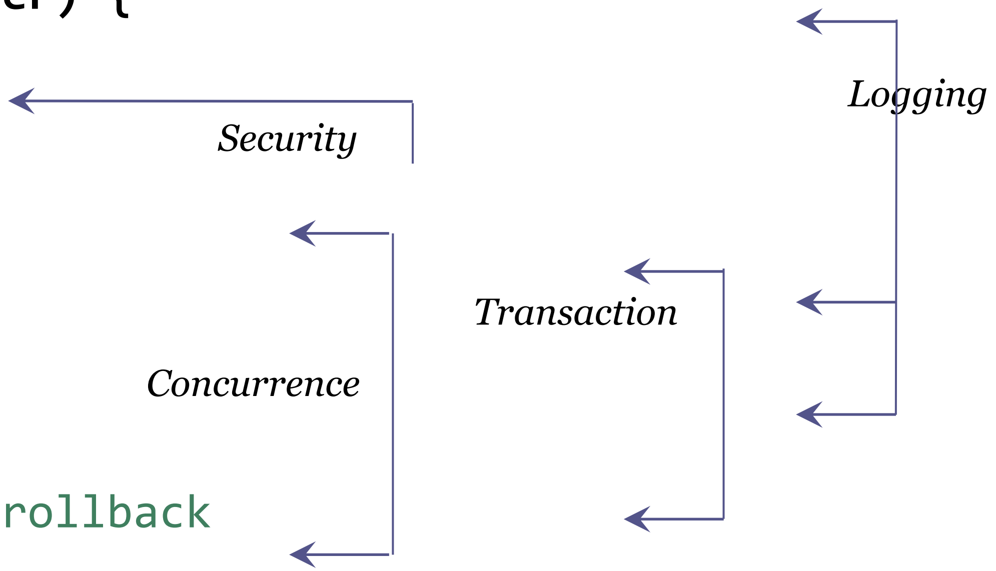
Example: Book flight seats

- Several methods to do the booking:
  - Book a seat
  - Book a row
  - Book two consecutive seats
  - ...
- En each method:
  - Check permission (security)
  - Concurrence (block seats)
  - Transactions (do the whole operation in one step)
  - Create a log of the operation
  - ...

Aspect Oriented

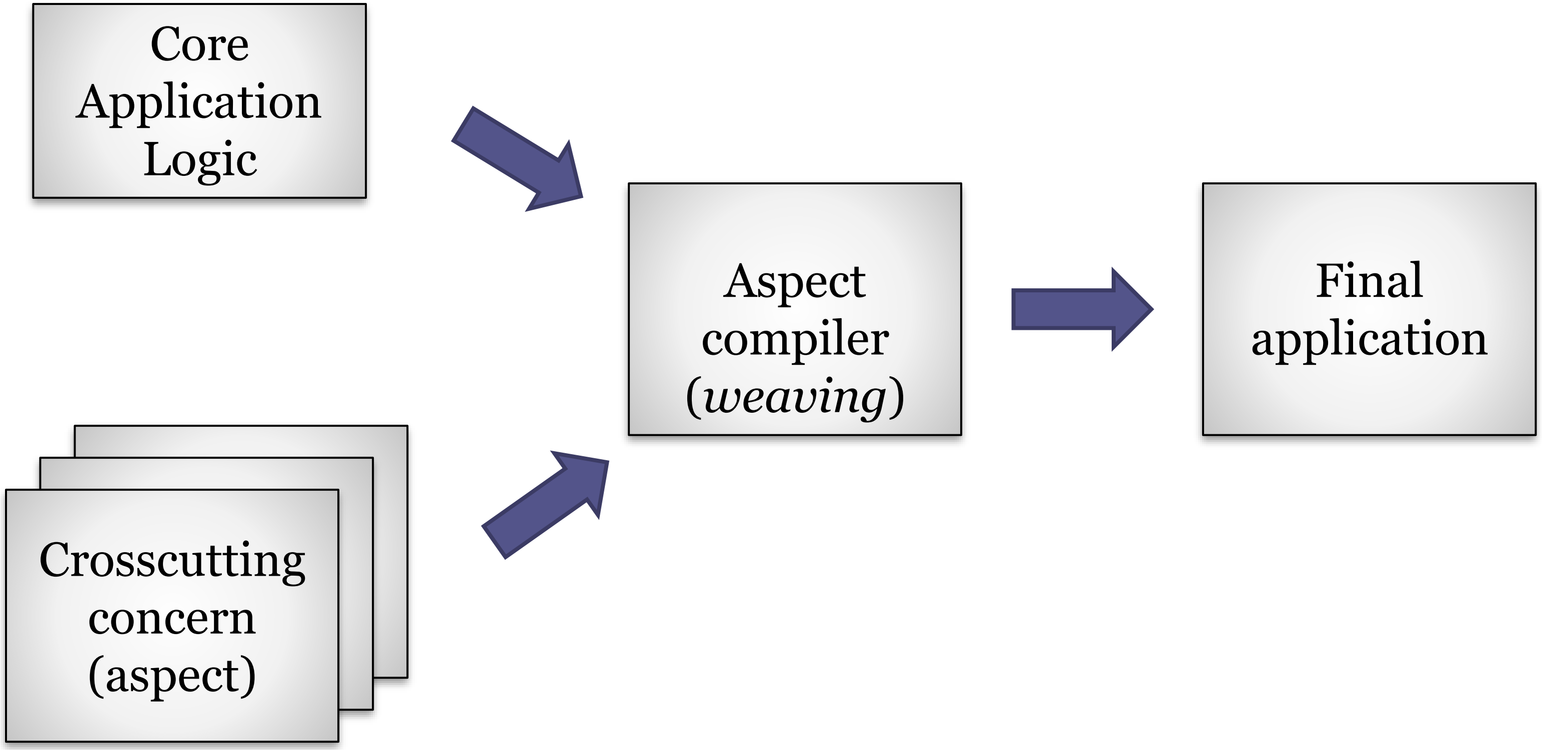
Traditional solution

```
class Plane {
 void bookSeat(int row, int number) {
  // ... Log book petition
  // ... check authorization
  // ... check free seat
  // ... block seat
  // ... start transition
  // ... log start of operation
  // ... Do booking
  // ... Log end of operation
  // ... Execute transaction or rollback
  // ... Unblock
 }
 ...
 public void bookRow(int row) {
 // ... More or less the same!!!!
 ...
```



Aspect Oriented

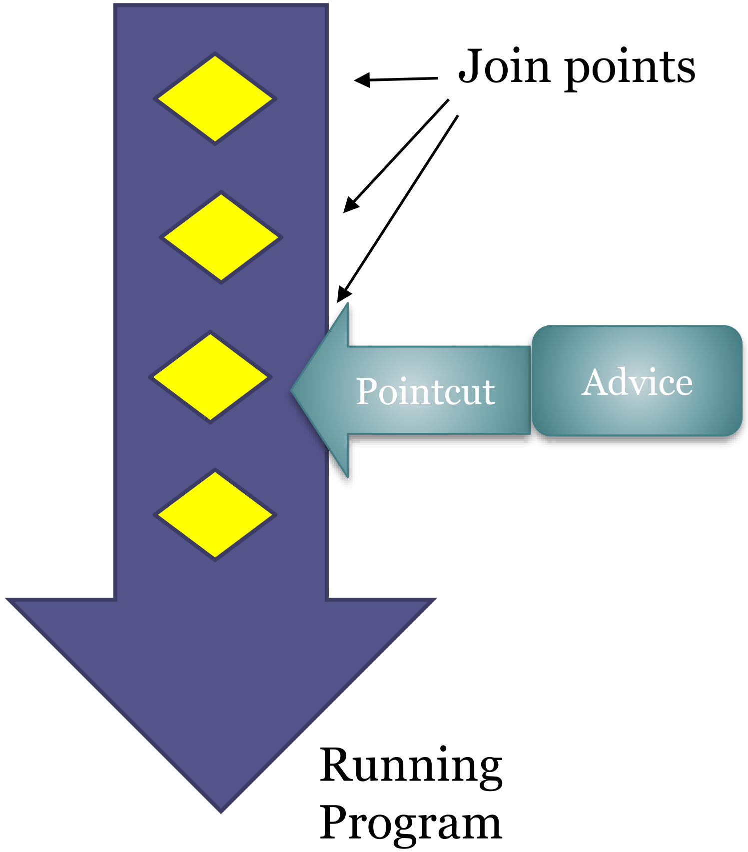
Structure



Aspect Oriented

Definitions

- *Join point:* Point where an aspect can be inserted
- Aspect:
  - Contains:
    - *Advice*: defines the job of the aspect
    - *Pointcut*: where the aspect will be introduced
      - It can match one or more *join points*



Aspect Oriented

Aspect example in @Aspectj

```
@Aspect
public class Security {

 @Pointcut("execution(* org.example.Flight.book*(..))")
 public void safeAccess() {}

 @Before("safeAccess()")
  public void authenticate(JoinPoint joinPoint) {
   // Does the authentication
 }

}
```

Methods book*

It is executed before to invoke those methods

It can Access to information of the joinPoint

Aspect Oriented

Constraints:

- An aspect can affect one or more traditional modules
- An aspect captures all the definitions of a *crosscutting-concern*
- The aspect must be inserted in the code
  - Tools for automatic introduction

Aspect Oriented

Advantages

- Simpler design
  - Basic application is clean of crosscutting concerns
- Facilitates system modifiability and maintenance
  - Crosscutting concerns are localized in a single module
- Reuse
  - *Crosscutting concerns* can be reused in other systems

Aspect Oriented

Challenges

- External tools are needed
  - Aspects compiler. Example: AspectJ
  - Other tools: Spring, JBoss
- Debugging is more complex
  - A bug in one aspect module can have unknown consequences in other modules
  - Program flow is more complex
- Team development needs new skills
  - Not every developer knows aspect oriented programming

Aspect Oriented

- Applications
  - AspectJ = Java extension with AOP
  - Guice = Dependency injection Framework
  - Spring = Enterprise framework with dependency injection and AOP
- Variants
  - DCI (Data-Context-Interaction): It is centered in the identification of roles from use cases
    - Apache Polygene

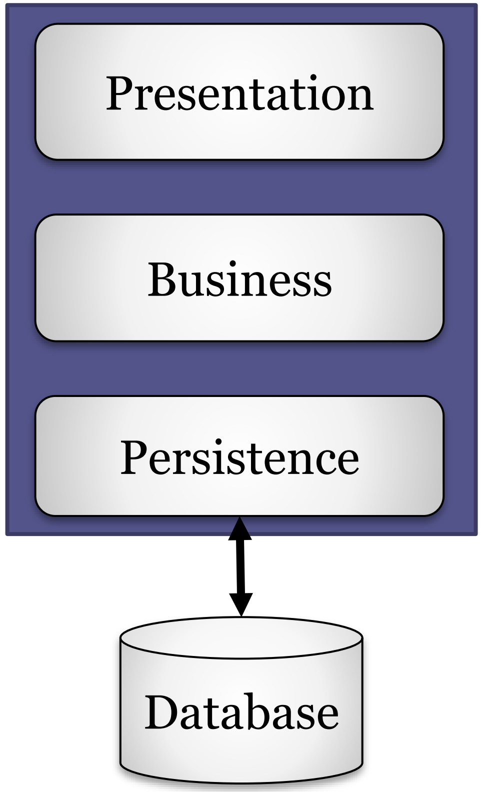
Domain based

- Domain driven design
- Hexagonal architecture
- Data centered
- Naked Objects

Technical vs domain partitioning

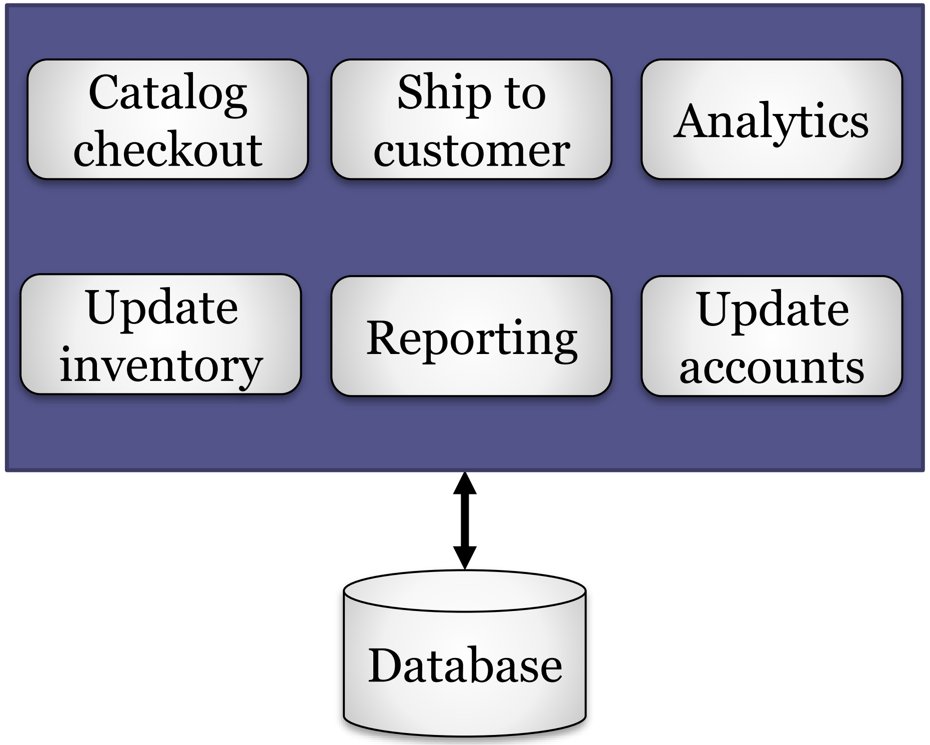
Technical partitioning

Organize system modules by technical capabilities



Domain partitioning

Organize modules by domain



Data model vs domain model

Data models

- Physical:
  - Data representation
    - Tables, columns, keys, ...
- Logical:
  - Data structure
    - Entities and relationships

Domain models

- Conceptual model of some domain
- Vocabulary and context
  - Entities, relationships
- Behavior
  - Business rules

Domain based

- Centered on the domain and the business logic
  - Goal: Anticipate and handle changes in domain
  - Collaboration between developers and domain experts

Domain based

- Elements
  - Domain model: formed by:
    - Context
    - Entities
    - Relationships
  - Application
    - Manipulates domain elements

Domain based

Constraints

- Domain model is a clearly identified module separated from other modules
- Domain centered application
  - Application must adapt to domain model changes
- No topological constraints

Domain based

Advantages:

- Facilitates team communication
  - Ubiquitous language
- Reflects domain structure
  - Address domain changes
  - Share and reuse models
- Reinforce data quality and consistency
- Facilitates system testing
  - It is possible to create testing doubles with fake domain data

Domain based

Challenges:

- Collaboration with domain experts
- Stalled analysis phase
  - It is necessary to establish context boundaries
- Technological dependency
  - Avoid domain models that depend on some specific persistence technologies
- Synchronization
  - Synchronize system with domain changes

Domain based

Variants

- DDD - *Domain driven design*
- Hexagonal style
- Data centered
- N-Layered Domain Driven Design
- Naked Objects

DDD - Domain Driven Design

- General approach to software development
- Proposed by Eric Evans (2004)
- Connect the implementation to an evolving domain
- Collaboration between technical and domain experts
- Ubiquitous language
  - Common vocabulary shared by the experts and the development team

http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design

DDD - Domain Driven Design

Elements

- Bounded context
  - Specifies the boundaries of the domain
- Entities
  - An object with an identity
- Value objects
  - Contain attributes but no identity
- Aggregates
  - Collection of objects bound together by some root entity
- Repositories
  - Storage service
- Factories
  - Creates objects
- Services
  - External operations

DDD - Domain Driven Design

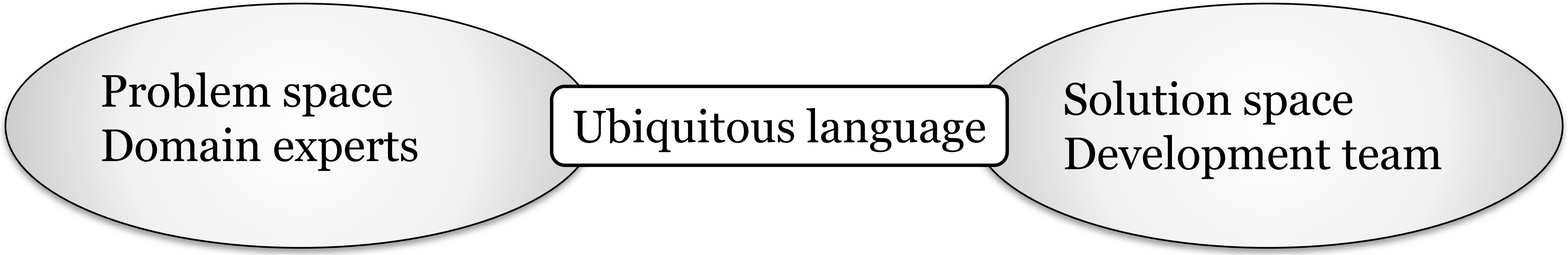
Constraints

- Entities inside aggregates are only accessible through the root entity
- Repositories handle storage
- Value objects are immutable
  - Usually contain only attributes

DDD - Domain driven design

Advantages

- Code organization
  - Identification of the main parts
- Maintenance/evolution of the system
  - Facilitates refactoring
  - It adapts to Behavior Driven Development
- Team communication



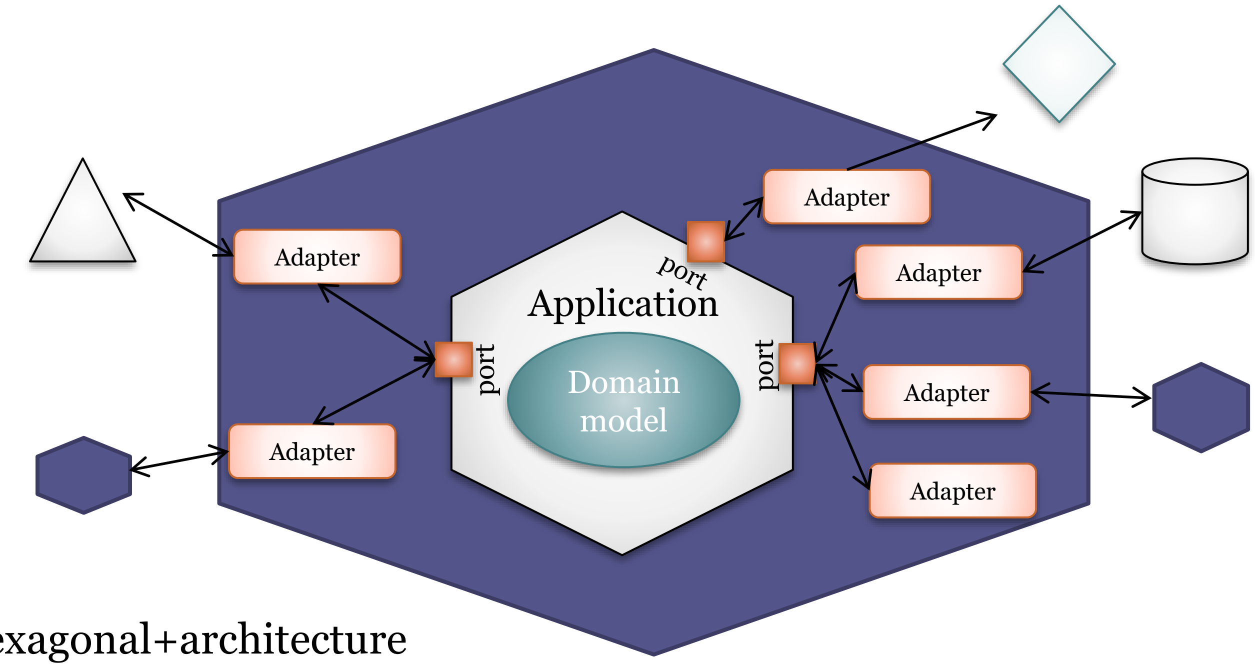
DDD - Domain driven design

Challenges

- Involve domain experts in development
  - It is not always possible
- Apparent complexity
  - It adds some constraints to development
  - Useful for complex, non-trivial domains

Hexagonal style

- Other names:
  - ports and adapters, onion, clean architecture, etc.
- Based on a clean Domain model
  - Infrastructures and frameworks are outside
  - Access through ports and adapters

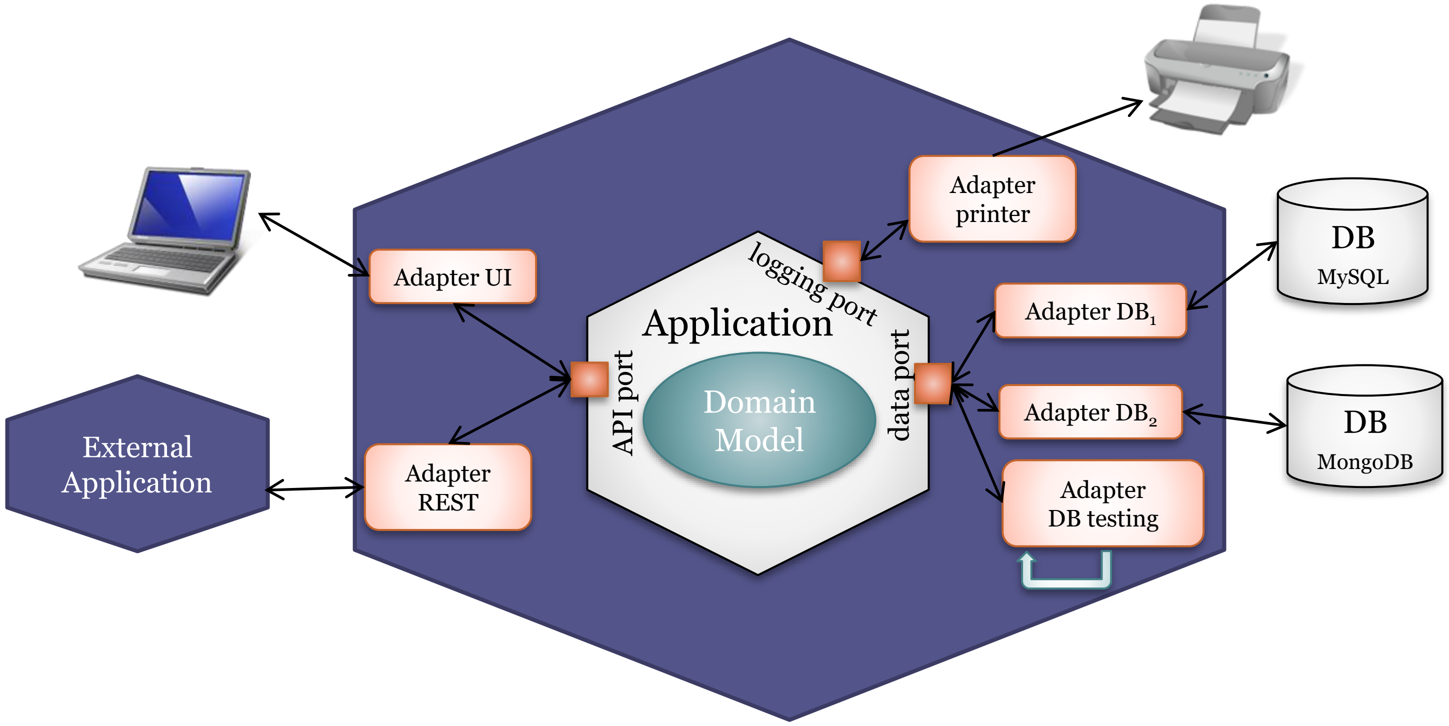


http://alistair.cockburn.us/Hexagonal+architecture
http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

Hexagonal style

Example

- Traditional application in layers
  - Incorporates new services
  - Testing DB



Hexagonal style

- Elements
  - Domain model
    - Represents business logic: Entities and relationships
    - Plain Objects (POJOs: Plain Old Java Objects)
  - Ports
    - Communication interface
    - It can be: User, Database
  - Adapters
    - One adapter by each external element
    - Examples: REST, User, DB SQL, DB mock,...

Hexagonal style

Advantages

- Understanding
  - Improves domain understanding
- Timelessness
  - Less dependency on technologies and frameworks
- Adaptability (*time to market*)
  - It is easier to adapt the application to changes in the domain
- Testability
  - It is possible to substitute real databases by mock databases

Hexagonal style

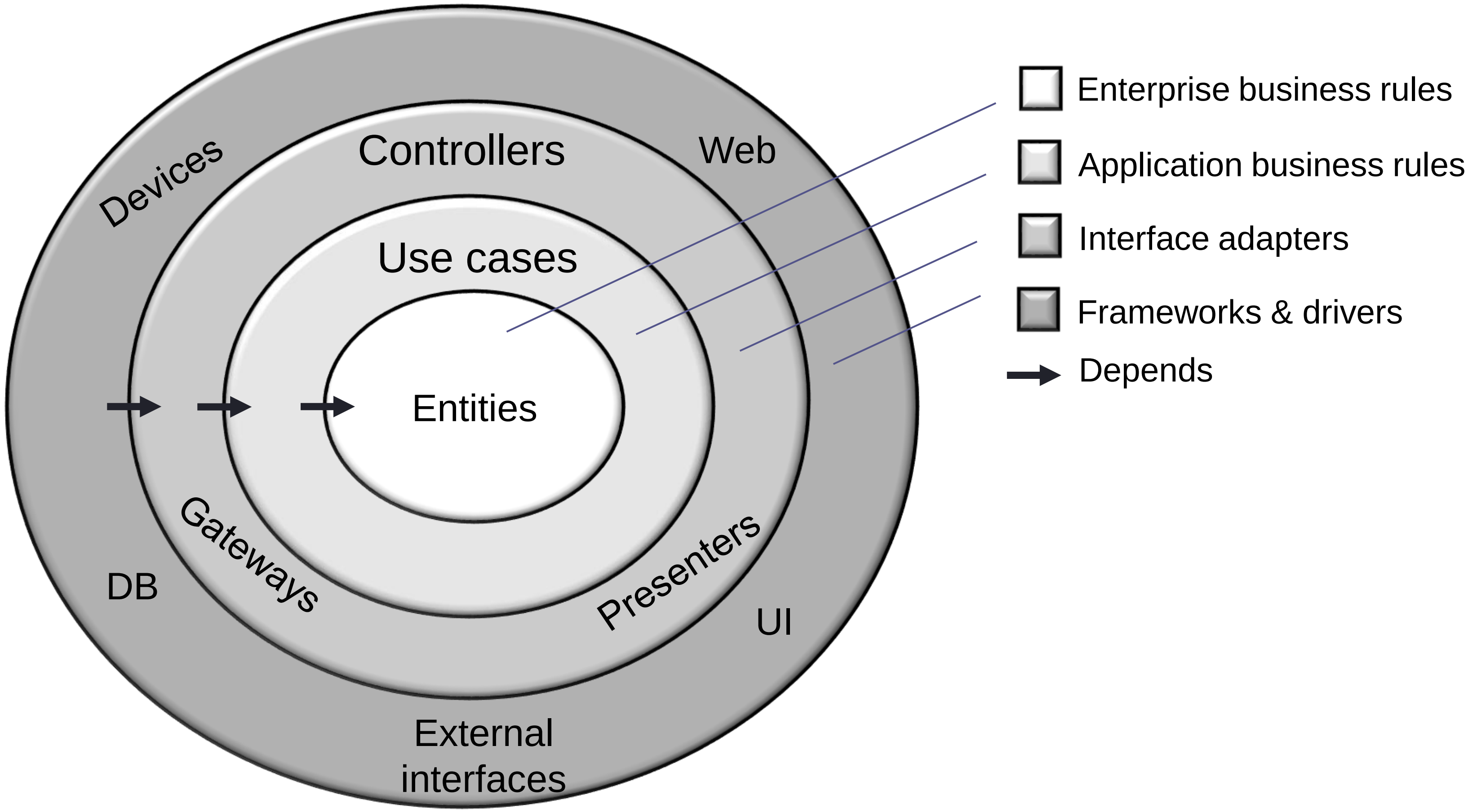
Challenges

- It can be difficult to separate domain from the persistence system
  - Lots of frameworks combine both
- Asymmetry of ports & adapters
  - Not all are equal
  - Active ports (user) vs passive ports (logger)

Clean architecture

Robert C. Martin

- Similar to hexagonal architecture
  - Presented by Uncle Bob - Clean architecture book



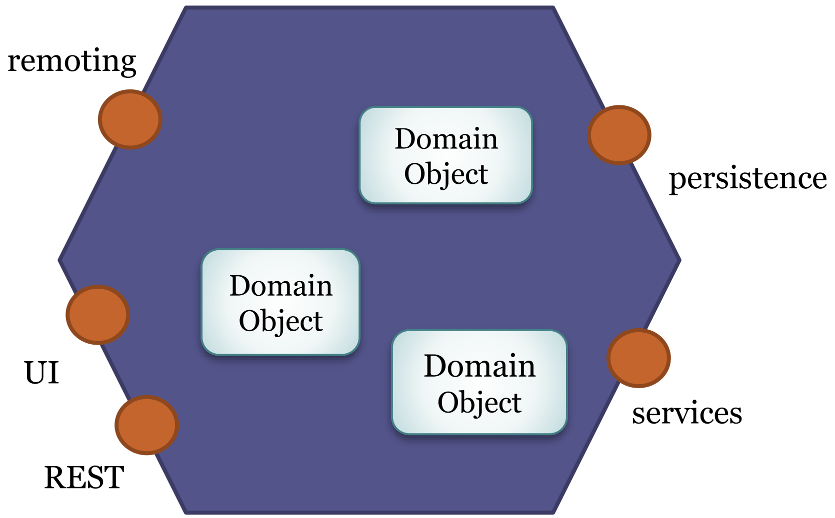
Data centered

- Simple domains based on data
  - CRUD (Create-Retrieve-Update-Delete) operations
- Advantages:
  - Semi-automatic generation (*scaffolding*)
  - Rapid development (time-to-market)
- Challenges
  - Evolution to more complex domains
  - Anemic domains
    - Classes that only contain *getters/setters*
    - Objects without behavior (delegated to other layers)
      - Can be like procedural programming

Anemic Models: `https://www.link-intersystems.com/blog/2011/10/01/anemic-vs-rich-domain-models/`

Naked Objects

- Domain objects contain all business logic
  - User interface = Direct representation of domain objects
    - It can be automatically generated
  - Automatic generation of:
    - User interfaces
    - REST APIs



Naked Objects

- Advantages
  - Adaptability to domain
  - Maintenance
- Challenges
  - It may be difficult to adapt interface to special cases
- Applications
  - Naked Objects (.Net), Apache Isis (Java)

End of Presentation