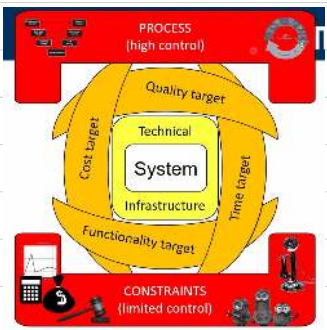
1. CONTEXT of S.E.

Technical & Mgm Domain



Application Domain

- Embedded Systems (ES)
  - → performs some of the requirements of larger systems of a larger systems
  - → characteristic: DEPENDABILITY - EFFICIENCY - REAL TIME
  - → REACTIVE: continual and environmental-driven pace
- Information Systems (IS)
  - → have a more holistic view than ES (business support)
- Cyber-physical Systems
  - → intersection between ES and IS for human use

2. EMBEDDED SYSTEMS

↳ information system, that is integrated into a larger product { real time system, small microcontroller, little memory, programmed in Assembly/C

I) MARKET AND REQUIREMENTS

i. Automotive Engine Control Unit

Requirements => . functionality { Engine Functions => injection/idle control, speed acquisition, exhaust system, ventilator control
Vehicle Functions => cruise control, diagnosis syst., bus communication

- Interfaces
  - SENSORS [Temperature/Pressure/Speed/Acceleration]
  - ACTUATORS [Injectors, fuel pump, ventilator control]
  - INTERFACES
  - APIs
- Process

SOP [Start of production] => differentiates automotive markets from the others

=> a newly realesed vehicle has to meet all requirements by the SOP, at (almost) all costs

requirements freeze one year before SOP

- Quality Assurance { Regression Tests ⇒ HIL (Hardware in the Loop) Tests
  Integration Test

ii. Automotive Infotainment Module

↳ operating control higher than (i) because, even if the engine is off, you might still use the radio, navigation info, maps, ...

Characteristics ⇒ • Users interface (consistent HMI over various displays)
• Drivers of Architecture • Advanced Driver Assistance Systems (ADAS)

iii. ES Domains, Characteristics, Trends

A large application of embedded systems is in trasportation, particularly automotive and have different subdomains: Vehicle motion / Driver Assistance / Body / Connectivity

⇒ Other domains than Automotive have different optimization criteria

- Consumer Products (Quality not very important)
- Home automation (focus on cost)
- Automotive (focus on cost and time-to-market)
- Railway technology (less important time-to-market)
- Aviation, aerospace (Quality very important)
- Industry (t-to-m increasing in importance)
- Medical engineering (' ')
- Defence technology (cost and t-to-m not important)

↳ many domains are included in trends such as IoT and Industry 4.0

back-end connectivity is key

iv. ES conclusions

There is not such thing as the embedded system, there are only commonalities:

- Is integrated on a physical thing
- Functionality is specific to the usage / environment
- Strong interaction with the physical environment through specific hardware / interfaces

II) Architecture and Implementation

i. Automotive Engine Control Unit

=> makes sure they satisfy REAL TIME REQUIREMENTS

- "Torque Path" as functional model
- 3 level monitoring Safety Architecture
  - plausible sensor check
  - 2nd torque path with a simulation that is confronted to real time to find out if smt is going wrong
  - monitoring of micro controller (watchdog), if smt is wrong the system goes in Safety Mode
- C programming and MATLAB simulation
- Calibration is done through 50 000 parameters by tool based calibration
- End line programming in the factory (without "flashing" vehicles rarely work)

ii. Automotive Infotainment Module

- Linux OS (quality / security / innovation)
- State-of-the art processes
  - continuous integration with Jenkins
  - Distributed control with git
- Separated Safety domain for safety-critical application
  - ↳ even if a hacker catches this specific domain, the functionality must be guaranteed

iii. Technology General Trends

- micro controllers with ARM core and specific hardware features (such as machine learning implementations)
- well known system stacks (open source, Linux and Android, REST Back-End connectivity)
- similar tools to classic IT
  - agile processes using JIRA
  - continuous integration
  - test automation

iv. Conclusions

- Increased similarity of ES Engineering to classical SE
- IT security requirements play an important role in architecture and implementation

=> Embedded is about connecting physical products to their services

ex 1 : UNIX, C Programming, Git

(1) Basic concepts from Unix-like OSs

- UNIX Philosophy:
  - write programs that do ONE thing and do it WELL
  - write programs to work TOGETHER
  - write programs to HANDLE TEXT STREAMS, because that is a universal interface

- UNIX SHELL => user interface for access to an OS's services in a command-line interface (CLI)
  - → bash : one of the most popular shells (default on macOS) and a shell session can be started by opening a terminal application

Basic bash commands:

- bash CLI's usual syntax is: name_of_program -opt1 -opt2 param1 (param1 = input)
- echo => print srt. to standard output
- pwd => print current working directory
- ls => list files/directories
- cd => change directory into
  pwd => print working directory again
- Pipes (|, >, <) => direct output to files/commands
- cat => concatenate and print files
- mkdir => create new directories
- man => gives a manual page how to use programs (help feature)
- cp => copy files
- mv => move files between directories
- rm => remove directory entries
- nano => open file with nano text editor
- sudo/su:
  - su usr => switch to user 'usr'
  - sudo ls => execute ls as superuser
  - sudo -u usr nano text => edit file.txt as usr
  - sudo su => switch to the superuser

- SECURE SHELL (SSH) => cryptography network protocol for operating network services securely over an unsecure network
  - → using SSH protocol (stored in .ssh directory), you can connect and authenticate to remote servers
    - ssh => connect to remote server
    - scp => copy files to a remote ssh server

- MANAGING PACKAGES => APT library helps to manage software and is accessed via apt-get
  - apt-get update => re-synchronize the package index files from their sources
  - ' ' install openssh-client
  - ' ' remove ' '

- BASH SCRIPTING => language that allows to write script on OS level. It supports:
  - basic concepts (variables, clauses, loops) - advanced conc. (regular expressions)
  - LOCAL_VAR = ' ' => local variable used inside shell scripts
  - export GLOBAL_VAR = '' => they can specify shell environments and be accessed by
  - $PATH => global variable that contains the concatenated paths to the binaries which can be used inside the shell
  - ls -l => print access control of files
  - chmod => change the file modes / access control options

Ex.
- show all files (included the hidden ones) => ls -a } combination: ls -la
- list long format => ls -l
- navigation → cd .. => move upwards in the directory
- vim
- grep
- sed

(2) C Programming

- BASIC TYPES

	int	char	short	long	long long	float	double
sizeof() =>	4	1	2	8	8	4	8

- BASIC OPERATORS => +, -, *, /, %, ==, !=, >, <, >=, <=, !, &&, ||, =
  - Arithmetic [a+=b => a=a+b] operators: +=, -=, *=, /=, %=
  - Increment ++ / Decrement --

① a=2 b=0
b = a++
a=3 b=2

② a=2 b=0
b = ++a
b=3 a=3

③ a=2 b=0
b-- = --a
a=1 b=-1

- CONTROL STRUCTURES

```
if (x==1){
} else if (x==2){
} else {
}
```

=

```
switch(x){
case 1:
case 2:
default:
break;}
```

```
printf("For loop 0-9:");
for(x=0; x<=9; x++){
    printf("%d\n",x);
}
```

=

```
printf("Do-while loop:");
x=0;
do{ printf("%d\n",x);
    x++;}
```

=

```
printf("While loop:");
x=0;
while(x<=9){
printf("%d\n",x);
x++;}
```

=> break => exit the loop vs. continue => skips current iteration for the next one

- ARRAYS => the user is in charge of the memory in C: they must be initialized with a concrete size. Size of allocated memory depends on { data type / number of elements

. int v[10] => allocates 10 (number of el.) x 4 (byte for int)

. char S = {"Alex"} => allocates 4x1 + 1 (byte for char); \0 => null character, automatically assigned by C at the end of a string

- GESTIONE DINAMICA MEMORIA

```
int *v, n, i;
scanf("%d", &n);
v = (int*) malloc(n*sizeof(int));
for(i=0, i<n, i++){
 scanf("%d", v[i]);}
free(v);
```

=> Introduzione di un puntatore e una variabile
=> L'user può decidere il numero di elementi che vengono salvati
=> Allocazione spazio in base a { n. elementi da salvare / grandezza data type
=> I dati vengono registrati
=> La zona di memoria precedentemente allocata viene liberata

- POINTERS => a pointer is a variable declared to store a memory address
  - Unary operators { & : address-of operator which you use to retrieve the add. of a var.
    * : used to declare a pointer and dereferencing to it

  - b* = b[0] (b+K) = b[K]
  - int *px = NULL, *pararray=NULL; => Initialize the pointer to NULL so that doesn't point to memory where other data is stored
    int x = 1, num_array[2];
    px = &x;
    pararray = num_array;
    printf ("*px : %d \n", *px); => returns '1'

- STRUCTURES => it allows to organise complicated data : it permits a group of related variables to be treated as a unit instead of as separate entities

```
struct point2 { int x;
                int y;
              }

struct rect myRect = { {1,1}, {2,3} };
int area = (myRect.pt1.x - myRect.pt2.x)*
           (myRect.pt1.y - myRect.pt2.y)
```

(3) A Web Server in C

Internet and inter-process communication is implemented using sockets, to build our web server we will use the POSIX (Portable Operating System Interface) sockets API (Application Programming Interface).

=> Most important header to #include are sys/socket.h and netinet/in.h (even arpa/inet.h)

Ex.2 ES & Microcontroller Programming

(1) Hexadecimal System - Signed vs. Unsigned

↳ 16 symbols => 0-9 + "A" - "F" (=10-15) [prefix: 0x or h]

- 1 byte can have values ranging from 00000000 to 11111111 [00 to FF in hexadecimal]
- For binary values, the prefix is b and for decimal values is d
- In the embedded world we have to distinguish types in C as this impacts { the range of values stored / their representation

→ UNSIGNED => es. unsigned integer type with width 8 : uint8_t [0,255]
→ SIGNED => " "signed" : int8_t [-128,127] => { 0x ff = -1 (~~255~~) / 0x80 = -128 (~~128~~)

ex.1

Bin	Dec	Hex
10b	2d	2h
10000b	16d	0x10
0110 1100b	18d	6Ch
1010 1011b	171	0x AB *
1111 1111b	255	FFh

* A=10×16+B=11
F=15×16+F=15

Bin	int8_t	uint8_t	Hex
0110 1100	108	108	0x6c
10101011b	-85	171	0x AB
1110 0110	-48	214	0xd6
1111 1111	-1	255	0xff
10001001b	-119	137	0x89
1111 0110	6	262	0xf6

↳ if the first bit is { 0 => positive signed value / 1 => negative " "

int8_t = uint8_t - 256

(2) C programming : Bit-operators

Bitwise operators in C : AND, OR, XOR (~, &, |, ^) , <<, >>

es.

1) 10001b | 00101b = 10101b
2) 10001b & ~00101b = 1000b
3) 0x E | 25d = 11111b / 0x1F
4) 01110b & 0x19 = 0x01000b
5) 14d ^ 11001b = 10111
6) 0x4 << 2 = 10000b / 0x10
7) 0x5 << 2 = 10100b / 0x14
8) (0x5 << 2) >> 3 = 10b / 0x2
9) 0x55 >> 2 = 00010101 / 0x15
10) (0x55 >> 4) << 4 = 0101 0000 / 50h / 80d

10001 OR 00101 = 10101

10 001 AND 11 010 = 10000

1110 OR 11001 = 11111

01110 AND 11001 = 01000

01110 Exclusive OR 11001 = 10111

00100b << 2

00101 << 2

00101 >> 1

0101 0101 >>4

0101 0101 >>4 → 0101 << 4

Intro to Microcontroller

Terms: - register => chunck of memory with dedicated memory addresses
- field => dedicated group of bits within a register
- pins => tiny 'legs' that surround a chip

=> GPIO (General Purpose Input/Output) : uncommitted digital signal pin on an integrated circuit which may be used as an input or output and is controllable by the user at run time
↳ has no predefined purpose and is unused by default

3) Memory Access

↳ there are are specific regions in the memory - called registers - that we can manipulate (perform read and write actions) during program execution

MSCR

This register only supports 32-bit access. Byte and half-word write accesses are not supported.

↗ there are 264 registers of this type

Address: FFFC_0000h base + 240h offset + (4d × i), where i=0d to 263d

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0		SRC[1:0]		0		OBE	ODE	SMC	APC	0	0	IBE	HYS	PUS	PUE
W																
Reset	0*	0*	0*	0*	0*	0*	0*	0*	1*	0*	0*	0*	0*	1*	0*	0*

Bit	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0								0				SSS			
W																
Reset	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*

* Notes:
- See the Signal Description details for the reset values and the availability of pad setting bits in MSCRs.

this is the 11th MSCR

1) Memory Address of the SIUL2_MSCR 10 => FFFC 0000h + 240h + 4·10d = FFFC 0268h
4·10d = 28h

2) Command that sets only the bits of the fields OBE and PUS to 1
'GPIO Output Buffer Enable' 'Pull Select'

↳ we have to set up a pointer called 'pDesiredMSCR' that references to the 2 fields we're interested in, on the SIUL_MSCR register

*pD |= 0202 0000h

! is for deferencing the pointer instead of overwriting the memory

= 0000 0010 0000 0010 → PUS
0000 0000 0000 0000
↓
OBE

3.1S: ENTERPRISE SOFTWARE ARCHITECTURE EVOLUTION

ENTERPRISE SOFTWARE => software to administrate the system { Mgm Systems => Planning and Controlling; Execution Systems => Administration

ex. SAP S/4 HANA => integrates: finance/procurement/engineering/manufacturing/supply chain/asset management projects/HR/sales or commerce

Cloud computing, ...

=> The technology innovation that we witness at the moment is made possible due to a cost decline in hardware solutions

↳ there was a shift overtime from Client-server to Cloud computing technology

Cloud layers: - Infrastructure as a service (IaaS) => storage in the data server of a provider
- Platform as a service (PaaS) => software development in the cloud run by IaaS
- Software as a service (SaaS) => applications running of providers' infrastructure

- Enterprise Software Deployment Phases from Cloud to "Intelligent" (AI enabled)
  ↳ ON PREMISE 2010 → CLOUD/SAAS 2025 → AI (using structured/unst./3rd party data)

#	Criteria	1. On-premise	2. Cloud / SaaS	3. „Intelligent" (AI, digital)
1	Scope	**System of record** / transactions • Focus on back-office • Main investment in business functionality • Limited automation	**System of engagement** / collaboration • Focus on front-office • Main investment in non-functional requirements „cloud qualities" • Limited automation	**System of intelligence** / orchestration • Focus on automation of administrative processes and assistance for management • High level of automation
2	Architecture	**Monolith** • 3 tier architectures • Single tenancy	**Service-oriented architectures** • Distributed architectures • Multi-tenancy • Internal APIs	**Micro-services-oriented architectures** • Lambda architectures, etc. + External APIs / integration platform
3	Data center / deployment model	On-premise • Mostly local data center • Managed by customer	Cloud • Global distributed data centers • Managed by software vendor	Cloud • Global distributed data centers + Managed by hyperscalers (AWS, Azure)
4	User experience	Screen for **experts** • User base: 20% of employees	Screens for **everybody** on **every device** (responsive design) • User base: 100% of employees + customers + business partners	+ More human-like interaction / assistance (conversational AI/chatbots)
5	Data	**Relational databases** • ERP • Structured • Internal	Relational databases • Enterprise software • Structured • Internal	**Big data & in-memory databases** • Internet platforms • Structure + unstructured => Data lake • First, second, third-party data
6	Extensibility	Coding, modification • Internal developers • IT consultants for customization	**Platform-as-a-Service** + External developers / ISVs + End-user self-service personalization	+ Low or no-code development + Product managers / business analysts + Robotic process automation
7	Business model / pricing	License model	**Subscription model**	+ Consumption model
8	Leading vendors	SAP, Oracle	Salesforce.com, Workday, Netsuite, ...	No clear leader so far

#3 - Cloud qualities : NON-FUNCTIONAL REQUIREMENTS

- Performance
- Stability
- Consumption (Implementation)
- Extensibility
- Integration
- Upgradeability
- Operations ability
- Internal Developer productivity
- Ecosystem co-innovation
- Data Protection & Privacy
- Security

Economics of running an ent. software : Total cost of Ownership (TCO)

TCO => Infrastructure management (IM) + Application mngm cost (AM) + Support (S)
Hardware cost — messages to customers

↳ Cost reduction levers : Automation / Multy-tenancy* / Product Quality

* MULTY TENANCY ARCHITECTURE => shared infrastructure of many customer, where every customer has a logic partition of the same application

#4 The importance of designers has increased over the years to make great user-interfaces

#5 · Traditional Separation between Transactional processing (OLTP) and Analytical processing (OLAP) that didn't make possible REAL TIME data processing and decision making

SAP HANA => Now it's possible to combine the two in one database where can be accessed much easier/faster
↳ IN-MEMORY db that has many libraries (Text Analysis & Mining, Data Models, Predictive/Planning eng.) to implement algorithms directly in the db

#6 EXTENSIBILITY => leveraging the developer community : the more customers, the more valuable the network to existing users

· New Trend : having technology of engaging people without dedicated coding skills
↳ low code tools : robotic process automation
no code tools : builder by engineer.ai

Examples:

- SAP S/4 HANA Evolution ⇒ · Responsive design (decisive / predictive / simulate)
  · Principle of one (1 recommended solution)
  · Re-architecting for in-memory platform (in memory compression)

- PROGRAMMATIC ADVERTISING PLATFORMS [auction length for delivering ads can be <100ms, if it arrives later there's a loss in cash flow]
  ↳ Data management platform: Capture Data ⟶ Analyse Data ⟶ Enable use cases { targeted adv. / content recomm. / direct market.

- Value Works.ai ⇒ startup application to support management

4. CONTEXT of SE

- PROCESSES in S.E.

Software Engineering is a collection of techniques, methodologies and tools* (for problem solving) that help with the production of software system while change occurs.

↳ understand trade-off, make informed decisions and implementing them

*Tools es.: Integrated Development Environment (IDE), Computer Aided Software Engineering (CASE)

⇒ SE. Categories: Req. Engineering, Architecture Design, Design of subsystems, Implementation, Test and quality assurance
↳ PROCESS MODEL ⇒ abstract representation of software process:

1. Waterfall model ⇒ phase oriented
2. Iterative model ⇒ activity oriented
3. Incremental model ⇒ e.g. Agile development

→ heavyweight process (process-centric, rigid)
→ lightweight process (code-centric, agile)

- Waterfall Model ⇒ starting from requirements (conventional view)
- V-Model ⇒ Development activities first and Quality assurance then (for German fed. projects)
- spiral Model ⇒ focus on minimizing risk through incremental, iterative and self-learning process

⇒ UNIFIED PROCESS : iterative software lifecycle model throug 4 phases { Inception, Elaboration, Construction, Transition
↳ software artifact ⇒ software work product produced by a process step

AGILE DEVELOPMENT ⇒ leaner, less burocratic development process with { · fewer rules · fewer not cost effective artifacts

· SCRUM ⇒ software development process is too complex to plain in details
- idea : daily group meetings of self-organizing teams (short duration w/5 minutes)
- PRODUCT BACKLOG : collection of product req. from customer
- SPRINTS : solving a task 100% in a short period of time and merging the different tasks at the end (≠ waterfall approach)
- roles : product owner / team / scrum master

· AGILITY ⇒ short term planning and incrementality, artifact orientation, empowerment, user orientation
- it leads to short time-to-market
- no artifacts that describe the system specifications / characteristics
↳ difficult / tricky to renovate the architecture because of few information
- agility is corporate culture, not only development processes

· ARTIFACTS ⇒ if an artifact doesn't produce customer value, it has to be dropped
· Requirements Engineering (es. System models)
· Software design (es. Architectural / Interface / Component / Database Design)
· Testing (es. Acceptance / System / Subsystem integration test plan)

· QUALITY ⇒ characteristic of a software product that fulfil requirements
· Non functional requirements (Product / Organisational / External Req.)
↳ quality ≠ N.F.R. because quality refers also to the process (not only product)
· External / Internal quality :
- Intrinsic (correctness / completness)
- Design-Time (portability / maintainability)
- Run-Time (Usability / Reliability / Efficiency / functionality)
} attributes applicable at multiple levels of granularity
amount of computation / amount of communication
· Software Quality is difficult to be assessed so many companies develops the software following QUALITY STANDARDS like ISO 9001 (not compulsory by law but very helpful)

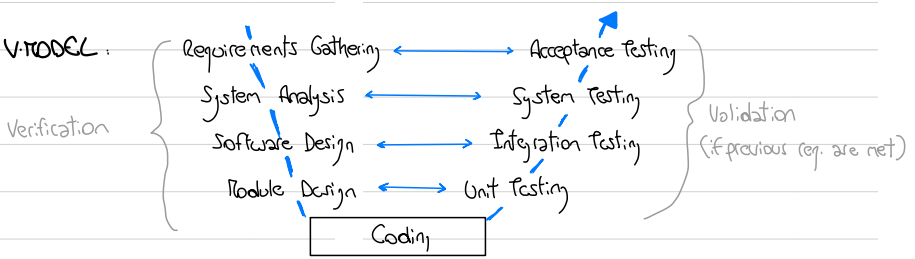
EX.3 SOFTWARE DEVELOPMENT PROCESS & SOFTWARE PROJECT MANAGEMENT

(1) i. Agile vs. Rigid Software Development

- Agile => SCRUM : - definition of requirements in project backlog
  - grouping of multiple user stories into sprints
  - combined development and testing

Roles :
- Project-owner : defines set of to-be-developed requirements (=> PRODUCT BACKLOG)
- Scrum master : supervises the scrum process
- Team member / developer : definition and development of small tasks (from)

- Rigid => V-MODEL :



Roles : more clear and rigid separation [developers / testers / project manager]

ii. - Agile characteristics :
- Short-term (+ reaction)
- People driven (+ motivation)
- Code / Test driven (- documents)
- Customer involvment
- Suitable for short time-to-market
- No clear roles / responsabilitie between team members

- Rigid characteristics :
- Long-term (difficult to spot early faults)
- Easier to exchange people
- Lot of documents (diagrams)
- Little customer involvment
- Aligned with industry standards and development best-practices
- Defined interfaces between roles and activities

iii.
1. Agile : unstable / changing requirements
2. Rigid : automotive suppliers certification / security control - acquiring a new company is easier to integrate if the process model is rigid
   Agile : the importance of time-to-market
3. Rigid : 3rd party contractor for better conformation / communication
   Agile : Mobility App is a setting better suited for agile processes

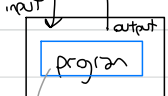
(2) Maven => build management tool that allows the developer to specify dependencies in the build file

pom.xml (=Project Object Model) => Maven configuration file

(3) Http Servlets => small Java Progrms that run within a Web server

↳ a sub-class of HttpServlet must override* at least one method (doGet(), doPost(), doPut, doDelete()) for the respective HTTP methods

2 Environments used { Tomcat [→ Catalina ↘ API]
Jetty



Servlet Container => Different servlets have to be registered into the right container so that the servlet container will call the right servlet on a specific HTTP request

↳ the programs know how to handle a HTTP request and to produce a response to the client

* Method overriding, in object-oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses

5. SOFTWARE ARCHITECTURE

↳ set of functional decompositions of a software system according to non-functional req.
= it is about the functions but has an impact of non functional characteristics

=> it is a set of design decisions :
- efficiency of development process
- risk minimization
- communication means
- knowledge management

- system = collection of components that has : · boundaries · interaction with environment (development / operational / political influences)
· interfaces · static structure with a dynamic behavior

- stake-holders => the architecture is seen as an architectural description seen from many perspectives with different interests (=> viewpoint * : languages for architecture e.g. diagrams)

UML (Unified Modeling Language)

* 4+1 VIEW MODEL :
1 - logical viewpoint => functionality from end-user perspective
2 - process ' ' => design's dynamic communication, concurrency and synchronization
3 - deployment ' ' => mapping into the hardware and its distributed aspects
4 - implementation => software's static organization in its deployment environments
+ scenarios => how the system is used (reflect the requirements)

↳ not all systems require all viewpoints
↳ additional viewpoints : data / security (functionality / logic / deployment)

ZACHMAN FRAMEWORK => gives more perspectives on the system to facilitate the communication between two different parts : no perspective is better, each one fits more depending on the context (their collection is the architectural description)

- models => abstractions of the system (functional, object, dynamic)
↳ collection of languages (Use Case diagram, Class diagram, sequence diagram)
↳ char. of models : reduction / pragmatism (purpose-oriented) / mapping

↳ AS-Is model => current state of system
↳ To-Be model => described the imagined system

I) SOFTWARE MODULES and SOFTWARE COMPONENTS

i) MODULARITY

- Partitioning => top-down approach of partitioning a system in subsystems of high cohesion and low coupling
- Clustering => bottom-up approach to group similar objects
- Encapsulation => push data in a model with access rules
- Dependences => degree of reliance of a component on another (- dependency → + system changeability)
  ↳ they should exist only in the external level (interfaces) but not internal

MODULARITY => decomposition of systems into modules for:

• information hiding • easier maintenance • increase understandability • reduce complexity

es. stakeholder perspective => for a customer modularity may allow more ability for customization

- FUNCTIONAL decomposition => each module is a major processing step in the application domain
  ↳ issue : complexity for even easy tasks (hard understanding)
- MODULAR ' ' => each module is a major abstraction in the application domain
  ↳ issue : depending on the purpose of the system diff. concepts might be found

BLACK-BOX Model => presents the functional perspective on a system and captures the functionality / behavior
es. Control or manageent model of enterprise ↳ functional decomposition

WHITE-BOX Model => presents the construction perspective on a system and captures the construction and the operation of the system
↳ constructional decomposition

=> **Decomposition**: technique to structure the S.A. to analyse dependencies between ELEMENTS to create clusters called COMPONENTS and the INTERFACE

- COUPLING => degree of interdependencies between elements (modules) or components
  ↳ strong coupling would requires to do many changes to upgrade the system

ex. • Pipes and Filters Architecture (low coupling degree)

ex. • Layed Architecture (information hiding / modularization / low coupling)

ii) COMPONENT BASED S.E. (CBSE)

- SOFTWARE COMPONENT => software element conforming to a component model that can be independently deployed and composed without modification of a composition standard => CHARACTERISTICS:

· Independent components are specified by their interfaces

· Middleware provides software support for component integration

· standardized · deployable
· independent · documented
· composable

- COMPONENT INTERFACES: if we describe an interface, we describe abstractly what a component does.

⇒ JAVA PLATFORM MODULE SYSTEM (JPMS): public methods within the public classes within the exported packages that other modules are able to invoke

```
// src/edu/tum/ase/SayHello.java
package edu.tum.ase;
import java.lang.*;

public class SayHello{
  public static void main(String[] args){
    system.out.print("Hello world");
  }
```

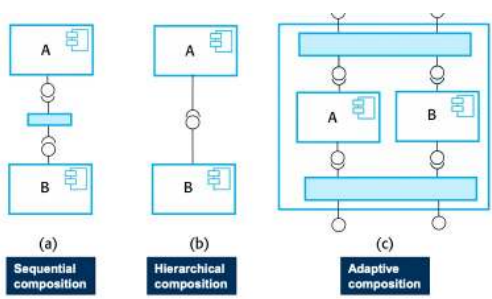
Module Declaration File (Java 9)

```
// src/module-info.java
module hello.world {
  exports edu.tum.ase
  requires java.base;
}
```

EX. → before Java 9 there were no mean to define a module's exported packages

- COMPONENT MODEL [⇒ definition of standards for component implementation/documentation/deployment
  - Interfaces ⇒ component model specifies the interface: definition / elements / language
    ex. WSDL for Web Services, EJB — operation names/parameters/exceptions
  - Usage ⇒ naming convention in order for components to be distributed and accessed remotely
    ex. URI for web servers [META-DATA: helps the user to find out what services are provided and required]
  - Deployment ⇒ packaging of components for independent/executable deployment

- COMPONENT COMPOSITION
  (a) One component in the middle ("glue code") that connect A to B (that are independent
  (b) There's an explicit relationship (direct dependency)
  (c) A and B are independent and have interfaces that allow to talk to both A and B



PRO ⇒ independent components / Component standards / Middleware for component inter-operability / Reusable development

CONS ⇒ . Component trustworthiness (with no available source code)
. Component certification (for functionality/security)
↳ an external library in a secure system weakens the data protection
. Emergent property prediction (two components interacting sometimes have unpredictable behavior)

MICROSERVICE ARCHITECTURE
↳ decomposing a software into: { autonomous, cohesive units; independently deployable*; communicate in a lightweight way }

*⇒ Independent Deployment: infrastructure automation techniques reduce the operational complexity of building, deploying and operating microservices

- continuous integration (build and test automation)
- continuous delivery (to specific environment)
- cloud infrastructure + monitoring (container / virtualization infrastructure)

- LIGHTWEIGHT communication

es. REST => architecture style that has microservices that provide on the web with 6 constraints:

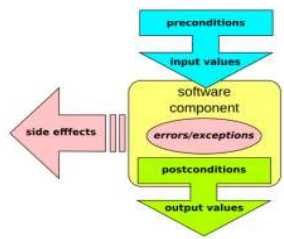
1) Client-Server (seperate evolution)
2) Stateless (no client context stored)
3) Uniform Interface
4) Cacheability => caching responses make it + efficient
5) Layered system
6) Code on demand

iii) DESIGN BY CONTRACT

↳ set of principles to produce dependable and robust object-oriented software [=>REUSABILITY]

A contract is an agreement between the client and supplier

- pre condition => requirements that any call must satisfy
  ↳ if not satisfied, the system stops working
- post condition => properties that are ensured at the end of execution



ex. EIFFEL

- Create a routine put that enters keys and values into a dictionary
- old => value of a var. on entry to the routine old count => value of count on entry to the routine

	Obligations	Benefits
Client	*(Must ensure precondition)* Make sure table is not full and key is a non-empty string.	*(May benefit from post-condition)* Get updated dictionary where the given element now appears, associated with the given key.
Supplier	*(Must ensure post-condition)* Record given element in dictionary, associated with given key.	*(May assume precondition)* No need to do anything if table is full, or key is empty string.

```
class DICTIONARY [ELEMENT]
feature
        put (x: ELEMENT; key: STRING) is
        -- Insert x so that it will be retrievable through key.
require
        count <= capacity
        not key.empty
do
        ... Some insertion algorithm ...
ensure
        has (x)
        item (key) = x
        count = old count + 1
end
```

Header comment
Preconditions
Post-conditions

- Interface : API - Contract - Code
  ↳ + level of abstraction -

What is the "right" level of abstraction for a contract?
- it depends on the purpose

- Machine Learning : learning a function f that can't be explicitely specified (es. face recognition)
  ↳ approximation of a function to get a machine learning algorithm g that interpolates between data
  - g is learn rather than implemented because we don't know how to specify/implement neither f nor g explicitely or it's too expensive to do so
    ↳ function g hence exists without specifications

II) DEPENDENCY STRUCTURE MATRIX (DSM)

↳ 2 dimensional matrix representing the structural or functional interrelationship of objects, ticks, teams

	A	B	C	D
A	-		x	x
B		-	x	
C	x		-	x
D				

↳ the funct. in the columns is dependent on rows
- A depends on C
- C depends on A and B } mutually dependent
- B doesn't depend on any of them
- D depends on A and C

		D	A	C	B
D		-			
A-C	A	x	-	x	
	C	x	x	-	
B				x	

PARTITIONING ⇒ A and C can be regarded as a single composite entity

BLOCK TRIANGULAR FORM ⇒ a DSM that has been rearranged so that all dependencies either fall below the diagonal or within groups

↳ HIERORCHICAL DSM ⇒ the grouping of A and C is shown by their IDENTATION

- LAYERED SOFWARE SYSTEMS ⇒ disentangles concerns when building a software

ex. Junit version

$root		1	2	3	4	5
org.example	application 1	-				
	model 2	X	-			
	domain 3		X	-		
	framework 4			X	-	
	util 5				X	-

⇒ every root depends on the one below (apart from 5 that doesn't depend on anything)

layered system with clean separation of the <= user interface layers from the underlying core logic

the numbers indicate the number of dependencies <=

junit		1	2	3	4	5	6
	awtui 1	-					
	swingui 2		-				
	textui 3			-			
	extensions 4		1		-		
	runner 5	3	8	4		-	
	framework 6	5	7	6	6	5	-

$root		1	2	3	4	5	6
org.example	artifact-test 1	-					
	core 2		-				
	project 3		X	-			
	artifact-manager 4		X	X	-		
	reporting-api 5		X			-	
	model 6		X	X		X	-

'core' depending on almost every component
↳ you risk change propagation [not optimal]

⇒ tool support for DSM : they make suggestions to reduce dependencies and get the block-triangular form

- BENEFITS of DSM : - the partitioning algorithms provide an automatic mechanism for architectural discovery in a large code base (possibility to spot structural patterns at glance)
  - integration of dependency rules { C1 can-use C2 / C1 cannot-use C2

- CHALLENGES of DSM : - unknown independencies can exist as they can emerge unexpectedly
  - DSM is less intuitive when compared to a graph

III) GUIDELINES for MODULAR DESIGN

i) LOW COUPLING and HIGH COHESION

- Parent relationship between components => parent : $C \rightarrow C$
- Interface-component relationship => assigned : $I \rightarrow C$
- Connection between interfaces => connected : $CON \rightarrow (I \times I)$

formalizing

$$\text{COUPLING}(S) = \alpha * \frac{\text{number of connection with relativ interfaces and same parents}}{\text{normalised number of connections}} + (1-\alpha) \frac{\text{'' different parents}}{\text{'' ''}}$$

α: weighting function 'NOT SO BAD' — same hierarchical level — 'A BIT WORSE' — different hier. level

I) · DATA COUPLING => component passes data to another component

ex calculateSalary (String name, int noOfHours, int salPerHour)

II) · COHESION => how closely related are different responsibilities of a component
↳ internal interdependencies

Strive for:
- low coupling
- high cohesion

I) · CONTENT Coupling => one component directly affects the working of another comp.

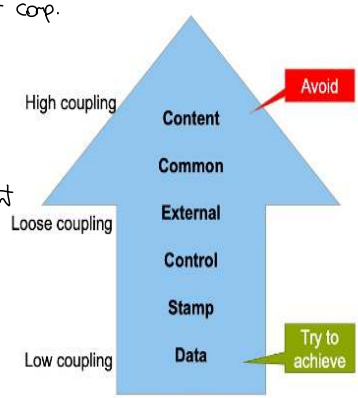
· COMMON '' => two components have shared data
↳ lack of responsibility / reduces readability / difficult to reuse components

· EXTERNAL '' => components communicate through an external medium

· CONTROL '' => one component directs the execution of another component
- good => if parameter allow factoring and reuse of functionality
- bad => if passed parameters indicate completely different behavior

· STAMP '' => complete data struct. are passed between components
ex. calculateSalary (Employee employee)

· DATA '' => component passes data to another component ex calcSal (string, int nofHrs, int salperHr)



II) · COINCIDENTAL => no significant relation between elements [have to separate modules]

· LOGICAL => components are logical but not functional related

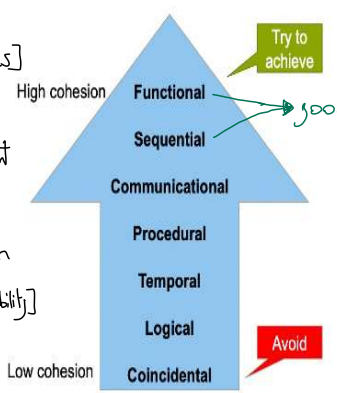
· TEMPORAL => functionalities invoked at same time even though they're logically independent
↳ code spread-out: actions weakly related but strongly to actions in other modules

· PROCEDURAL => elements are related only to ensure a particular order of execution

· COMMUNICATIONAL => elements of a component operate on same (external data) [-reusability]

· SEQUENTIAL => data flows between parts

· FUNCTIONAL => every essential element to a computation is contained in the component
↳ + reusability / testability / understandability / learnability / maintainability / extensibility of product



formalizing
↳ COHESION
1. Syntactically defined : analyse the binding of methodes and used attributes
2.* Description-based heuristic : use natural langage to describe the function of the module
3. Data flow : more d.f., more cohesion

heuristic rules

2.* Heuristics => describe the component using one sentence in natural language and analyse it using

- if the sentence has to be a compound sentence, contain commas or more than one verb => communicational / sequential
- if the sentence contains temporal terms such as "first", "after", "start" => sequential / temporal binding
- if the sentence has more than one object => LOGICAL cohesive

LCOM (Lack of Cohesion in Metrics) => number of method pairs that don't use the same attributes

ex. LCOM, TLCOM, CAMC, NHD, ... → if semantically similar, then there should be cohesion

ii) SINGLE RESPONSIBILITY PRINCIPLE

↳ 'a class should only have one reason to change' => cohesion at package/implementation level
if there are two reasons for a class to change, we have to split the functionality into 2 classes

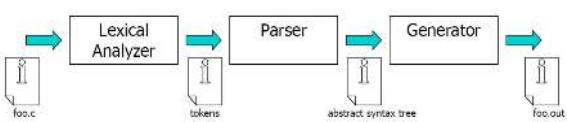
=> responsibility = a family of functions that serves one particular actor

iii) SEPARATION of CONCERNS

↳ components should only encapsulate semantically related functionalities

- Aspect-oriented programming => separate sort that occurs in all the code from the rest

ex.



=> functional cohesion:
3 independ functionalities that are encapsulated within component

OPEN/CLOSED PRINCIPLE => software entities (classes/functions/modules) should be { open for extension, closed for modification }
extending behavior doesn't result in changes to the source, binary or code of the module
↳ avoids unanticipated effects on dependent components

1. Client → Server => Closed Client

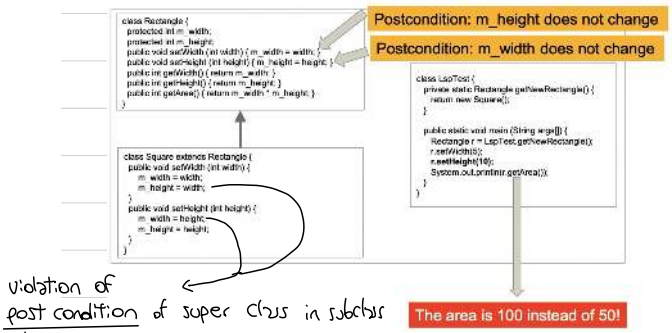
2. Client → Abstract Server ⇒ Open Client
Server

Client talks to an interface of the Server

iv.) LISKOV'S SUBSTITUTION PRINCIPLE (LSP)

↳ any instance of a subtype also satisfies that property

- child classes should never break the parent clss' type definitions
- all derived classes must be substitutable for their base classes



violation of post condition of super clss in subclss

↳ if we change the height/width, we don't chge the width/height

=> LSP is a great programming idea but can't be easily checked

- we may see objects as state machines: when redefining states of the state machine in a subclss, we need to make sure that transition retargeting doesn't violate the super clss's state invariant of the transition's target

v) INTERFACE-SEGREGATION PRINCIPLE (ISP)

↳ clients should not be forced to depend upon interfaces they don't use [=> single responsibility principle]

the idea is to have small interfaces to make sure that classes are not forced to implement methods that don't make sense in their specific context

VIOLATION of ISP

↳

```
interface IEmployee {
  public int calculateWorkHrs();
  public int getNoOfVacations();
}

class WiMi implements IEmployee {
  public int calculateWorkHrs() {
    // .... returns the work hrs
  }
  public int getNoOfVacations() {
    // ..... returns no of days in vacation sheet
  }
}

class HiWi implements IEmployee {
  public int calculateWorkHrs() {
    //.... returns the work hrs
  }
  public int getNoOfVacations() {
    throw new NotImplementedException();
  }
}
```

```
class Secretary {

  public List getAllEmployees() {
    // returns a list of hiwis and wimis
  }

  public void manageVacations() {
    List allEmployees = getAllEmployees();
    for(IEmployee emp : allEmployees) {
      int nov = emp. getNoOfVacations();
    }
  }
}
```

vi.) ANTICIPATE CHANGE

- have an architecture that guarantees reusability by compromising between generality and specificity
  ↳ low coupling and high cohesion over change

- you may have an overly complex diagram not needed entirely

vii) DON'T REPEAT YOURSELF

↳ every piece of knowledge must have a single, unambiguous, authoritative representation within a system

- redundancy has to be avoided because causes higher maintenance efforts, inconsistencies and reduces understandability
- How does DUPLICATION arise?

1 - IMPOSED Duplication => multiple representation of information (developers feel that environment requires that)

e.g. application with different languages on client and server, both need to represent some shared structure on both

↳ GENERATIVE PROGRAMMING : single representation that generates code or DSLs (data description languages)

2 - INADVERTED Duplication => bad design decisions (developers don't realize that they're duplicating info)

e.g. encoding 'driver' twice for 'trucks' and 'delivery routes'

↳ need to store the value and manage the synchronization of that [tradeoff { efficiency / maintainability }]

3 - IMPATIENT and INTER-DEVELOPER Duplication => developers are lazy and prefer to duplicate code rather than understand/ the code of others

↳ willingness to spend time to make code easy to reuse to avoid pain later by:

- have clear design, strong tech. project leader, clear division of responsibilities
- encourage communication between developers
- have a central place in the source tree where utility routines and scripts can be deposited

⟹ Conclusions:

INTERNAL QUALITY => MODULARITY influences { maintainability / reusability / testability }

EXTERNAL QUALITY => Users (not developers) are concerned with ext. quality:

- many layers may lead to many indirections (→ bad performance)
- a large attack surface (many entry points), it makes a system vulnerable

3 Tiers Architecture => web server also runs application server { + efficient / − larger attack surface }

4 Tiers Architecture => web and app. servers separated { + attack surface reduced / − more efforts invested on operations }

↳ DEMILITARIZED ZONES (DMZ) : separate internal network as much as possible from "the hostile internet"

ex. Sen:- automated surveillance and burglary detection

- Key functional requirements : · aggregate sensor and camera data · accessible through mobile devices · send notification to security personnel · monitored by customers

- Key non-functional requirements : · secure and reliable · scalable · privacy-preserving · fast and cheap

→ they must be defined more clearly (they're too vague) to be measurable / mappable, description of alternatives:

(1) Component & Connector View =>

A) Decomposition not deployment-oriented
- scalability (-) · maintainability (-)
- portability (-) · efficiency (+)

B) Decomp. deployment-oriented
- scalability (+) · maintainability (+)
- portability (+) · efficiency (-) => cross-component

*

(2) Module View =>

A) 2 "epic-classes"
- portability (-) · maintain. (-) · efficiency (+)

B) Sequential processing (sep. of concerns)
- efficiency (+) · scalability (+) · maint. (+)

C) Generic measurment controllers
- maintain (+) · portability (+) · scalability (+)

* => (1) and (2) form the Allocation View

EX.4 ARCHITECTURE VIEWPOINTS, COMPONENT-BASED S.E. and REST

(1) UML diagrams => a type of model to have different views on system

↳ 4 + 1 View Model :
- Logical => End-user / Designer [Functionality - Abstractions - Mechanism - Sep. of Concerns]
- Process => System Integrators [System interaction, Sequence, Performance, Scalability]
- Implementation => Programmers [configuration mgm]
- Deployment => System engineering [System topology, Communication, Provisioning]

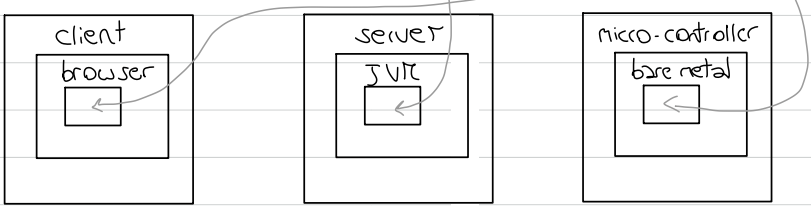
=> DEPLOYMENT DIAGRAM for dark mode service :

(i) device => computer (client & server), micro controller

(ii) execution environments => JVM (Java Virtual Machine), browser, bare metal

no environment for micro-controller

(iii) artifacts => dark-mode.jar, Online Ide.js, tcp-client.elf



=> Model-based S.E. (MBSE)

- several stake holders work collaboratively ; different perspectives
- consistent documentation of the overall system
  - ↳ inconsistency [state of conflict] { a contradiction between 2 models / a discrepancy between the system as-is and its documentation

=> if the deployment diagram changes, you might have to adapt there by changing smt in the other views

ex. - change of artifact name => change it into other views not to create inconsistencies

add RPOS to mC => change of environment => doesn't change other views as the env. is present only in dep. view

exchange mC => - change of device => doesn't change other views as it's not specified the type of Micro C.

add button on microcontroller => - change of structure => changes everything (user-interaction, ...)

logical view

(2) REST API with HTTP SERVLET => architectural style to design Web services

=> RESTful Web Service { HTTP => communication protocol
URI => how resources are identified } identifier to obtain a greeting (GET /greeting)

↳ APIs designed to request a resource by providing a URI (ex. URL + GET request method)

XML Schema Definition (XSD) => · use it as a documentation artifact
· providing to someone else

↳ Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding document
-even though XML is established in Web communication, JSON data format is slowly replacing XML for WEB APIs

(3) Spring Boot REST API

· A controller (~ HTTP Servlet) contains functions that define behavior that is executed when requests are arriving at the Web Server

↳ @RequestMapping => annotate methods with parameters to be used in the function

@SpringBootApplication => creates a Spring Context with instances of classes that only exist once

↳ looks for annotations of object (**Spring Beans**) that Spring handles the dependencies between the various Beans

beans are components defined at the core of the Spring framework e.g. { @Controller, @Service, @Component, @Repository

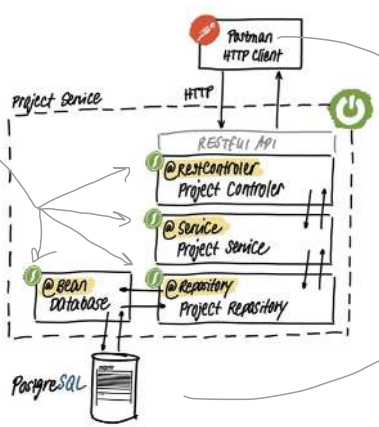


Figure 1: Big picture of the *project service* of the OnlineIDE

Postman => tool that allows to send request by selecting http port and has a nice interface

PostgreSQL => relational db (in ONLINE IDE is responsible for storing and querying project entities)

⇓

Spring Bean handles the connection to the database

EX. 5 MODULARITY, DESIGN by CONTRACT, DEPENDENCY STRUCTURE MATRIX

(1) ASSERTION => small statement that specifies a certain predicate that must evaluate to true once the assertion is hit during program execution [otherwise program is terminated with an exception]

↳ assert [boolean predicate];

⇒ the assert construct can help to support an informal design-by-contract style of programming. Asserts can be used for:

- PRECONDITION [what must be true when a method is invoked]

↳ you can use an assertion to test a non public method's precondition that you believe will be true no matter what a client does with the class

ex.
```
private void setRefreshInterval (int interval) {
    assert interval > 0 && interval <= 1000 / MAX_REFRESH_RATE : interval;
    ... // set the refresh interval
}
```
↳ the above assertion will fail if MAX_REFRESH_RATE is greater than 1000 and the client select a refresh rate greater than 1000 [this would indicate a bug in the library]

- POSTCONDITION [what must be true after a method completes successfully]

↳ you can test postcondition with assertions in both public and non public methods

Answers:

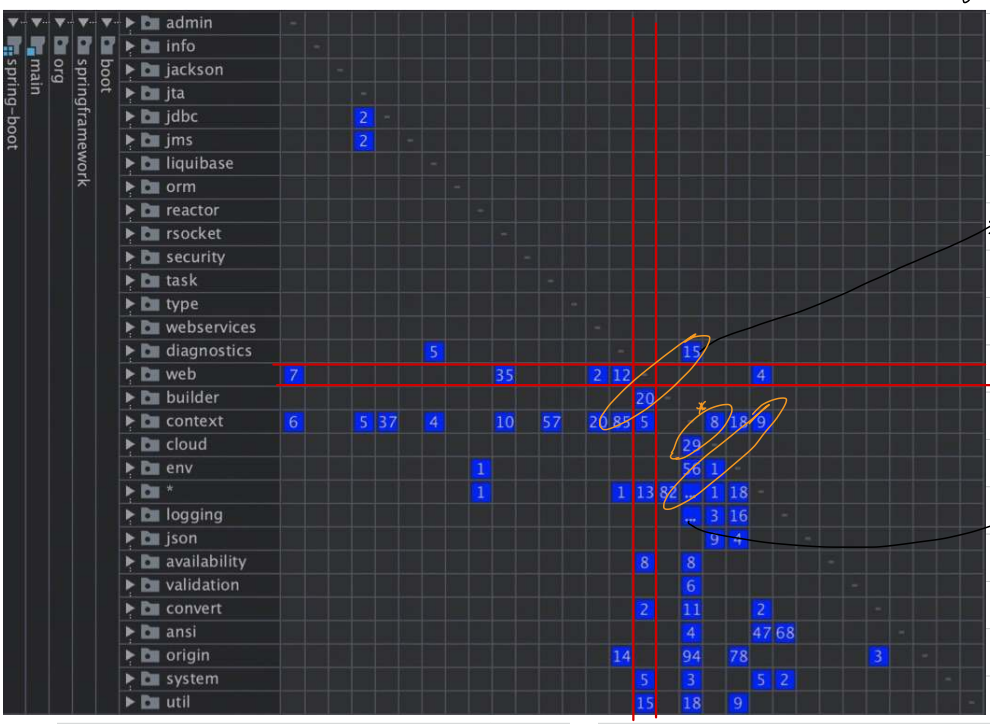
1. The code is a sorting algorithm of an array
2. To check if the lenght of the array corresponds to the value stored in 'len'
3.
```
assert input != null;
assert len == input.lenght;
```
} pre-conditions to make sure only valid parameters are passed to the function

4.
```
for (int i = 0; i < input.length -1; i++)
    assert input[i] <= input[i+1];
}
```
} post conditions to make sure that output is not wrong (array is not sorted)

5. These assertions don't prevent the function from providing wrong results as e.g. the function is not prevented from giving different outputs from the inputs given (the original numbers in the array)

(2) Dependency Structure Matrix (DSM)

- it is a compact way to express dependencies in a software project
- most IDEs provide DSMs on module, package or class level → Spring Boot package's DSM by Intellij IDE



- A column depicts the packages that one package uses
  - ↳ the higher the number, the higher coupling (class that uses functionalities from other packages)
- A row depicts how many times one package is being used from others
  - ↳ the higher the number, the less cohesive (not isolated, many packages rely on it)

1. sum (column) = 68
2. sum (row) = 60
3. Cyclic Dependencies (es: * context depends on cloud and viceversa)
   - ↳ an ideal scenario is to have elements only in the bottom left triangle
4. *

EX. 6 SOFTWARE ARCHITECTURE DESIGN PRINCIPLES & QUALITY

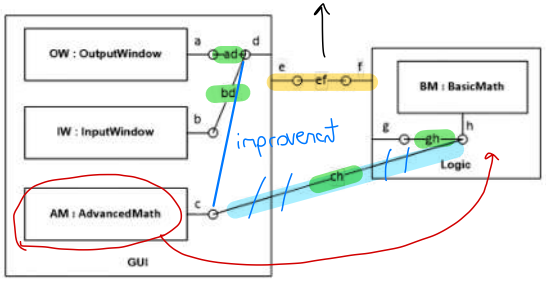
(1) Software Architecture Principles

=> Levels of Coupling: DATA - STAMP - CONTROL - COMMON - CONTENT

[Robust] Loose ⟶ ⟵ Tight

less ↑ interdependencies / coordination / info flow ↑ more

1. component with same hierarchical lvl



Components : C = {OW, IW, AM, BM, GUI, Logic}

Interfaces : I = [a, b, ..., h]

Connections : CON = {ad, bd, gh, ch, ef} = 5

=> Parents : parent(GUI) = env
parent(BM) = Logic

=> The interface h is assigned to IW Component

=> connected(ad) = (a, d)

2. $coupling(s) := \alpha * \frac{|\{con \in s.CON \exists i,j \in s.I : (i,j) = connected(con) \wedge parent(assigned(i)) = parent(assigned(j))\}|}{s.CON}$ => same parent of component

$+(1-\alpha) * \frac{|\{con \in s.CON \exists i,j \in s.I : (i,j) = connected(con) \wedge parent(assigned(i)) \neq parent(assigned(j))\}|}{s.CON}$ => NOT

$\alpha = 0,5$ for simplicity $\quad Coupling(S) = C(S) = 0,5 \cdot \frac{1}{5} + 0,5 \cdot \frac{4}{5} = 0,5$

3. ch breaks the principle of "information hiding", but even if we try to improve the graph, the coupling doesn't change (due to $\alpha = 0,5$)

4. Given the semantics of EZCalculator, the functional cohesion can be improved by moving AM from GUI to Logic because of "separation of concern"

(2) Liskov's substitution Principle (LSP)

=> if a program module is using a base class T, then an object of type T can be replaced with an object of a derived class S, without affecting the functionality of the program module

ex.

```java
public abstract class Vehicle {
  private Door door1;
  private Door door2;
  private Door door3;
  private Door door4;

  public abstract void drive();
  public abstract void addLuggage();
  public abstract void playRadio();

  public void unlockFourDoors() {
    door1.unlock();
    door2.unlock();
    door3.unlock();
    door4.unlock();
  }
}
```

```java
public class VehicleHireService {
  // ...
  public Vehicle hireVehicle() {
    return availableVehiclePool.getNextVehicle();
  }
}
```

1.

```java
public class LamborghiniCar extends Vehicle {

  public LamborghiniCar() {
    setDoor1(new Door()); // or some other door implementation, e.g. `new FrontDoor()`
    setDoor2(new Door());
    setDoor3(null);
    setDoor4(null);
  }
}
```

=> the new LamborghiniCar class violates the LSP because you can't substitute all object of class Vehicle with Lamborghini type because unlockFourDoors() has 4 Doors, while Lamb... only 2

3.

```java
public class BMWX1Car extends Vehicle {

  public BMWX1Car() {
    setDoor1(new Door());
    setDoor2(new Door());
    setDoor3(new Door());
    setDoor4(new Door());
  }
}
```

4. ⟹ BMWX1Car doesn't violate LSP because all 4 doors can be unlocked

(3) LSP - revisited

1. * For normal behavior, you would expect for the Account class that the postconditions (state invariants) hold unless the account is closed
   ↳ but in the StudentAccount the invariant is broken
   * because you withdraw money, but the balance isn't changed

2. Since you should be able to replace any object of the superclass by an object of subclass, without limiting what the superclass can do
   ↳ CAN HAPPEN the RESTRICTION: superclass can withdraw and sub not

ex.

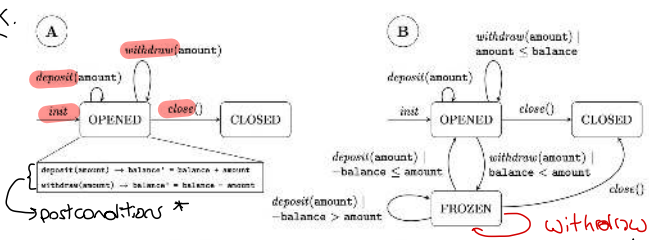


Figure 2: State machines for the two classes Account. (A) and its subclass StudentAccount. (B)

```java
import java.math.*;

class Account {
    protected BigDecimal balance = BigDecimal.ZERO;
    protected boolean closed = false;

    public Account() { }

    public boolean isOpen() {
        return !closed;
    }

    public void printBalance() {
        System.out.println(balance);
    }

    public void close() {
        closed = true;
        assert !isOpen();
    }

    public void deposit(BigDecimal amount) {
        if (isOpen()) {
            assert amount.compareTo(BigDecimal.ZERO) > 0;
            BigDecimal prevBalance = balance;
            balance = balance.add(amount);
            assert prevBalance.add(amount).compareTo(balance) == 0;
        }
    }

    public void withdraw(BigDecimal amount) {
        if (isOpen()) {
            assert amount.compareTo(BigDecimal.ZERO) > 0;
            BigDecimal prevBalance = balance;
            balance = balance.subtract(amount);
            assert prevBalance.subtract(amount).compareTo(balance) == 0;
        }
    }
```

relation

```java
class StudentAccount extends Account {

    public StudentAccount() {
        super();
    }

    @Override
    public boolean isOpen() {
        return !isFrozen() && super.isOpen();
    }

    public boolean isFrozen() {
        return balance.compareTo(BigDecimal.ZERO) < 0;
    }
}

public class BankingApplication {
    public static void main(String[] args) {
        Account a = new StudentAccount();
        a.printBalance();          ⟹ 50
        a.deposit(new BigDecimal("50"));
        a.printBalance();          ⟹ 0
        a.withdraw(new BigDecimal("60"));
        a.printBalance();          ⟹ -10
        a.withdraw(new BigDecimal("20"));
        a.printBalance();          ⟹ -10
    }
}
```

*

3. Solution => we throw an exception that is part of the behavior of the superclass => LSP is NOT violated

EXTENDING mean: to add behavior to a class that is not restricting the behavior of the superclass

```java
import java.math.*;

class Account {
    protected BigDecimal balance = BigDecimal.ZERO;
    protected boolean closed = false;

    public Account() { }

    public boolean isOpen() {
        return !closed;
    }

    public void printBalance() {
        System.out.println(balance);
    }

    public void close() {
        closed = true;
        assert !isOpen();
    }

    public void deposit(BigDecimal amount) {
        if (isOpen()) {
            assert amount.compareTo(BigDecimal.ZERO) > 0;
            BigDecimal prevBalance = balance;
            balance = balance.add(amount);
            assert prevBalance.add(amount).compareTo(balance) == 0;
        }
    }

    public void withdraw(BigDecimal amount)
            throws InsufficientBalanceException {
        if (isOpen()) {
            assert amount.compareTo(BigDecimal.ZERO) > 0;
            BigDecimal prevBalance = balance;
            balance = balance.subtract(amount);
            assert prevBalance.subtract(amount).compareTo(balance) == 0;
        }
    }
```

```java

class StudentAccount extends Account {

    public StudentAccount() {
        super();
    }

    @Override
    public boolean isOpen() {
        return !isFrozen() && super.isOpen();
    }

    public boolean isFrozen() {
        return balance.compareTo(BigDecimal.ZERO) < 0;
    }

    @Override
    public void withdraw(BigDecimal amount)
            throws InsufficientBalanceException {
        if (isFrozen()) {
            throw new InsufficientBalanceException();
        }
        super.withdraw(amount);
    }
}

public class BankingApplicationSolution {
    // Note: You should handle the exception rather than passing it on.
    public static void main(String[] args)
            throws Account.InsufficientBalanceException {
        Account a = new StudentAccount();
        a.printBalance();
        a.deposit(new BigDecimal("50"));
        a.printBalance();
        a.withdraw(new BigDecimal("60"));
        a.printBalance();
        a.withdraw(new BigDecimal("20"));
        a.printBalance();
```

(4) Decoupling Frontend and Backend (App) Server [3-tier → 4 tier architecture]

- SECURITY => if you split your application server into a front-end server and an application server, then the attack surface is smaller (at least 1 layer hidden)

- SCALABILITY => better scalability is expected because you can spawn multiple instances of a specific front-end web server (=microservices) and have a router that redirects the request to the instances of the server

=> REFACTORING a monolithic application.

1. Create new application for Back End (BE)
2. Remove dependencies from Front End (FE) and BE that are unnecessary
3. Set ports differently
4. BE : refactor the controller to be a Rest Controller (=> Json API)
5. FE : . Replace direct calls from the DataBase (DB) with request to the BE
   . Remove parts from FE that relate to DB access

6. ANTI-PATTERNS

↳ solution that don't work (sometimes as a consequence of { insuffient comm. w/ client / unfulfilled req. / insufficient testing }

- Software Development Anti-patterns ⇒ describe useful form of software refactoring
- " Architecture " ⇒ focus on the system-lvl and entreprise-lvl structure of applications and components

- "7 deadly sins" in software practice :
  1. Haste ⇒ sacrifice of long term benefits for faster development
  2. Apathy ⇒ ignoring future challenges and problems with poor support for change
  3. Narrow-Mindness ⇒ not using patterns of other developers to similar problems as yours
  4. Sloth ⇒ poor decisions based on 'easy' answers
  5. Avarice ⇒ over complicating things (no use of abstraction)
  6. Ignorance ⇒ not acknowledging problems for 'intellectual sloth'
  7. Pride ⇒ instead of reusing, you always try to do things yourself

ex. "The blob" ⇒ one 'god class' monopolizes the processing that has all responsibilities (NOT SAFE)
· no understanding of O.O.P. · lack of architecture · limited intervention
↳ seperate main class in secondary classes
↗ "No Object Oriented"

ex. "No OO" ⇒ only functional design (1 class x function) ⟶ too many relations between classes
· lack of obj-oriented understanding · lack of architecture · specified disaster

ex. "Autogenerated stovepipe" ⇒ system that was meant to operate on 1 machine, runs on multiple
↳ a separate, larger-grained object model should be considered for the distributed interfaces

ex. "The Golden hammer" ⇒ reuse the same solution to different problems
· mindset of figuring out a solution before understanding the whole problem

ex. "Design by committee" ⇒ no clear responsibility : design made by many people with no prioritization

F. REUSE

↳ reuse the solution for same problems in other projects (works well in { libraries / frameworks })

⇒ ISSUES:

Technical { · Reuse for small parts of code doesn't have many benefits and may have a negative impact on non-functional requirements
· For re-using components, you need a fully understanding of interfaces and overall system NFRs

Organis. { · Reuse is rarely planned in advance and is caused by lack of motivation
· There is a lack of marketplaces and widely deployed standards

· There's a conflict between flexibility and stability: { need to fit multiple contexts without leading to expensive adaptat. / need to be stable to generate reusability benefits without limiting generic applicability
↳ need to assess trade-offs (Product Line Engineering)

⇒ TYPES of reuse { 1 → OPPORTUNISTIC => reuse of artifacts that were not meant/designed to do so / 2 → PLANNED => systematic planning and development of making artifacts reusable (Soft. Product Lines)

1. Problem of "Ad Hoc" reuse: Clone and Own Paradigm => having to manage 2 artifacts that are almost the same and need much effort (search, evaluation, adaptation, integration)

2. · Configurable Software Solutions (functionality parametrized embedded in deployment routine)
· Software Product Lines (stable core architecture with managed variability modeling)
· Frameworks (planned development of reusable libraries of artifacts)

· VARIABILITY

{ EXTERNAL => variability domain visible to customers (es. door lock identification mechanism) / INTERNAL => variability of domain artifacts that is hidden from customers

· Managed variability (defining var. in domain engineering and exploiting var. in application engineering)

Variability Management => var. can be defined at { as an integral part of dev. artifacts / in a separate variability model

i. REQUIREMENTS ↳ Orthogonal Variability Model: model that defines the variability of a software product line
· Variability is typically modeled as FEATURE DIAGRAMS with variation points (repres. of variability subjects) to describe the features (mandatory, optional, alternative) of a product line member

ii. CODE LEVEL ↳ Var. at code lvl can be supported by programming paradigms (Conditional Compilation - Polymorphism - OOP)

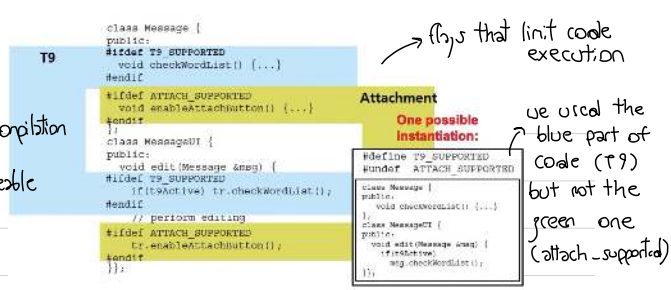
- Conditional Compilation

↓

- Decouple common from variable code
- Automatically include or exclude var. code from compilation

PRO
- variable parts are easily identifiable and manageable (not to cross-cut syntactic borders)

CONS
- pre-defined variants that may result into inconsistencies or hard to manage



flags that limit code execution

we uscal the blue part of code (T9) but not the green one (attach_supported)

- PRODUCT LINE ENGINEERING (PLE)

→ iii. ARCHITECTURE LEVEL => var. at this level is handled through PLA

- PLA is a set of application with a common architecture and shared components, with each application specialized to reflect different requirements
- PLE has the focus of proactive development for reuse => building applications for mass customization (adopting traditional hardware product line approaches ex. automotive engineering) through managed variability to systematically model commonalities and the differences in software applicatio
  - → specializations:
    - Platform => different versions for different platforms
    - Environment => ' operating environments
    - Functional => ' customer requirements
    - Process => ' buisinesss processes

- Why PLE?
  - Cost
  - Quality
  - Time-to-market => enables shorter cycles (than Single System Development) by reducing necessity of re-development and customization

- Product Line Life Cycle:

(1.1) Product Line Artifact Based: acts as a repository for reusable artifacts of a product line
→ captures characteristics of p.l. artifacts (content - life cycle phase - granularity - data type)

1. Family Engineering: it takes advantage of commonalities within products of a company
→ contains common elements and defines the scope* of software p.l.

2. Application Engineering: instantiate the defined variabilities according to specific product requirements
→ exploits commonality and variability of the software p.l. during the p.l. application development

* SCOPING => identify what is going to be reused and what not through establishing boundaries based on concrete product requirements by tackling { existing products, competitor products, future or envisioned products }
→ identify / uses / prioritize products / features / subdomains / assets through - for example - a product feature matrix => STRATEGY (future prediction)

- REFERENCE ARCHITECTURES and FRAMEWORKS (es. Spring Boot)

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than classes

- Extending framework : adding concrete classes that inherit operations from abstract classes in the framework
- Extension Points : where a framework can be adapted

- Class Library : control flow within the application
- framework : inversion of control => invert the control flow of the application

=> the architecture of many web application frameworks is based on MVC patter (composite patter that includes observer, strategy and composite patterns

=> a REFERENCE ARCHITECTURE gives an abstract structure of architecture for software systems
- a system may conform to more than one ref. arch., if its structures can be found in it
- types of Ref. Arch. :
  - Functional (structure sys. functionality through functional areas)
  - Logical (define layers and components to be implemented)
  - Technical (define programming language and infrastructures to be used)

ex. Database-centric architectures, Component-based architectures (Spring, Android), Service-oriented architectures (web services, RESTful HTTP)

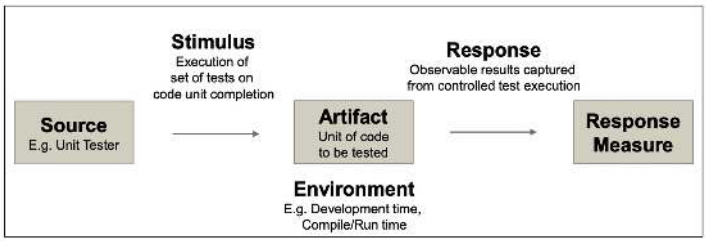
8. TESTABILITY

Problems with testing some systems with certain characteristics:
- Testing GUIs => How to write automated test
- Testing Databases => test means changing the state (difficult to reproduce the test)
- Testing cloud-based systems => difficult to observe/track the change of the state
- Testing embedded systems => you can't look from the outside what is happening inside

- Software Testing => process of finding input values to check against a software to reveal failures
  - DEBUGGING : process of finding a fault in the code given a failure

30-50% of developing costs are taken by testing

- Software Testability => facilitate designing, executing and analyzing test and reducing the cost of testing
  - ↳ what is a good test case? - one that finds a potential bug with good cost effectiveness



=> Definitions of software testability:

1. Test efficiency => facilitation of testing (establishing test criteria and if they were met)
   - difficulty to develop for a program P a test suit satisfying test criteria C
     - ↳ testability depends on both P and C
   - not all test selection criteria correlate with bug detection likelihood
     - ↳ testability is "how difficult (size required by test suites) is to test system"

2. Test effectiveness => facilitation of revealing faults (how easily software exposes faults when tested)
   - RIPR Model: 4 conditions necessary for a failure to be revealed
     - Reachability (fault is reached)
     - Infection (execution of the fault leads to an incorrect program state)
     - Propagation (infected state raises a failure due to an incorrect final state)
     - Revealability (the observed program state through the test oracle must contain incorrect parts)

     ex. Smartcard manufacturers need to make sure that blocking PINs actually works
     Problem => the internal state actually allows the PIN to be entered
   - testability as "the likelihood of a program to fail with the next test if the software includes a bug"
   - properties of programs affecting testability:
     - observability - isolateability - controllability - complexity / simplicity

3. Information loss => it occurs if the values computed in a program are not propagated to the output
   - ↳ Domain / range ratio (DDR) correlated with implicit losses
     - + output → − DDR : "more information" means "more can go wrong" → better testability
     - having large DDR is a problem for testability

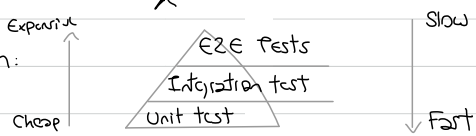
4. Injecting faults => by "purposly getting things wrong" and performing random testing and see if the change of the program yields different output
   - measure the likelihood on an injected defect changing output
     - high likelyhood => high testability

5. Measure for testing effort => testability (effort to specify test cases, develop drivers, oracles) correlates with properties of code, contracts and requirements

=> Tactics for Testability
- Test-driven Development => making sure the code is developed to be tested
- Design for Testability
  - Control and Observe System State
    - little info hiding mean lots of required testing
    - weak cohesion means that need of testing (too) byg subsystem
    - strong coupling makes unit testing more difficult

=> Ceterum Censeo [= furthermore] 

Software Test Automation:



→ what is an itgration test of 2 or more microservices? functionality emerges from single microservices

Microservices (rather than Monolithic) seem to favour integration tests over unit tests so architectural styles inpact on the way we're testing

EX.F ANTI-PATTERNS & SOFTWARE REUSE

(1) Identify anti-patterns ("7 deadly sins") and provide improved solutions

1. A junior developer is tasked with parsing a guestbook and outputting all greetings. At the end of the business day he proudly presents you his solution:

```
1 int start = xml.indexOf("<content>") + "<content>".length();
2 int end = xml.indexOf("</content>");
3 String content = xml.substring(start, end);
```

problem → tries to parse XML on his own · narrow-mindedness · ignorance

```
1 SAXBuilder builder = new SAXBuilder(false);
2 Document doc = builder.build(new StringReader(xml));
3 String content = doc.getRootElement().getChild("content").getText();
```

reusing this SAX Builder makes solution easier (parses automatically XML)

2. Browsing your company's code base you come across these lines:

```
1 File tmp = new File("C:\\Temp\\1.tmp");
2 File exp = new File("export-2013-02-01T12:30.txt");
3 File f = new File(path +'/'+ filename);
```

problem → paths are relative to the OS
· haste (not enough time)
· apathy

```
1 File tmp = File.createTempFile("myapp","tmp");
2 File exp = new File("export-2013-02-01_1230.txt");
3
4 File f = new File(path + File.separatorChar + filename);
5 // or even better
6 File dir = new File(path);
7 File f = new File(dir, filename);
```

use some OS path indep. implementation

3. An inexperienced programmer gives you a prototypical implementation that you have to improve. It contains this try-catch block:

```
1 Query q = ...
2 Person p;
3 try {
4   p = (Person) q.getSingleResult();
5 } catch(Exception e) {
6   p = null;
7 }
```

problem → you catch everything (no exception)
· sloth (too lazy to think about exception)

```
1 Query q = ...
2 Person p;
3 try {
4   p = (Person) q.getSingleResult();
5 } catch(NoResultException e) {
6   p = null;
7 } catch (...) {
8   //...
9 }
```

how to handle the NoResultException

4. A network application that you maintain sometimes crashes silently. You already spend hours trying to find the cause. Suddenly you spy these lines of code and want to kill the original author.

```
1 try {
2   ... do risky stuff ...
3 } catch(SomeException e) {
4   // never happens
5 }
6 ... do some more ...
```

```
1 try {
2   ... do risky stuff ...
3 } catch(SomeException e) {
4   // never happens hopefully
5   throw new IllegalStateException(e.getMessage(), e);
6     // crash early, passing all information
7 }
8 ... do some more ...
```

communicating that smt went wrong

problem → if something goes wrong, nothing happens · ignorance

5. Maintaining an e-commerce website you receive an angry call from a customer. He just sold 100 different chewing gums for € 0.30 each and the bill to the buyer totaled to only € 29.999971 instead of € 30. After inspecting the code you start to write a letter of apology to your customer.

```
1 float total = 0.0f;
2 for (OrderLine line : lines) {
3   total += line.price * line.count;
4 }
```

problem → calculation with floating points is always difficult and smt could go wrong · ignorance · haste

```
1 BigDecimal total = BigDecimal.ZERO;
2 for (OrderLine line : lines) {
3   BigDecimal price = new BigDecimal(line.price);
4   BigDecimal count = new BigDecimal(line.count);
5   total = total.add(price.multiply(count)); // BigDecimal is immutable!
6 }
7 total = total.setScale(2, RoundingMode.HALF_UP);
```

using BigDecimal allows precise count

(2) Software Reuse

1. SOFTWARE PRODUCT LINES

Mockia is a rising star in the highly competitive smartphone market. Their smartphone *MeiFon* is the first smartphone that allows almost full customization of the hardware configuration. Although the new concept already found wide reception and pre-orders by far exceed the immediate production capabilities, the CTO of Mockia raised concerns about the flexibility of their current development approach. Currently the entire software stack of Mockia's smartphones is custom built; reuse is, if at all, entirely done opportunistically (i.e. *ad-hoc reuse*). There is an enormous mass of configuration possibilities: a user can choose between 4 CPU models, 6 persistent memory and 3 different camera modules, and 2 GPS chips. A closer look at the pre-order data revealed that more than 92% of the customers pre-ordered a configuration that included the cheapest CPU and the second-smallest memory module. You are hired to propose a new development concept to make the software development more efficient. From previous projects you know that developing a new controller software for a CPU takes about 6 person months (PM), for a memory module 2PM, for a camera module 4PM, and for a GPS chip 2PM. Furthermore, your experience taught you that writing a SOLID platform that a-priori is designed for reuse typically yields 3 times the effort of the normal development effort of the respective components.

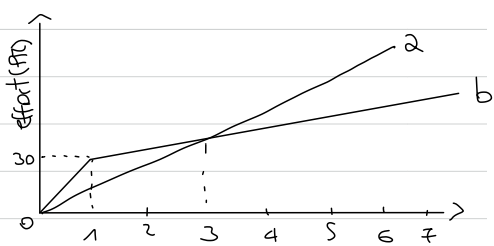
configs

Possible variants for configuration: $4 \cdot 6 \cdot 3 \cdot 2 = 144$ configs

$3 \cdot 2 = 6$

1. Core features: generic PLE core effort (for 92% used configuration) $\Rightarrow 3 \cdot (6PM(CPU) + 2PM(Memory)) = 24PM$
2. Variability points: CPU, Memory, GPS and Camera
   - reduce variability options for CPU memory-core
   - or
   - only allow the customer to select camera and GPS
3. BEP consider the cumulative effort:
   a. For single system deployment (i variants): $e_{single}(i) = i * (6PM + 2PM + 4PM + 2PM) + 0PM = 14PM \cdot i + 0PM$
   b. For the PLE case: $e_{PLE}(i) = i * (4PM + 2PM) + 24PM = 6PM \cdot i + 24PM$

initial effort



reuse comes into play

⇒ from generation 3 onwards, it's economically more convenient to use PLE

⇒ for a small business (selling 1-2 variants), single system development is better

2. LIBRARIES vs. FRAMEWORKS

- **Framework:** Provides a semi-complete (domain-specific) application that is often in charge of running the system. You can typically extend the frameworks through dedicated interfaces and use components to compose your own application in a prescriptive way. *
- **Library:** The developer is in charge of running a system and embedding the library as a pluggable component for performing problem-specific tasks. Thus, the library does not make assumptions nor has it implications on how the system built around it should look like.

* ex. in Spring Boot framework, we define @Spring annotation in main class and it triggers the framework to take control of the application

Potential Problems with frameworks:

- cannot be (de) composed
- are dogmatic / opinionated
- are complex and hide the magic
- try to solve everything at once
- shackle you

(3) TypeScript (TS) => superset of Java Script (JS) that can highlight unexpected behavior in your code, lowering the chance of bugs

ex.
```
interface User {
  name : string;
  id : number;
};
```

```
const user : User = {
  username : "Hayes",
  id : 0,
};
```

TS warns you that you have provided an object that doesn't match the interface

Type '{username : string; id : number;}' is not assignable to type 'User'

Angular => JS framework : flexible front-end development (content is loaded from API and displayed through FE)

↳ Angular is a platform and framework for building single-page client applications using HTML (it is written in TS / it implements core and optional functionality as a set of TS libraries imported into your app)

Simple file that starts a large process (source files that loads helpers/model/configuration and passes control off to controllers)

· An Angular app is defined by a set of NgModules (1 for bootstrapping + other feature modules) where are contained COMPONENTS:

- components define VIEWS (sets of screen elements that Angular can choose among and modify according to a program's logic and data)
- " " use SERVICES (provide specific functionality not directly related to views)
  ↳ service providers can be injected into components as DEPENDENCIES

↳ they are classes that use DECORATORS (which provide metadata that tells Angular how to use them):
- metadata for a component class, associates it with a template* that defines a view
- metadata for a service class provides the information Angular needs to make it available to components through dependency injection

* a template combines ordinary HTML with Angular directives & binding markup that allow Angular to modify the HTML before rendering it for display

=> Angular provides a ROUTER service to help you define navigation path among views that are arranged hierarchically
and is modeled on the familiar browser navigation conventions:
- enter a URL in the address bar and the browser navigates to a corresponding page
- click links on the page and the browser navigates to a new page
- click the browser's back and forward buttons and the browser navigates bkf through the history of pages you've seen

{{project.name}} => double curly brackets are in html a ONE WAY BINDING, which means that we can write TS code inside (project.name). Whatever is outside of the controller will be showed inside the view

⇒ Together, a component and a template define an Angular view :

- a decorator on a component class adds the metadata, including a pointer to the associated template
- directives and binding markup in a component's template modify views based on program data and logic

---

A template looks like regular HTML, except that it also contains Angular syntax such as :

1. data binding ⇒ coordinate app and DOM data

ex. ③ `<input [(ngModel)] = "hero.name">` → two-way data binding (combines property and event binding): a data property value flows to the input box from the component as with property binding* the user's changes also flow back to the component, resetting the property to the latest value, as with event binding

2. pipes ⇒ transform data before it is displayed (in HTML template: `{{ interpolated-value | pipe-name }}`)

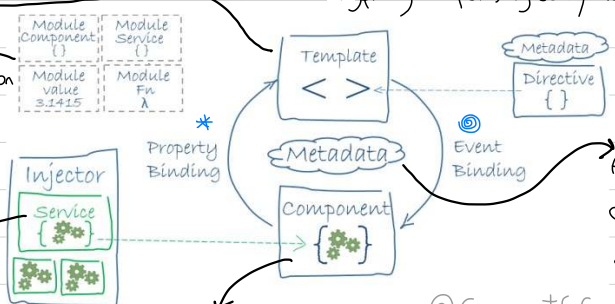
ex. `<p> The date is {{ today | date : 'fullDate' }} </p>` ⇒ output 'Monday, June 15, 2015'

3. directives ⇒ apply app logic to what gets displayed (give instructions in rendering Angular pipes to transform the DOM)

ex.1 `<li *ngFor = "let hero of heroes"></li>` ⇒ REPEATIVE : stamps one <li> per hero in the heroes list

ex.2 `<app-hero-detail *ngIf = "selectedHero"></app-hero-detail>` ⇒ CONDITIONAL : includes the Hero Detail only if a selected hero exists

ex.3 ⇒ ATTRIBUTE : ngModel directive (implements a 2way data binding) modifies the behavior of an existing element (typically <input>) by setting its display value property and responding to change events



angular loads as a collection of JS library modules

- install npm
- import {Component} from @angular/core

Associates a template with the component by inline code or by reference :

ex. src/app/hero-list.component.ts (class)

```
export class HeroListComponent implements OnInit {
    heroes : Hero[]
    selectedHero : Hero
*   constructor (private service : HeroService) { }
}
```

```
@Component ({
1.  selector    : 'app-hero-list'
2.  templateUrl : './hero-list.component.html'
3.  providers   : [ Hero Service ]
    })
export class HeroListComponent implement OnInit {
    /* ... */
}
```

1. If an app's HTML contains the tags <app-hero-list> then Angular inserts an instance of HeroListComponent between the tags
2. the module-relative address of this component's HTML template (⇒ template defines the component's host view)
3. array of providers for services that the component requires

ex. src/app/hero.service.ts (class)

```
export class HeroService {
    private heroes : Hero[] = [];
*   constructor (
        private backend : BackendService,
        private logger : Logger ) { }
}
```

* when Angular creates a new instance of a component class, it determines which services/dependencies from constructor

EX. 8 SOFTWARE TESTABILITY & TESTING

(1) Testability

1) Factors influencing testability:

- - OBSERVABILITY => degree to which it is possible to observe the system state
  - CONTROLLABILITY => " " control the state of the system
- - COUPLING => low coupling means less communication (more separated components), so the testing is easier as it doesn't need to worry about interactions
  - SEPARATION OF CONCERNS (SRP) => more obvious which functionality to test
- - A flaky test is a test which could fail or pass for the same configuration
  - ↳ such behavior could be harmful for developers because test failures do not always indicate bugs in the code. It is particularly prone to affect tests with a broad scope (ex. functional/UI tests)
  - Cause of non-deterministic behavior of the system during testing:
    - Setup/Teadown
    - Caching
    - Infrastructure Issues
    - Concurrency
    - Dynamic Content
    - External Systems

2) Approach for assessing testability

- Domain/range ratio (DRR): if |input domain| >> |output domain|, then testability is poor (high size) because many things look the same even though there are many input values

(less observability)

3) Test selection criteria

- Code Coverage: if a test covers many lines of code, that doesn't correlate with detection likelihood (it tells what you haven't tested but not what you did cover)

(2) Convention for writing test (Arrange-Act-Assert or Given-When-Then)

↳ devide test in 3 parts:

1. Set up test data and define expected outcomes of test
2. Triggering an action on the system under test (call a function that returns actual outcomes)
3. Asserting that actual outcome = expected outcome

Code Coverage => to inspect how much code was tested you can use the JaCoCo Maven plugin and inspect the index.html

```
13.     public static ArrayList<Integer> findFactors(long input) {
14.         ArrayList<Integer> factors = new ArrayList<Integer>();
15.         int i = 1;
16.         while (i <= input) {
17.             if (input % i == 0) {
18.                 factors.add(i);
19.             }
20.             i++;
21.         }
22.         return factors;
23.     }
```

Figure 1: Code coverage as displayed by JaCoCo

1. Even though there's full coverage, the method findFactors() can receive a input number that is bigger than the maximum range of integers and be automatically converted into a negative number that would return no factors

2. To write better test for this critical situation, we would need to write more test that checks negative numbers and numbers that are greater than max (integer size)

Full Adder method

↓ test all 8 combinations

```
1 public static Boolean[] fullAdder(Boolean a, Boolean b, Boolean c) {
2   Boolean s, tmp;
3   tmp = a != b;
4   s = c != tmp;
5   c = (a && b) || (tmp && c);
6   return new Boolean[]{s, c};
7 }
```

```
@Test
public void testFullAdder() {
    assertArrayEquals(SuperMath.fullAdder( a: false,  b: false,  c: false), new Boolean[]{false, false});
    assertArrayEquals(SuperMath.fullAdder( a: false,  b: false,  c: true), new Boolean[]{true, false});
    assertArrayEquals(SuperMath.fullAdder( a: false,  b: true,  c: false), new Boolean[]{true, false});
    assertArrayEquals(SuperMath.fullAdder( a: false,  b: true,  c: true), new Boolean[]{false, true});
    assertArrayEquals(SuperMath.fullAdder( a: true,  b: false,  c: false), new Boolean[]{true, false});
    assertArrayEquals(SuperMath.fullAdder( a: true,  b: false,  c: true), new Boolean[]{false, true});
    assertArrayEquals(SuperMath.fullAdder( a: true,  b: true,  c: false), new Boolean[]{false, true});
    assertArrayEquals(SuperMath.fullAdder( a: true,  b: true,  c: true), new Boolean[]{true, true});
}
```

Inputs			Outputs	
A	B	C—IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Binary Addiction Method

```
1 public static Boolean[] BinaryAddition(Boolean[] a, Boolean[] b) throws
      ArrayLengthsMismatchException {
2   if (a.length != b.length) {
3     throw new SuperMath.ArrayLengthsMismatchException();
4   }
5   Boolean[] result = new Boolean[a.length + 1];
6   Boolean carry = false;
7   int i;
8   for (i = 0; i < a.length; i++) {
9     Boolean[] tmp = fullAdder(a[i], b[i], carry);
10    result[i] = tmp[0];
11    carry = tmp[1];
12  }
13  result[i] = carry;
14
15  return result;
16 }
```

1. It can't be completely tested because the can be input of infinite length arrays
2. We need to write a test that checks that exception (different length of 2 arrays)

```
@Test
public void testBinaryAddition_throwsArrayLengthMismatchException() {
    //given
    Boolean[] a = new Boolean[]{false,false,false};
    Boolean[] b = new Boolean[]{false,false};

    //when
    try{
        Boolean [] res = SuperMath.BinaryAddition(a, b);
        fail("Exception expected");  //if we don't get immediately into the catch, it fails
    } catch (Exception e) {
        //then (asserting that the exception thrown is of the type ArrayLengthMismatch
        assertThat(e, instanceOf(SuperMath.ArrayLengthsMismatchException.class));
    }
}
```

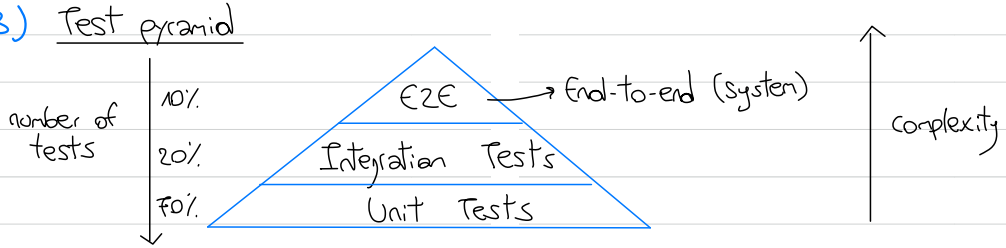
- Handler Code added to TestStarterApp

```java
public static void someCaller(Boolean[] a, Boolean[] b) {
  Boolean[] res = null;
  try {
    // give it a try
    res = SuperMath.BinaryAddition(a, b);
  } catch (SuperMath.ArrayLengthsMismatchException e) {
    // fail gracefully
    res = new Boolean[]{false, false, false};
  }
  System.out.print(Arrays.toString(res));
}
```

```java
@Test
public void testSomeCaller_handlesException() {
    //given
    Boolean[] a = new Boolean[]{true,false,false};
    Boolean[] b = new Boolean[]{false,true};
    Boolean[] expectedResult = new Boolean[]{false,false, false};
    //capture output
    final ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    //when
    try{
        TestStarterApp.someCaller(a,b);
        //then
        assertEquals(Arrays.toString(expectedResult), out.toString());
    } catch (Exception e) {
        //if exception still occur we fail
        fail();
    }
```

(3) Test pyramid



1. Unit Test => test if a controller method for retrieving a project by its id will return an empty project, if called with a project id that doesn't match any of the projects in the database

2. Integration Test => test if a JPA repository responsible for querying projects from the database will do so correctly when it is not queried with an id, but with the project name as identifier.

3. System Test => test if an up-and-running Spring Boot application that receives an HTTP request to GET a specific project by its id will return an empty JSON response to the requester in case a project with this id is not found in the database

9. MODEL-BASED ENGINEERING

↳ application of modeling to support system (req., analysis, design,...) beginning in the conceptual design phase and continuing throughout development and later life cycle phases

MODEL : appropriate abstractions for many different particular purposes in different lifecycle dev. phases

- encapsulation : without loss of information, it's hidden (functions, libraries,...) => generation of code possible
- information deliberately missing, complex communication abstracts into just one message => generation of skeleton code

[code itself is a model : doesn't reflect time and memory consumption]

SYSTEM MODELLING :
- Operational Context
- Interfaces
- Behavior at the border of interface
- Inner structure of artefacts

} Artefacts that can be represented by models

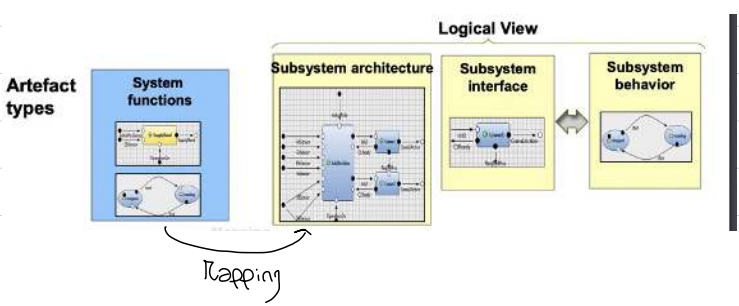
MBSE principles :
- consideration of different viewpoints for differentiation and structuring of the transition from problem analysis to solution
- explicit distinction between levels of granularity of system description according decomposition
- artifact-model including defined semantics of artifacts and their relationship

Approaches
- Projective vs. Synthetic
- Code-based (ex. C, C++) vs. Model-based Development (Requirement Models ↓ Application Models ↓ Platform Models)

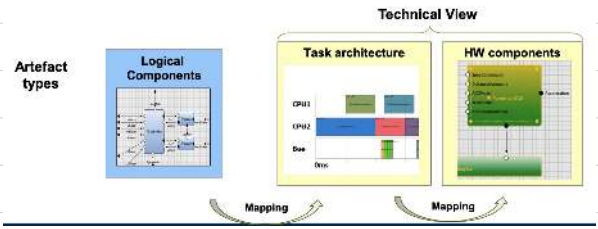
=> Viewpoints [give structure to SE to abstract, analyse and understand better]

- LOGICAL : captures the decomposition of the system in terms of logical components
  - Reactive Behavior (ex. State Transition Diagram / Code Spec. / Message Sequence Charts)
  - Computational Behavior (ex. Data flow Diagrams)

ex. Pipes and filter architecture



- TECHNICAL : (modeling goals)
  - Hardware Architecture (components / ports)
  - Software Architecture (Dynamic: Run-time Server / Static: Design-Time Software)
  - Deployment Logic (used to map components from logical to technical architecture)



- PRO of MBSE => Separation of concerns / Model in the Loop / Understandability / Re-Usability / CodeGeneration / Analysis
  - in Document-based Engineering the main problem is syncronization (concern also in MBSE)
  - Safety Analysis => they are performed at different levels of abstraction during {concept phase / development}
  - "The richer the models, the more analyses are possible"
    - Simulation / Model Checking

ex. MB Analyses => Non-determinism Check : given a state automation, is it well defined for each state and combination of inputs (for transitions)?

- Summary:
  1. An ISO 26262-conforming component model must model the functional, logical and (SW/HW) technical aspect of a system, including the corresponding functional-logical & logical technical deployments
  2. Can be exploited in modular development for discharging assumptions and analysing impacts of modification
  3. Perform safety analysis

10. INFORMATION SECURITY

↳ How a person can affect (negatively) a products that affect othe Users

{ Security => protection against external hazards
Safety => protection against hazards originating from the operation of a system (environment)

· Security Properties : Confidentiality / Integrity / Availability [+ Authenticity, Availability, Anonymity, ...]

↳ implementation { by design => minimal attack surface, logging and audit infrastructure, ...
by mechanism => access control mechanisms, digital signatures, message authentication codes, ...

· Security Problem :

- "program is broken into" => the vulnerability of a software is exploited by hacker
↳ data and commands are not separated from each other
↳ system is misconfigured [firewall not working properly]

when we configure a system there's the need to follow heuristics to achieve a good level of security
↳ the only way is to tackle the security issue at every step of the software development

· DESIGN PRINCIPLES (that help to implement security in a system)

- Least privilege => every subject should not have more privileges than necessary to complete the job
  - minimizes negative consequences of inadvertent operating errors
  - reduces negative effects of deliberate attacks carried out by a subject

  ex. a web-server's processes do not need to run with administrative privileges

- Complete mediation => access to every object must be controlled in a way not circumventable

  ex. airport ensure that every subject is checked before entering sensitive areas

- Secure, fail-safe defaults => security mechanism should start in a secure state and return to a secure default in case of failures

  ex. firewalls often use white-list rather than black lists (default is to deny any network packet not matching white-list rules)

- Compartmentalization => organise ressources into groups isolated from others except from controlled communication
  ↑ making sure the corporised part don't infect others
  ex. partition network in separate zones

- Minimum exposure => minimize the "attack surface" a system presents to the potential adversary
  ex. harden OS by disabling all unneeded functionality

⟹ provide high level guidelines for analyzing mechanism and trade-offs (≠ patterns)
concrete implementation-oriented designs ↑

- IMPLEMENTATION LEVEL CONCERNS
  - Broken Authentication
  - Sensitive Data Exposure
  - XML External Entities => XML processors evaluate external entities that may not be protected
  - Broken Access Control
  - Security Misconfiguration => result of insecure default configuration, open cloud storage,...
  - Insufficient Logging & Monitoring

- Cross site scripting (XSS) => XSS allows attackers to execute scripts in the victim's browser site which can hijack user-sessions, deface web sites or redirect user to malicious

  i. Reflective XSS => client tricked into contacting a server (clicking on a mail URL) that has an Action Name that is executed not on the server but on client side that may steel cookies
  with e.g. a JS code that create an HTML file

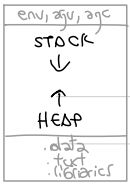
  ii. Persistent Attacks => make the server store "bad code" that is executed whenever client accesses the page
  ↳ you can try to filter Java code (that can be encoded in many malicious ways)

- Injection Attacks => providing data which will be misinterpreted as code (commands)

  i. SQL injections

  If a server receives a post request of a $MAIL and calls the database via
  SELECT fieldlist FROM table WHERE field = 'anything OR 'x' = 'x' , we get a true evaluation of $MAIL field and complete access to the db is granted

ii. Buffer overflows => when functions are called, frames are pushed to the runtime stack and popped when the function returns



the stack usually grows "in the wrong direction" from high addresses to lower addresses

ex. we have space only for 50 elements but try to store 80, so we try to override 30 elements from heap to stack and make the system crash
↳ even worse: overwrite return address with address of the stack that contains malicious code (usually opening a shell inside the program)

Defense
- => since the problem is an underspecification in these situations on C language, it's less likely to happen if we use different languages such as Java
- => use fuzzers that apply random input to a program to test if it crashes or not
- => perform security reviews for systems and code

- SECURITY ANALYSIS
  - Attack Trees => think about which steps to take to obtain a top level goal
    - prioritize nodes with special needs by assigning attribute to nodes { possibilities / probabilities / estimated impact } to figure out possible / cheap / easy attacks
      * policy decision points
    - the Web Browser request data from the server and transmits a policy along with the data that shows up whenever we try to consume data at the PDP*
      - browser side: a plugin called BRUCE receives policy and forwards PDPs on different levels
      - server side: the SCUTA component makes sure the policy is shipped to the browser at data
    - enables to think about an attack and your system in a structure way
  - Baseline Protection (BSI) [or "IT-Grundschutz"] => perform a first security analysis & apply standard safeguards for std. problems (for high risk, perform a dedicated risk analysis)

- DATA PROTECTION (GDPR: General Data Protection Regulation)
  - Since 2018, regulates privacy in EU: there needs to be a consent when collecting data
  - There can be fines up to 4% of revenue for non-compliance to these rules
    ↳ compliance can be proved through Paperwork / Technical Solutions and Process Automation / Leveraging personal data

11. DISTRIBUTED SYSTEMS & MIDDLEWARE

↳ physically disjoint compute resources, interconnected by a network (the failure of a computer can make the network unusable)

- Characteristics of DS => reliability, availability, heterogeneity, openness, security, scalability, failure handling, transparency
  - Reliability : system's working well (=> uptime/(uptime/downtime) → 99,99% : 8.64s/day of downtime
  - Availability : being able to connect to the server
  - Transparency :
    - Location => hide where a resource is located
    - Migration => hide that a resource may move to another location
    - Replication => hide that a resource is replicated (backup for avoiding observable failure)

- MIDDLEWARE => services and abstractions that facilitate the design development, and deployment of distributed applications in heterogeneous, networked environments
  ex. remote invocation, messaging, RPC monitoring, locking service
  - ↳ deals with interoperability (ex. between Java code and C code)
  - ↳ provides and abstraction of hardware to be able to be accessed by application

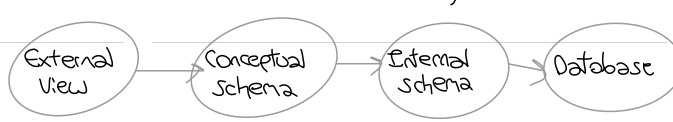
(1) DATABASE CENTRIC ARCHITECTURE

- ↳ main purpose : data access and update
  - traditional db (passive) : responds to requests
  - blackboard system (active) : clients solve problems collaboratively and system updates clients when information changes
- ↳ Client server evolution :
  - Mainframe { Early single-user, Batch processing, Time-shared }
  - Personal Computers { PCs, Networked PCs }
  - Cloud / Grid / Edge Computing

- MAINFRAME => uses the mainframe as a central repository for information as well as processing for every appl.

- 3 LAYERED CLIENT/SERVER ARCHITECTURE
  - Client => Browser
  - Web server { Presentation layer (UI components), Business layer, Data layer }

- DB-centric Architecture



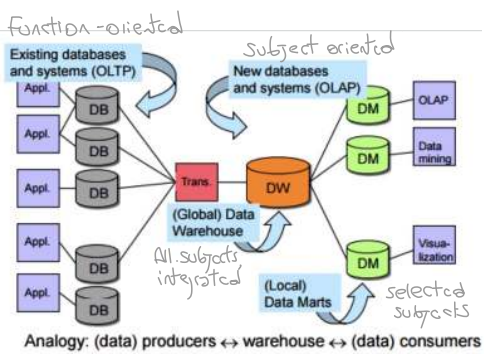
ex. CREATE VIEW " "
SELECT " "
FROM " "
GROUP BY " "

- VIEW => dynamic result of 1 or more relational operations → operating on the base relations to produce another relation
  - queries can be reused
  - → data accessed in a customized way / flexible security mechanism / simplify complex operations

- SUBPROGRAMS => named PL/SQL blocks that can take parameters and be invoked
  - STORED-PROCEDURES => stored in the DB (ex. SQL > exec remove_employees(10))
  - FUNCTIONS => always return a value

i. Data Warehousing and business intelligence

- DATA WAREHOUSING : collection of methods / techniques used to support knowledge workers to conduct data analyses that help with decision making / info resources
  - => Data Analysis Problems : heterogeneous sources / data suited for OS / bad data quality / volatility
  - → DW features : subject oriented / integrated and consistent / shows evolution over time



DATA MART (DM) => is a subset or an aggregation of the data stored to a primary data warehouse

- OLTP (Online Transactional Processing) => Top down
- DW (Data Warehouse) => In-between
- OLAP (Online Analytical Processing) => Bottom-up (+fine grained)

- OLAP data cube => Data analysis tool generalised by GROUP BY queries that aggregate facts based on chosen dimensions
  - → good for visualization / multi-dimensional / support interactive OLAP operations [Slice/Dice/Drill/Aggregation]

- ETL => getting multidimensional data into the DW [Extract / Transformations / Load data to DW]
  - → semantically integrating different data sources

- DW Architectures

1. Central DW => all data in one, central DW
2. Federated DW => data stored in separated data marts
3. Tiered => data is distributed to data marts in one or more tiers
   ↳ data is aggregated/reduced as it moves through tiers

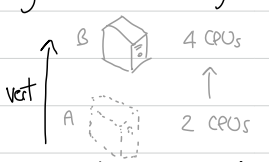
Challenges of DW: Security of data during ETL / Data visualization / Decision Analysis (what-if analysis)

ii. Big Data architectures

- BIG DATA => dataset whose size is beyond the abilities (store, manage, ...) of typical database software tools

(1) SCALABILITY => ability of a system, network or process to handle a growing amount of work

- Vertical scaling => when an existing IT resource is replaced by another with higher (scaling up) or lower (scaling down) capacity
  - PRO: easy to design for vertical scaling
  - CONS: limited by maximum hardware capacity



- Horizontal scaling => allocating IT resources that are of same type (scaling out) or releasing of res. (scaling in)
  - ↳ common within cloud environments
  - PRO: less expensive, IT resources instantly available, resource replication and automated scaling
  - CONS: additional IT resources needed

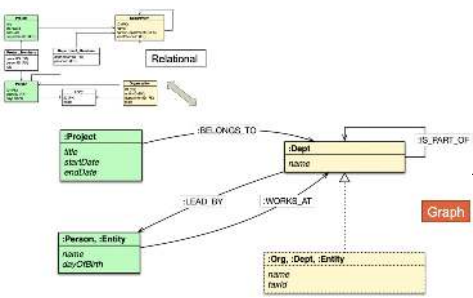
- Sharding => divide the data store into horizontal partitions (shards) that hold the same schema but holds its own distinct subset of the data
  - ↳ a shard is a data store in its own right, running on a server acting as a storage node

=> same schema, each shard placed on a separate node in a cluster, better cache locality, central lookup registry to map requests to shards

- Lookup strategy => map that routes a request for data to the shard that contains that data by using the shard key
- Range strategy => groups related items together in the same shard and orders them by shard key (keys are sequential)
- Hash strategy => uniform distribution of data across the shards (reduce the chance of hotspots in the data)

(2) NoSQL data stores => next generation dbs mostly addressing: being non-relational, distributed, open source and horizontally scalable

1. KEY-VALUE STORES => data is stored in unstructured records consisting of a key + the values associated with that record
   - ex. Redis => in-memory db that works in-memory (+ very fast, - little space: computer RAM)
2. BIGTABLE => distributed storage system for managing structured data that is designed to scale to large size
   - horizontal sharding that works very well for non-complex queries that don't use joins
3. DOCUMENT-ORIENTED DATABASES => each record and its associated data is thought of as a "document"
   - ex. Mongo-DB => a row is a tuple of cells that corresponds to a column (encapsulation)
   (+ performance / + writability / + storability)
4. GRAPH DATABASES => rather that computing relationships at run-time, they are stored in a graph



=> easier to retrieve the information because it's already stored and doesn't need to be computed over and over

NoSQL:
- data is generally duplicated
- no standardized schema
- no std format for queries
- no std. query language

(3) DATA PROCESSING => split db into sub-class, work on the parts separately (PARTITION) + and combine results (COMBINE)

MAP REDUCE:
- iterate over a large number of records
- extract smt of interest from each
- shuffle and sort intermidiate results
- aggregate intermidiate results
- generate final output