The software design process

Paolo Burgio
paolo.burgio@unimore.it

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

High Performance Real Time Lab

"Weeks of coding can save you hours of planning." - Unknown

Why do we do this?

For the present

- Enable collaborative development (collaborative tools)
- In > months of dev, the team might change! (documentation, both internal and external)
- Automate testing and releasing

Ultimately, every team member should focus on few tasks, the ones that fit him/her better!

For the future

- Make the mantainance process easier
- Enable future extensions/development

The Apollo 13

"We need find a way to put this, in the hole for this, using these"

- https://www.youtube.com/watch?v=ry55--J4_VQ

Software engineering

The discipline of building big, complex systems

- Applies methodologies from the engineering world, to software development process
- The only way of dealing with complex software systems and teams
- A systhematic approach to design, development, testing, deploy and maintenance (IEEE 1990)

...don't worry... ☺

- This won't make of you an engineer
- It will help you engineering software

Think today, for tomorrow

Engineering:

- Have a strong focus on the process
- Treat everything as "a resource", either physical (a brick for a wall, a chip for a server farm) or non-physical (software artifacts, licenses, etc)
- In years, they (..we... ☺) developed an common methodology for multiple application areas

Software, on the contrary, is (correctly) treated as a non-physical entity, "a product of the mind"

- During the development, you care less of physical assets (mostly, computers, desks...)
- *"Sul mio computer funziona"*
- When you design SW, you care more of people and their skills

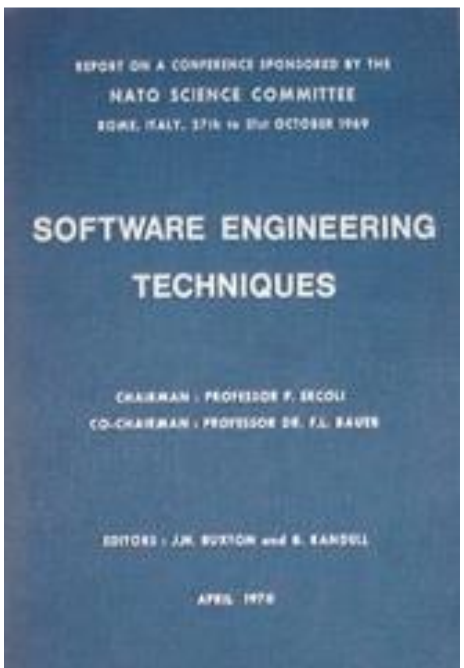
Adapt this for future complex SW!

- Future systems will be distributed CPSs, made with different computers (high-performance, energy efficient, etc..) with tight(est) interaction with the world
- Will be large-scale 24/7 distributed systems, updated over-the-air
- Designed today, thought for tomorrow (e.g., Software-Defined Vehicles)

A bit of history...

SW engineering was born in 1968, at the NATO conference, focusing on

- software crisis
- software reuse
- software engineering



In the nineties...

- ....the era of object oriented programming
- Design tools (UML), and patterns

In the last decade(s)

Internet-of-Things

- Large scale cloud projects
- Ubiquity, pervasiveness, CPSs
- Massively parallel computers (GPGPUs)

The era of machine learning

- How can you structure a probabilistic-based SW component?
- Design of the training process
- Data engineering!

Example - The Waterfall model

(aka: where do we all come from)

Analysis of the requirements and spec, identifying KPIs

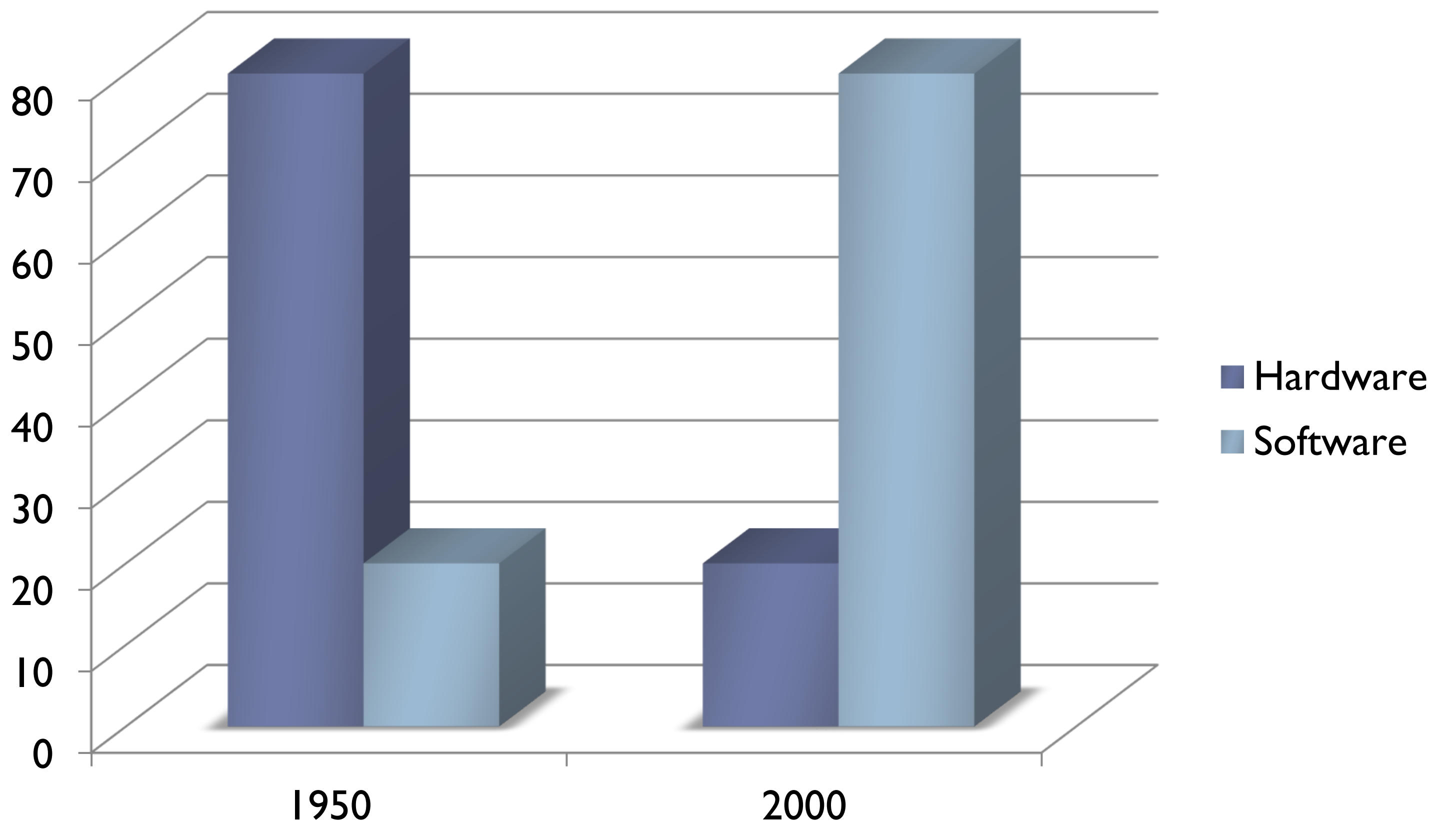
System design

Development and unit testing

Integration and system-wide testing

Delivery and maintenance

Costs: HW vs SW



Who is our customer?

For whom it is developed?

- (This includes also open source, and free software)
- We will generically speak of ***customer***, without loss of generality

Custom solutions, tailored for specific customer, or customer segment

- Time-to-market agreed with the customer
- Internal R&D projects (e.g., iotty, Hipert “Fire”)
- Market-wide products (E.g., MS Windows)

General-purpose software, released for the masses

- For research (e.g., our F1tenth stack, tkDNN)
- Business models based on open-source (Home Assistant, GNU/Linux, Erika..)

Quality of software, and process

Of product / extrinsics / external

- Refers to the functionality, it's the main quality against which software is assessed!
- By the customer / segment / community / domain
- Has directly reflect on co$t pricing
- Assessed by functional requirements

Of process / intrinsics / internal

- Refer to the process, i.e., how the software is developed
- Relates mainly to the specific domain, or company/team (hence, even more important!)
- Hard to map onto product pricing
- Reflects into non-functional requirements

Assessing the quality / functional properties

Correctness

- *Does my software do what I want it to do?"*
- The easy part: captured by functional requirements, which are directly negotiated with the customer

Ease-of-use

- Involves UX, documentation...

Performance/efficiency

- *"Is it fast?"*
- What do "fast" mean? FPS? E2E latency? On which computer?
- How many resources does it need (e..g, power/Watts? Enery/Joule?, physical space)?

Dependability

- *"Can I rely on it?"*
- Definition applies to specific fields...

On dependability

“a measure of a system’s availability, reliability, maintainability, and in some cases, other characteristics such as durability, safety and security. In real-time computing, dependability is the ability to provide services that can be trusted within a time-period. The service guarantees must hold even when the system is subject to attacks or natural failures.”

Specific of application domains (can overlap!!!)

- Real-Time systems
- Embedded systems
- Exascale systems
- …

The IFIP working group identified three main elements

- **Attributes**, i.e., Key Performance Indicators (KPIs) to assess the dependability
- **Threats** to dependability
- **Means** to increase dependability

Non-functional properties

A system is

- **Verifyable** if we can assess its characteristics (what about ML?)
- **Mantainable**, if we can easily modify it (docs are in order??)
- **Reusable**, if it's well packed for deployment (docker?)
- **Portable**, if we get the same functionality on different HW/OS/…. (and also performance!!!! – see GPGPUs)
- **Interoperable**, if it's open for interaction with other systems…
- …

Basic design principles
~~of software engineering~~

Our friends

- Strictness, formalism
- Separation of concern
- Modularity
- Right level of abstraction
- Resiliency / robustness

SW best practices (we'll see them later)

- Design patterns (architectural, coding)
- SOLID principles
- ~~Tools and methodologies~~ Methodologies and tools

Strictness, formalism

Still, we are artists! Software is a piece of art!

- GPL licenses apply also to books, paintings...

But...we need to systhematize the process

- Typically, we borrow from mathematics / logics / engineering ☺
- ...to create well-known schemes...
- ...and leave the artist programmer the freedom to improvise within them

(Like a painter with a frame)

Separation of concern

Divide-et-impera

- Split the problem onto subproblems

..but, which problem?

- Lifecycle (Waterfall vs. Agile/Scrum)
- System architecture (e.g., Microservices)
- Internal system architecture MVC – MVVM
- Testing and deployment (CI/CD)

Modularity

Comes directly from the separation of concern principle, affects system design

Two main approaches

<u>Top-down</u>

- Where we have a complete view of the project, and we split it into components
- When we typically start from scratch

<u>Bottom-up</u>

- where we first develop the components, and then integrate them
  - A typical scenario is when we need to re-use existing modules

Abstraction

Example: how shall we model a user?

- In a gym club: name, age, weight, height, gender, email
- In the City Servers: name, age, address, CF, phone nr
- In a smart city roundabout: Lat, Long, velocity, class (car, bike, pedestrian)

What about costs?

Software costs outperform all other “structural” costs

- Licensing for libraries
- Fees for platforms (who has in-house servers anymore??)
- Electricity/heating/cooling

Maintenance is the main component

- A bad design is costly on the long term (see Apple’s)

Personnel costs

- Developers (80%)
- Support/aftermarket

Other costs

- HR, generic costs (chairs, laptops..)

Maintenance

The need for modifying the system after it has been deployed

- Bugfixing (typically for free in 12-24 months) – Functional testing with customer is really important to mitigate this – 20%
- Performance improvement – 60%
- Changes in the operational domain (e.g., new version of libraries, OS, hardware) – 20%

But most of all..

- Customer **never** knows what they want
- ...l'appetito vien mangiando ☺ - customer might ask modifications, even paying them in advance!

Often, more than 50% of the overall costs!

- 75% (Hewlett-Packard)
- 70% (US Defense)

Main issues with SW, today

Aka: “the generational debt”

- Old, legacy systems, developed with obsolete technologies, which cannot be replaced due to bad engineering practices
- The “Comune di XXX” example
- “Big bang” migrations/updates vs (take longer) step-by-step migrations

Systems are increasingly complex and heterogeneous

- The rise of micro-services architectural pattern

Time-to-Market

- Rush, rush rush!
- Tackled with agile methodologies

Professionality, and ethics

Sw developers shall always keep an ethic and professional conduct of work

- What does “ethic” mean?
- E.g., Hipert srl doesn't, and will never do, produce weapons

Confidentiality

- Often, you need to sign an NDA – Non disclosure agreement, before working
- After resigning a contract, some pros might not be hired by other competitor companies for 1-2 years!!!
- Example: Maserati SpA

Intellectual Property and licensing

- How much do you know about licenses/patents?

Structuring ethics

- Personal ethics
- Company rules (see Hipert)
- Professional
  - see “Ordine degli ingegneri”
  - Association for Computer Machinery (ACM) ed Institute of Electrical and Electronics Engineers (IEEE) - http://www.acm.org/about/se-code

Structuring the process

Modeling the process

We need to structure the entire development flow

There are multiple chances, depending on

- Type of technologies - *Might fit or not fit a methodology*
- The time we have for producing it - *Agile vs. more "traditional" methodologies*
- Legacy codebase - *Migrating/updating, or re-implement from scratch?*
- Company processes - *Use the right tool for the right processes*
- Standard processes for specific domains – *The V process in automotive*

A typical pattern

1. Specs
2. Design
3. Implement (SW and HW)
4. Integration
5. Test
6. Deploy
7. Maintenance/Aftermarket (AM)

Methodologies mainly differ in

- *How much time we dedicate in every phase?*
- *How early do we want to test/integrate?*
- *Shall we iterate on the main process? Or provide sub-processes?*
- *How early/late do we involve the customer? (e.g. graphic interfaces L&F)*
- *Do we need to standardize?*
- *Do we have legacy codebase/constraints on the tech to use?*

Typical choices / mistakes

1. Specs
2. Design

MISTAKE: mixing them

3. Implement (SW and HW)

BEST PRACTICE: tight overlapping

MISTAKE: underestimating documentation

4. Integration
5. Test
6. Deploy

BEST PRACTICE: can automate this!!!

7. Maintenance/Aftermarket (AM)

MISTAKE: undersestimate its cost

1) Requirements and specifications

Involve the commitments, and flood them with questions

- On what the system should do (functional) / how it interacts with user (non-functional)
- On the environment / ecosystem (hardware vs. software)
- Any legacy codebase? Any legacy process? Any legacy vendor?
- Try to imagine possible issues (for maintenance) and possible future steps (for aftermarket)

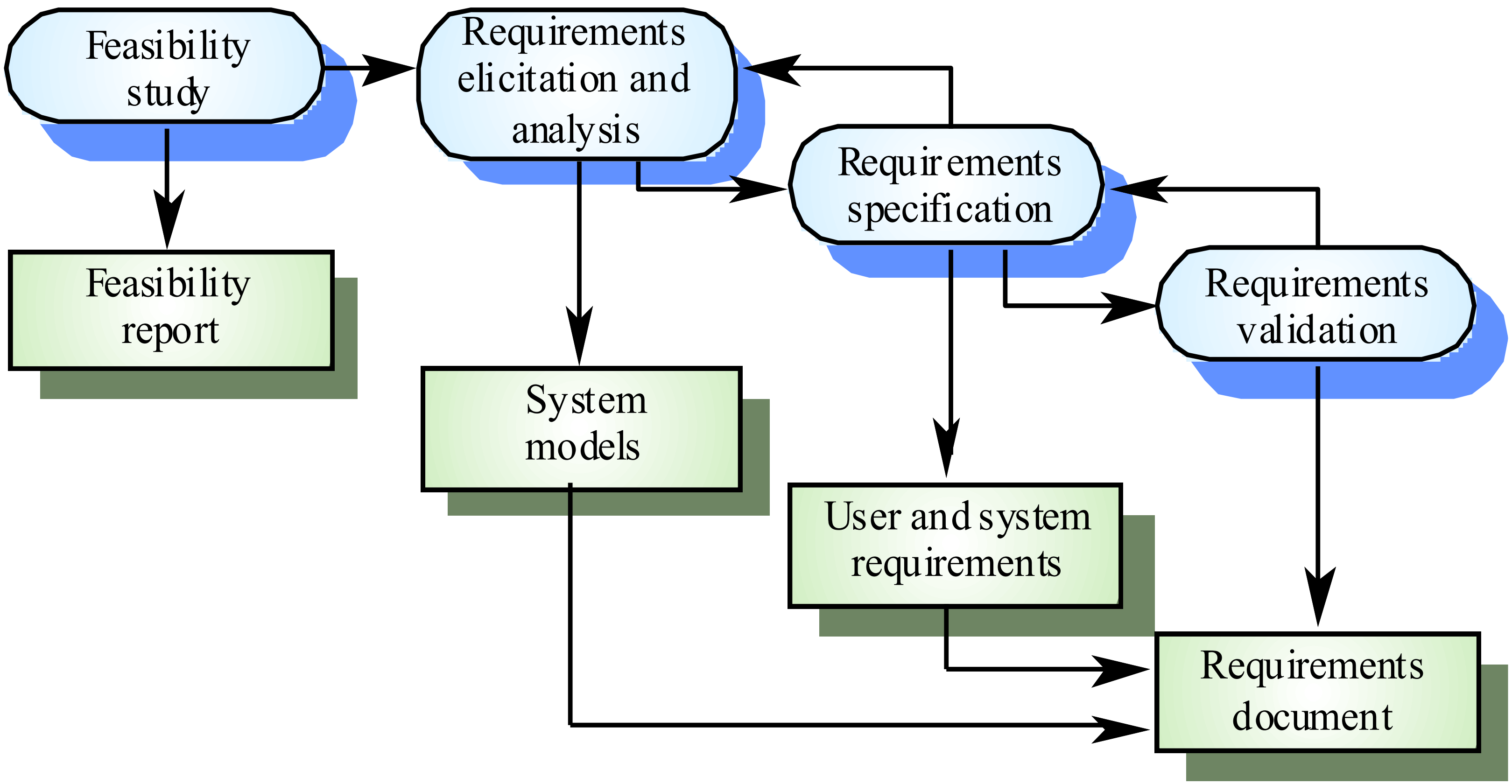
Yearly expertise on this / on specific application domains help

- Often, specific non-dev people: *Business analysts, Performance engineers, Requirements engineers*

The output: a (set of) document(s) that clearly cover all of these topics

- Identifying measurable identifiers (KPIs) to assess whether the system is working properly
- Directly affects testing phase!!!
- This is what defines the **value** for our customer

Reqs and specs – cont'd



Wait another minute…

An agreement shall be signed with our customer

- Then, we set up / deal the price of our services / sw
- Clearly state what we will do and don't
- …also what the customer shall do
- E.g., who's in charge of maintaining the server infrastructure? Who's paying the possible licenses (yearly)?

At this stage, the customer will still be nice, and lovely…

- …but they will start to get anxious, because they want the product done

DON'T RUSH! (yet, try to be as fast as possible)

- Remember, we're in Italy: 95% of companies are PMI/SMEs, and Project Manager have limited expertise/culture of sw design
- They think that *"They have a friend that might also do that"*

This phase is the most important!!! <u>Here, you are making promises</u>!

- Typical scenario: in 6-12 months the customer says: *"Well, the system doesn't' work, unless you also do this, this, and also this. If the system doesn't work, of course I won't pay you"*

2) System design

Let's sketch a working system

- How quick do we want a working prototype?
- Are we starting from legacy code, or from scratch?
- Shall we also design functional tests, from KPIs?

At this point you must take some decisions

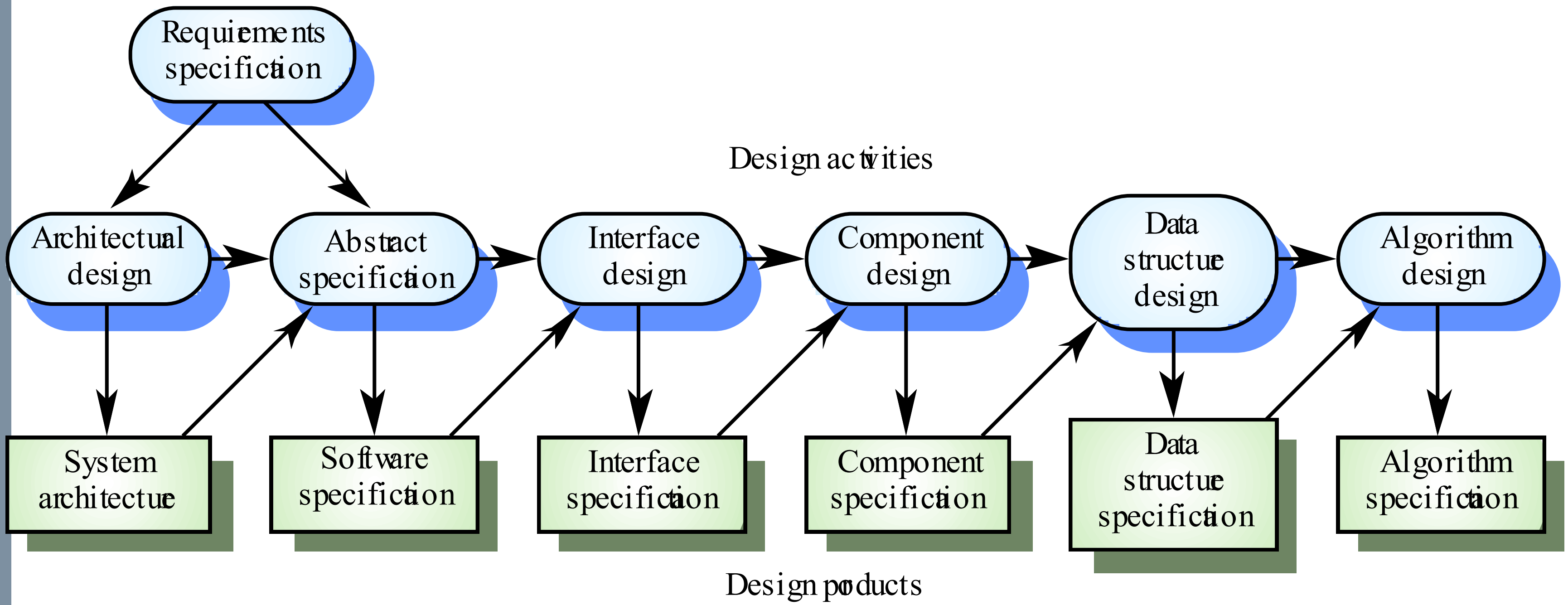
- Architectural design
- Choose the most appropriate language(s), framework(s)
- Choose the hardware
- Plan the integration / define protocols and comm infrastructure

What about the team?

- Which tools shall we use? Which methodology?
- Git (of course..?), Agile, Waterfall?

2) System design

- A typical high-level design



2) System design

At this point, we can start drawing how our system is actually made, not only what it does

- And the technology it uses

There are several tools / graphical models

- Unified Modeling Language / UML (a family of models)
  - For describing the behavior, and the way classes are done in OOP
- Data-Flow Diagrams / DFD
  - If data representation is our main concern
  - Pipeline-like architectures if we cannot store data / have too many data
- Entity-Relation (E-R) diagrams
  - Streamlines the development of DB models, and the main operations
  - E.g., Relational DBs are good

3) Implementation

After a good (even sub-optimal) design phase, implementation is streamlined!

Most of the applications we design today follow a very well-known paradigm

1. We need to store some data – aka: "CRUD"
   - Create, Read, Update, Delete
   - See HTTP(s) verbs
2. Process it somehow
3. And make it available to customers via an interface
   - Web? UX? REST service?

This means there are tons of tools and frameworks, to do so!

- In theory, no need to re-invent the wheel...
- ...unless the customer wants so

4) Integration

The process of re-uniting the different parts of our software

- Call them component/modules/services/etc...depending on your actual architecture
- Streamlined by the *divide-et-impera* principles
- Following adequate architectural patterns during the design phase enables us using specific tools and methodologies

Example:

1. Reqs & Specs (summarized)
   - we need a web server, with a lot of Eps, and possibly making it work with other existing services
   - there are licenses for MS azure and AWS we can exploit
   - a lot of read/write in our business flow, but we might not modify the data
2. Design
   - we follow the micro-services arch pattern
   - will deploy on Azure FAAS and AWS Lambdas, hence use c#/dotNet and Python
   - MongoDB (or in general, no-SQL) are our man

4) Integration (cont'd)

From Design phase

- We decided use dotNet to build the system, we can quite easily streamline testing, CI and CD

As a consequence:

- Integration can happen using MS tools and methodologies
- We can test it using the in-house testing framework
- CI/CD is integrated both in VS, but also in Vscode, but also in github, or devOps

5) Testing & validation

Verify that we meet the specifications and requirements

- Both functional, and not functional
- It is based on the concept of test cases / scenarios
- At different abstraction levels

Examples

- "I can store and modify information on user accounts" – high level, less details
- *"I can store the birth date, but only if it's > 1800 and < today"* – lower level, more details
- *"If I insert Feb 30th as birth date, I get an error"* – lower level, specify a constraint on a number

5) Different levels of testing

"Testing can find errors in your code, but they cannot prove its correctness" (E. Dijkstra)

How do we proceed?

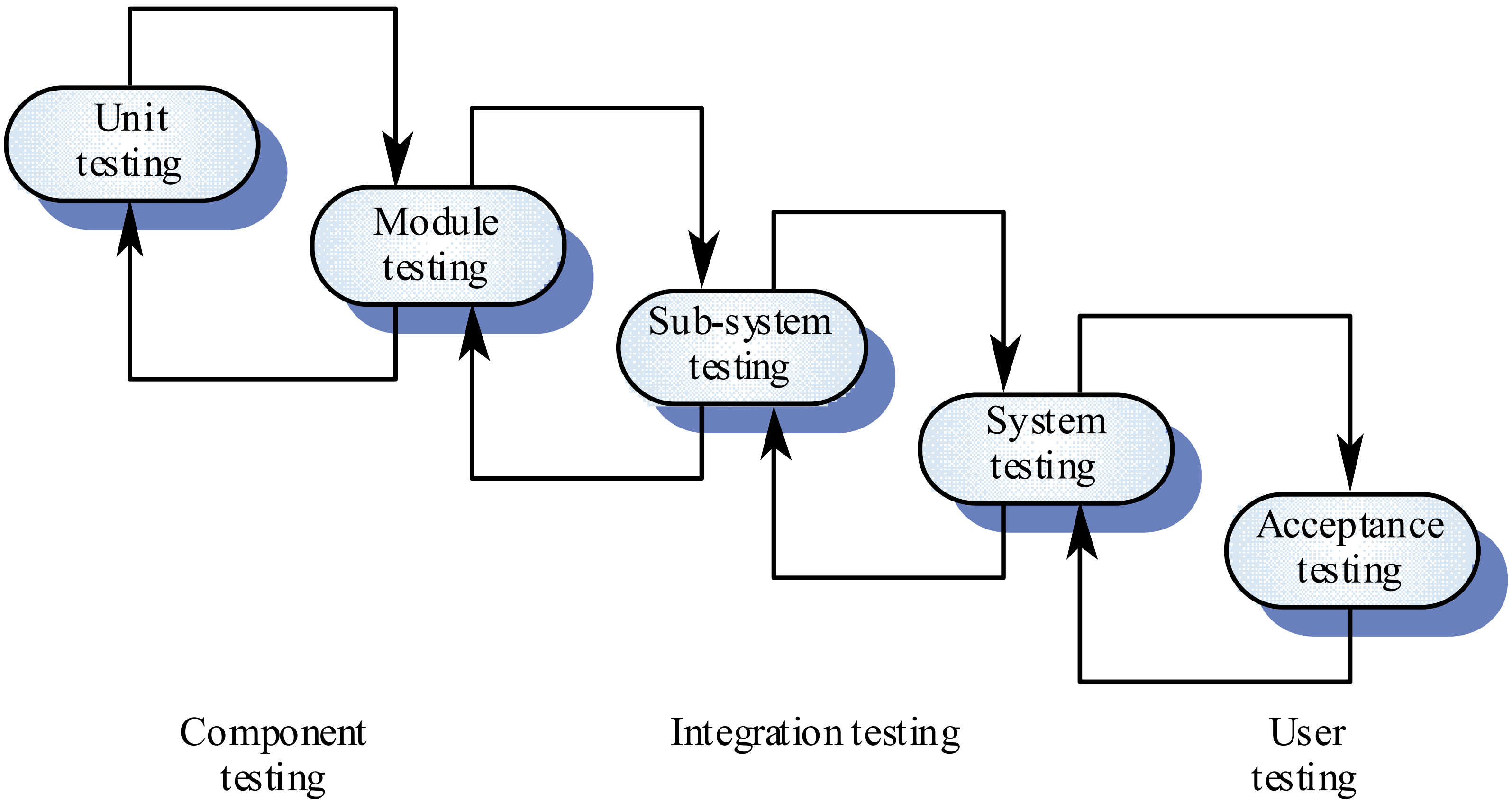
- We need to isolate a number (high number) of test cases that streamline directly by the functional specifications, which enable us to have the customer saying "Yes, it's working properly"

This is called **testing in the large**

- Then, we isolate smaller test-cases for specific components, or blocks of components, to assess that they do the functionality that they are supposed to do
  - E.g., store a number (age) in a DB depending on the FK (Codice Fiscale?)
  - Directly streamline by the system partitioning (*divide-et-impera*) in the system design phase

This is called **testing in the small**

5) Testing & validation – how it was



5) From the small to the large

Unit testing

- Testing a single component (es: a class) in isolation
- Es: store in my age in a table in a DB, calc the area of a rectangle

SMALL

Module testing

- Testing more components that constitute a “functional block”
- Es: store my age in a table, and log this action, in another table (*transactional)*

Integration testing

- Test that different components can work together
- Webervice can work with DB, for instance (see next slide for an example)

LARGE

Functional testing / user acceptance testing

- Involve the customer / end user – always remember, this is what **gives value** to your SW, and, ultimately, to you!
- It is also the boring part...
- Match what's written in the requirements document

5) Testing in the small

- Check single code portions / functionalities, inside a single module/component

In OOP, we partition code within classes and functions

- So, this typically boils down to testing single functions, then single classes

Unit testing

- ..or aggregations of classes (which indeed are other classes)

Module testing

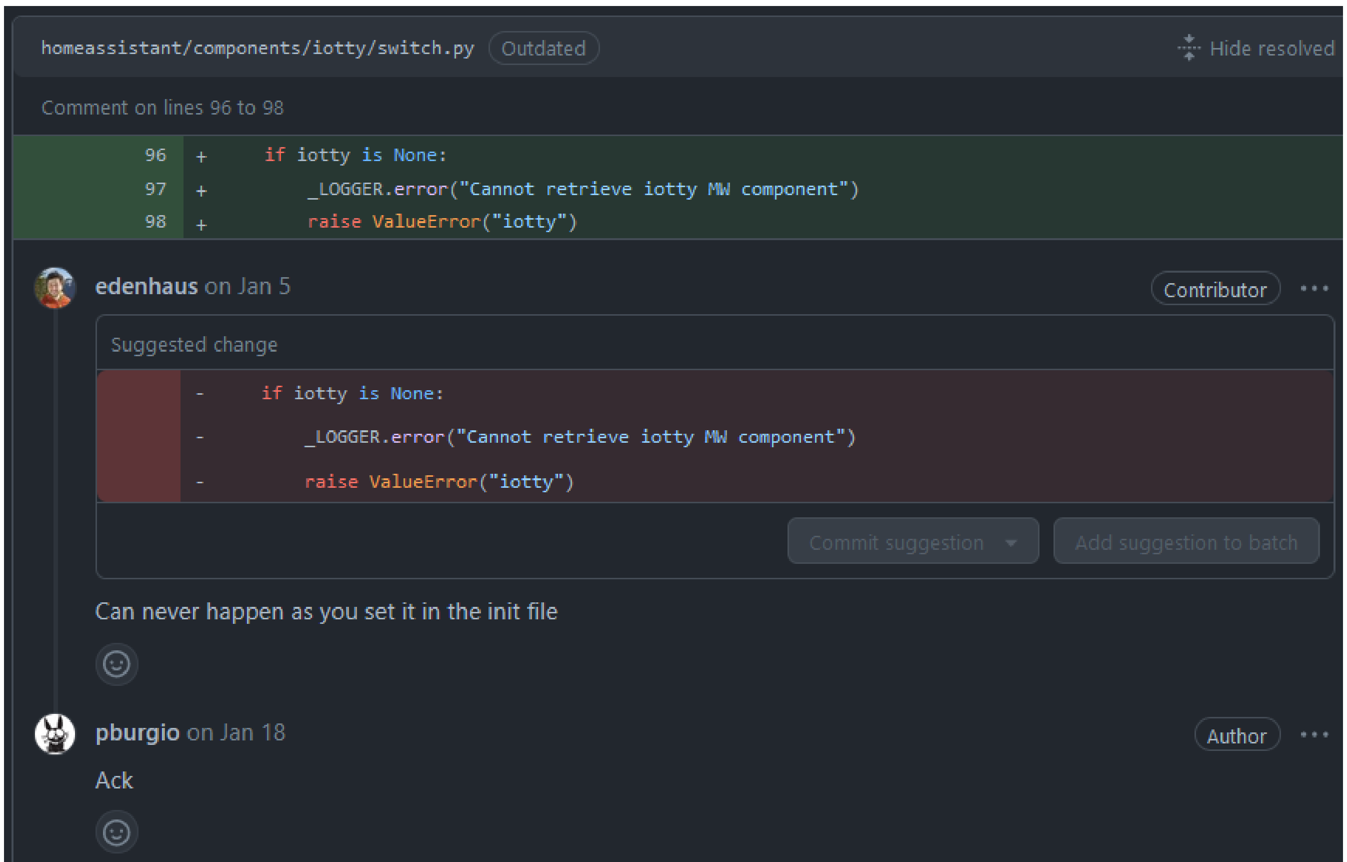
We shall test every single line of code – **coverage test**

- We can test private functions – get deep inside the class – **white-box testing, o**r we can "use" the class – **black-box testing**
- Whe shall test branches and conditions – **branch and condition tests**
- Of course, the #test easily explodes, so we might omit some test, or even understand that some lines of code are unnecessary!!!!

5) In the small – Detect unnecessary code

Example: we null-check for a parameter in a ctor that is only called in a single point of the application!

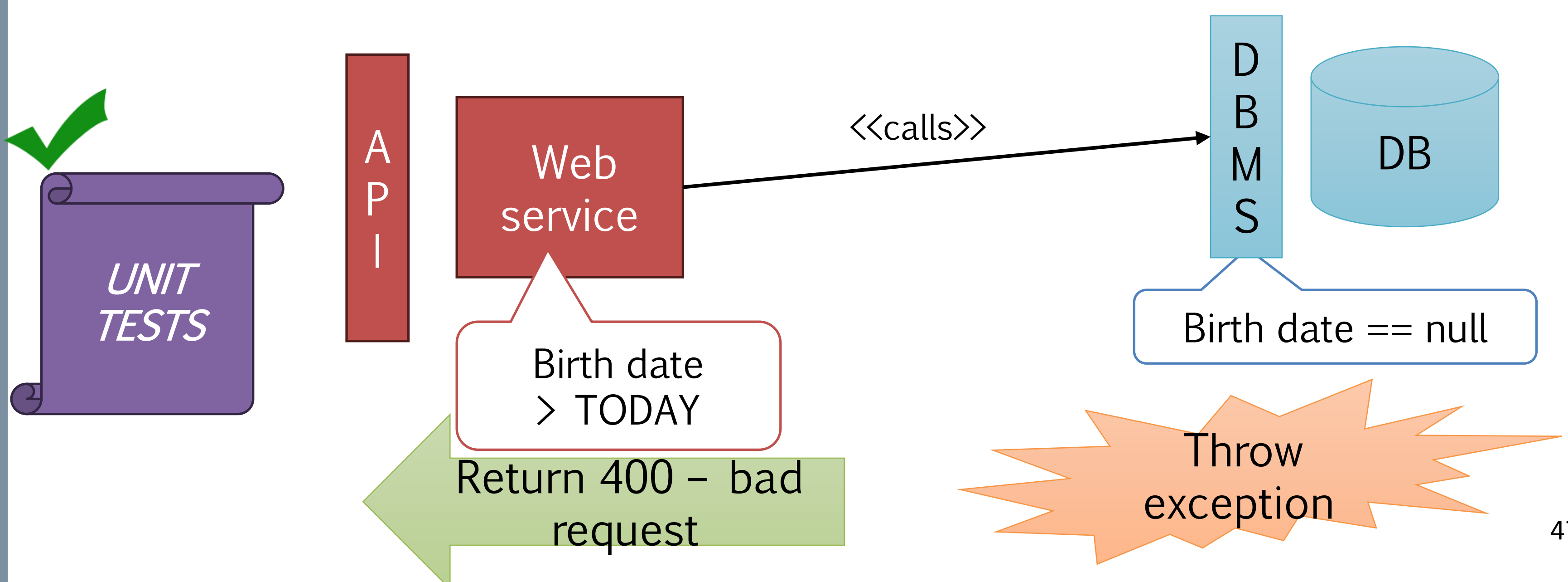
- If this class were released into a library, we need to re-add the check – *why?*



5) Integration test - example

Often, people are confused, as the multiple definitions can be misleading

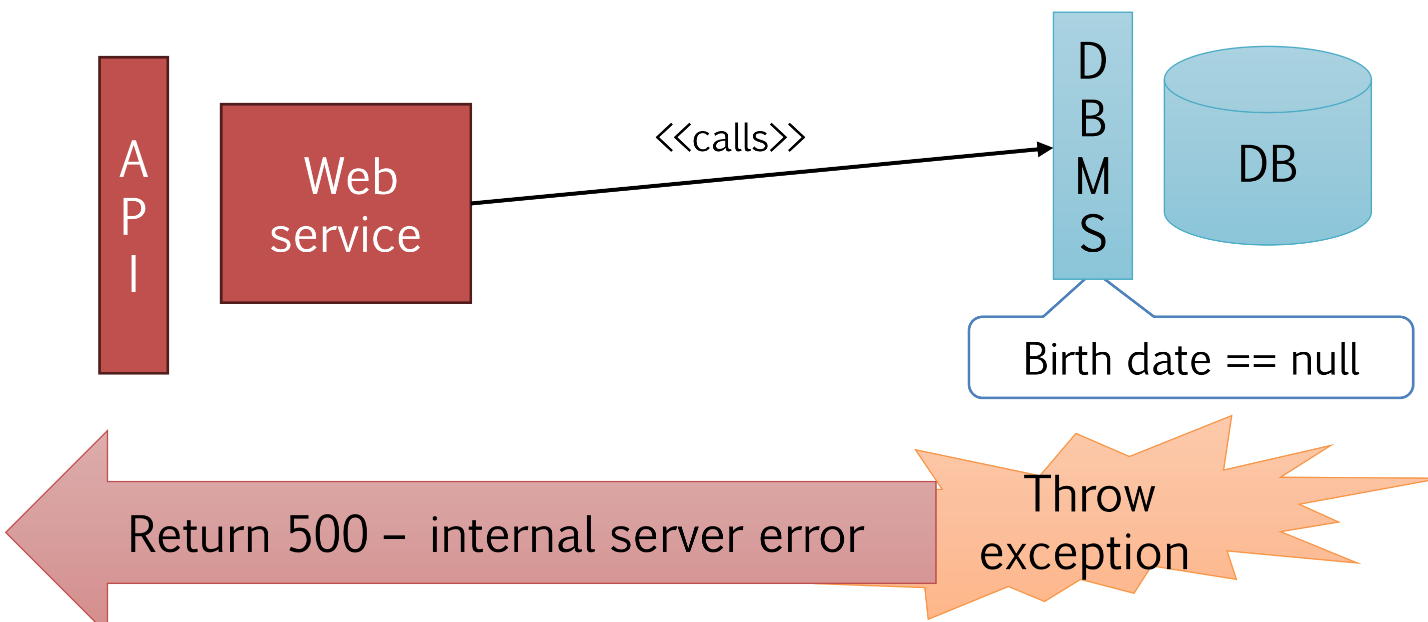
- Example
- We want to store our age in a DB, and we have a Web service; the birth date 1) cannot be > *today*, nor 2) it cannot be null
- Design: check 1) performed by the Web frontend, check 2) performed by the DB
- They both do their work, in isolation (i.e., they throw an exception, or raise some error)



5) Integration test - example

When we glue them together, they might not work properly

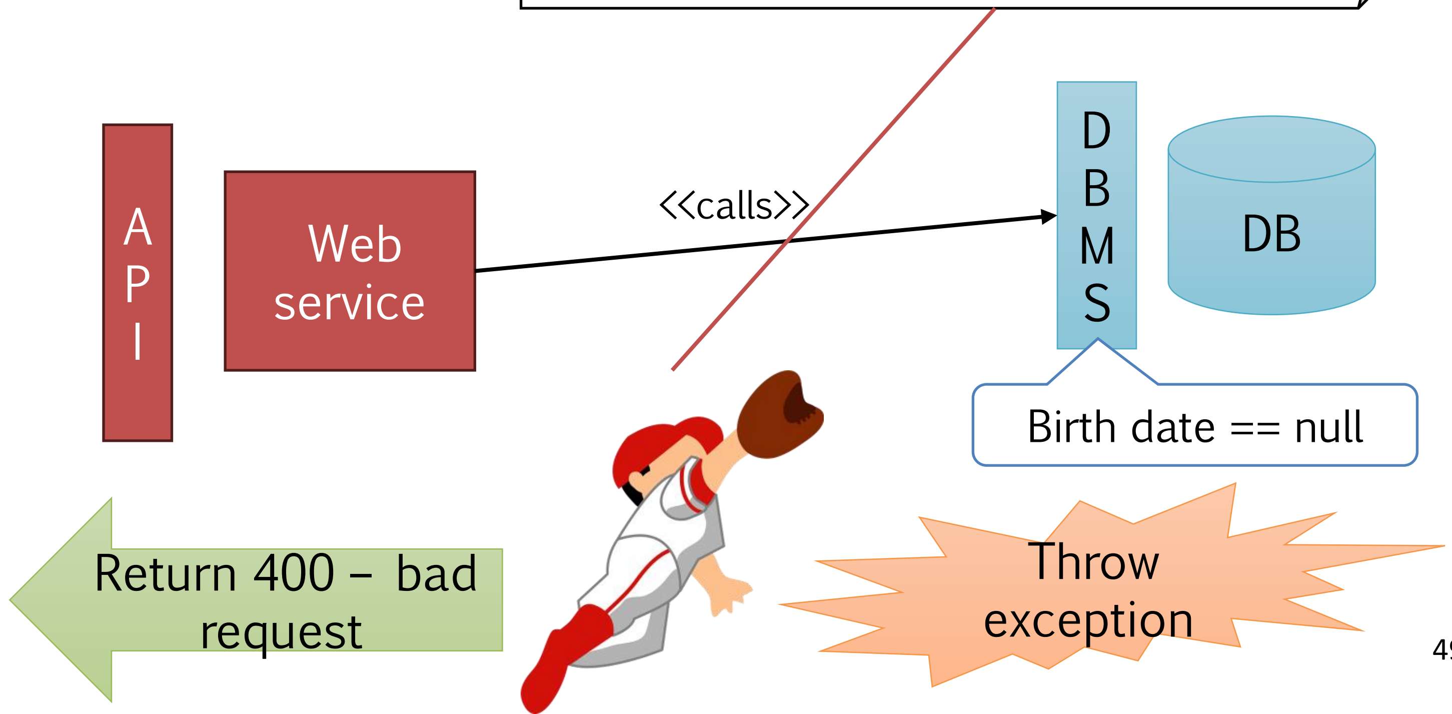
- Typically, a web service **never** returns 5** family errors! Use 4** instead!
- We need to test the behavior of WS when DBMS throws an exception



5) Integration test - example

Solution: simply use **try-catch**

```
try
{
  db.save(cf, <ANY>);
}
catch (NullValueException ex)
{
  return Web400Result(
      ex.Msg + " cannot be null");
}
```

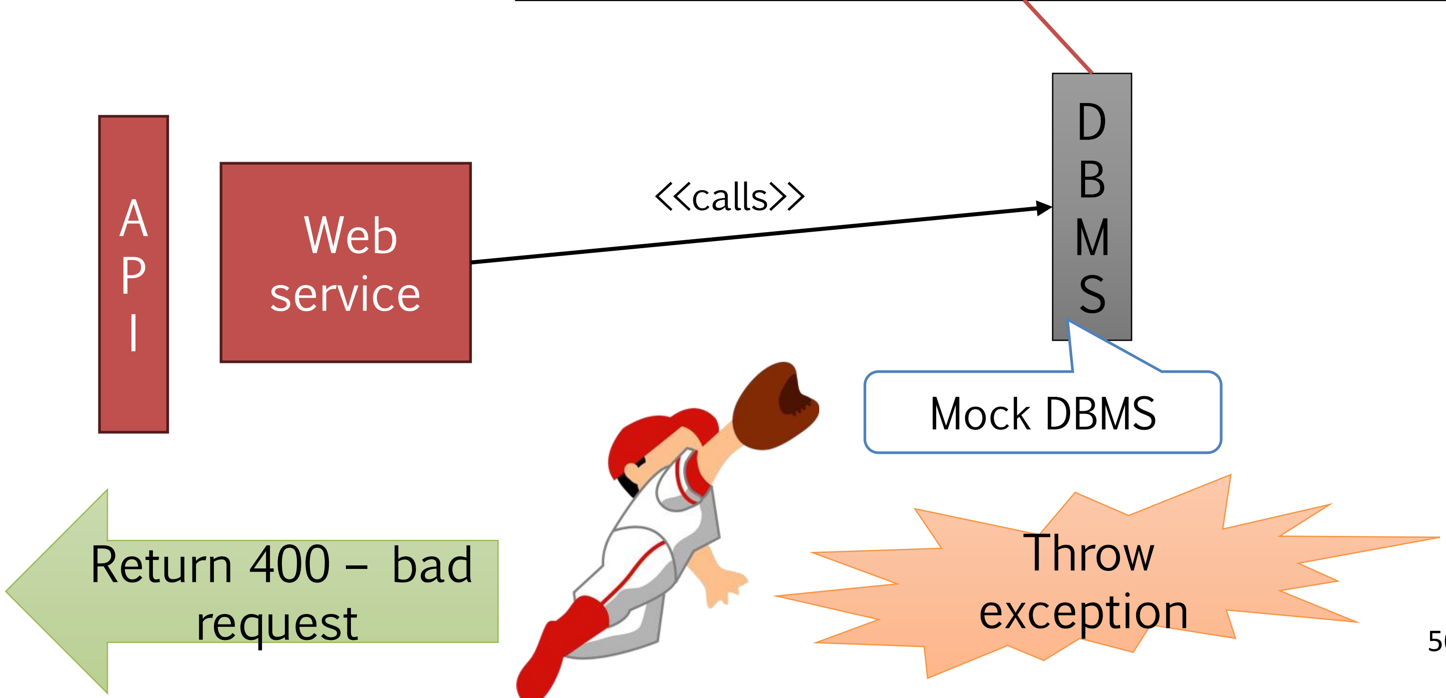


5) Mocking tests

Problem: do we really need a DB to test this?

- This means, potentially, having the whole system in place for tests
- It Of course not, we can **mock** the DBMS with a fake one

```
void save(int key, int age)
{
   throw new NullvalueException("age");
}
```



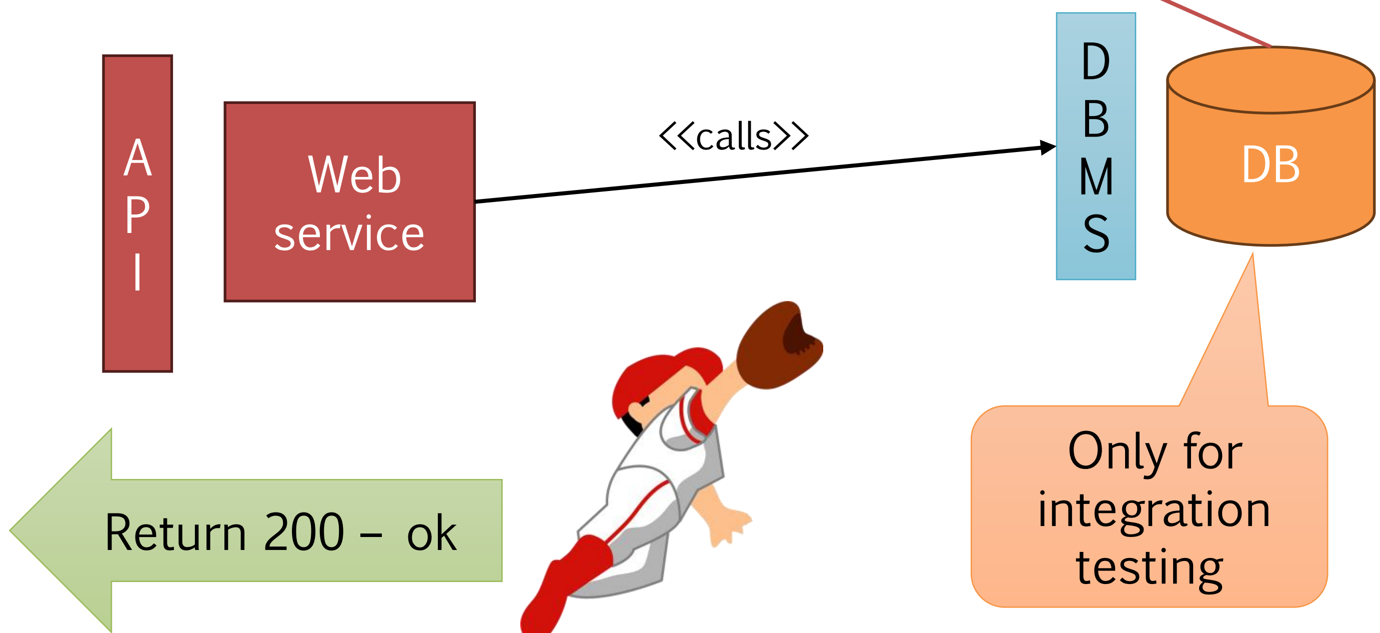
5) Mocking tests

Problem: do we really need a DB to test this?

- We can also mock an **in memory** DB, with no storage
- Of course, data is in RAM, hence won't persist

```
Dictionary<int, int> _ages = ...;

void save(int key, int age)
{
  this._ages.update(key, age);
}
```



5) “A manazza”

Code walk-through

- Run the code with random patterns, to detect problems that we might not have caught in previous phases
- Limited set of persons: not who develop
- Few hours

Code inspection

- More structured
- Search for well-known problems
  - Non-initialized vars
  - Infinite loops
  - Memory leaks/issues

6) Deploy

Continuos Integration / Continuos Delivery – **CI/CD**

- Again, using well-known methodologies and tools that streamline from the framework we want to use
- Often, integrated with the testing phase
- "Click, and the framework does everything" – *PARADISE*

Not always the case...

- Often, projects are smaller and does not require the complexity of a full-fledged framework
- We might even want to deploy our system on-premise (i.e., on our server room) – like we do @ Hipert Lab
- These steps can anyhow be automated "simply" at the "repository level", using GitHub

7) AM / maintenance

The process of modifying/extending the software after the first release, which we agreed with customer

- Debug is always included, for 6/12/24 months (depending on the contract)
- You might even agree with the customer to “sell” Man-Month for working on / directly a perk of functionalities – often to be agreed later
- We will not cover this, as it is really customer/project specific

I can only give you some hints / golden rules

1. Customer **never** know what they want, often neither in the design phase
2. “L’appetito vien mangiando” (for them) – make them hungry! Show the full **potential** of your SW!!
3. This is **not** about your software, this is about **customer mindset!**

Try to enter their mind, speak with them as much as you can, collect as many information on them as you can!

References

Course website

- http://hipert.unimore.it/people/paolob/pub/ProgSW/index.html

Book

- I. Sommerville, "Introduzione all ingegneria del software moderna", Pearson
- Chapter 9 for testing

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/
- https://github.com/pburgio