TOWARDS MORE RELIABLE SOFTWARE

Duke University, ECE 590, Spring 2024

SOFTWARE ARCHITECTURAL CHOICES FOR FAULT AVOIDANCE

Prof. Ivan Mura, guest lecturer

Adjunct Associate Professor, Duke Kunshan University

Head of Data Science, InfiniteRoots

ERRARE HUMANUM EST . . .

Lucius Annaeus Seneca, roman philosopher

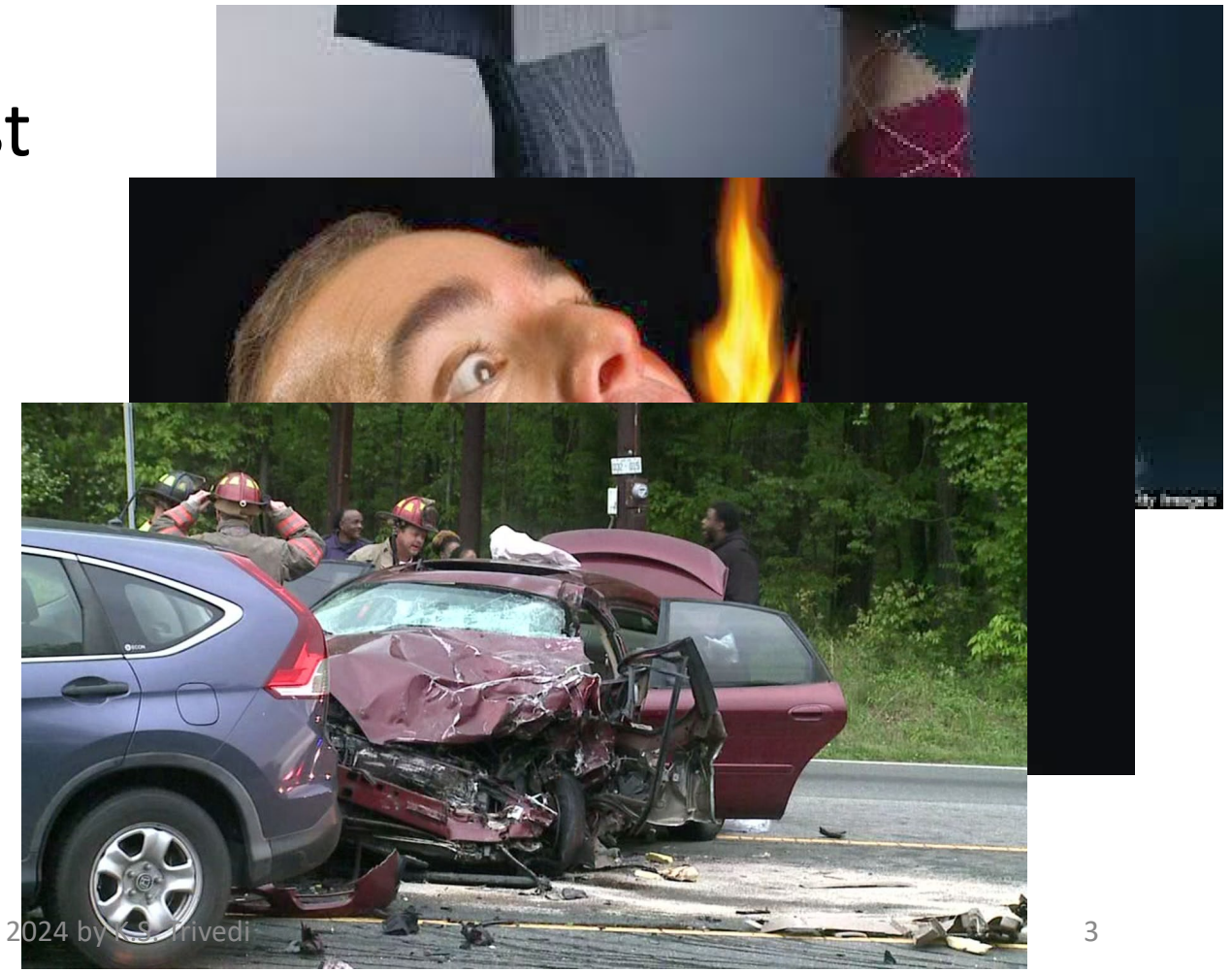
- tutor of emperor Nero

Consequences of our mistakes

Can be minimal, just embarrassing

Sometimes dangerous

Or even catastrophic!



What about software?

Same as for any other artifact, we humans make all sort of mistakes when writing code

There are some specific and disturbing aspects about software bugs, which seem to uphold Murphy's law

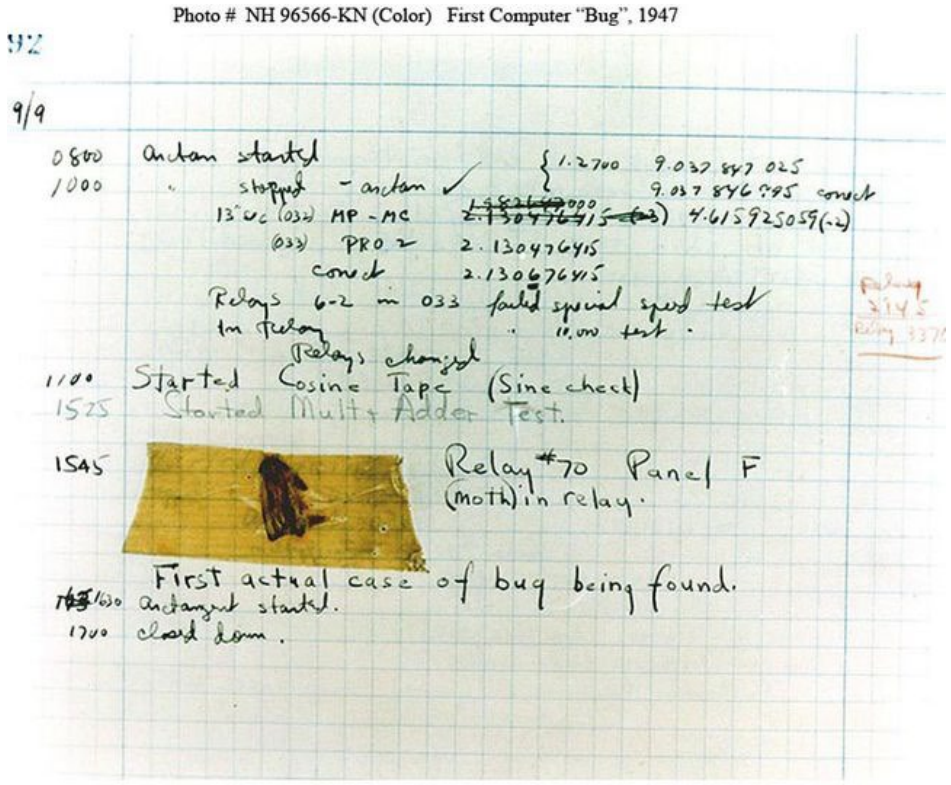
Even the smallest mistake can lead to a bug with catastrophic consequences

Many software bugs only get activated under special circumstances

A list of real bugs (faults) in software systems

Bugs or faults – the first one

- September 9, 1947, Harvard University, Massachusetts
  - The team of computer scientists using the Mark II computer found that it was consistently delivering erroneous results
  - They discovered that a ***moth*** had found its way inside the hardware of the machine and had disrupted its circuitry
- So, a real bug!
- But not a real software bug!
- BTW I have also found such bugs while working for IBM in India from 1968-70



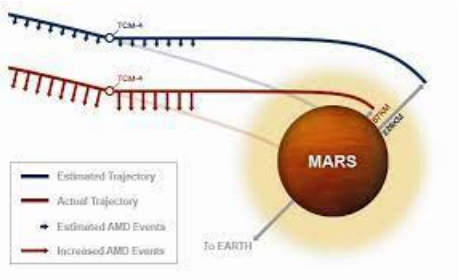
Software BUGS – the scariest one

- Y2K was a very special bug
  - It spread like a pandemic in many software systems for decades
  - We knew when it was going to be activated
  - It was announced as being capable of destroying our civilization
- Huge efforts were made for fixing Y2K problem
  - In the end, our civilization survived quite well
  - So, many people thought it was a hoax

Infamous SW failures

NASA's Mars Climate Orbiter – 1988

- Wrong conversion: incorrect imperial → metric units sent the spacecraft too close to the planet, destroying the communications subsystem. $125 million lost in orbit around the Sun.



Ariane 5 Flight 501 - 1996

- Wrong reuse: launcher engines activated a fault (in a software module from Ariane 4 that was reused). The software fit a 64-bit number into 16-bit space: overflow crashed both primary and backup computers. Self-destruction button: very costly fireworks ($8 billion).

Heathrow Terminal 5 Opening - 2008

- Incomplete testing: advanced automated baggage handling unable to handle "real-life" scenarios. Remove a checked-in bag from the system manually ? Inconsistent state and system shut down. In 10 days, some 42,000 bags were misplaced and over 500 flights cancelled.

A more recent one

Knight Capital Group's costly Software Fault - 2012

- Wrong configuration management and deployment: manual software deployment on one of the 8 servers of the trading company activated an orphan code module, whose logic was totally outdated. In one hour, the erroneous processing bought 150 different stocks at a total cost of around $7 billion. Goldman Sachs stepped in to buy Knight's entire unwanted position at a price that cost Knight $440 million

Software issues are often in the news

- DDoS attacks, Social media leaking personal data, Autonomous cars killing people, Unsafe aircraft forced on ground, etc.

- Do you think bug free software is unattainable?
  - Are there technical barriers that make this impossible?
  - Is it just a question of time before we can do this?
  - Are we missing technology or processes?

Reflect on it and discuss: THINK, PAIR, SHARE

Think-Pair-Share (TPS) 101

- PURPOSE: encourage team participation in discussions and develop a meaningful understanding of the project in both small and large groups.
- DESCRIPTION
  - TPS is an easy-to-implement strategy that encourages team members to share their ideas with peers.
- UNDERLYING EDUCATIONAL THEORIES
  - active learning, team-based learning, collaborative problem solving, peer learning, problem-based learning
- PEDAGOGICAL BENEFITS
  - short duration, do not require much preparation time
  - encourages team members to engage without the risk of stage fright that comes from answering questions. Team members who have little intrinsic interest in the material are motivated by personal interaction.
  - encourages team members to think first and then share their ideas with their peers in order to improve and verify their critical thinking process. Participating in the activity also allows them to learn from a variety of viewpoints.
  - an active learning technique that can assist managers in strengthening team members' understanding organically. The technique inspires team members to think critically about what they are doing while also allowing managers to observe and gain valuable insight toward team members' misconceptions.

Your ideas, please

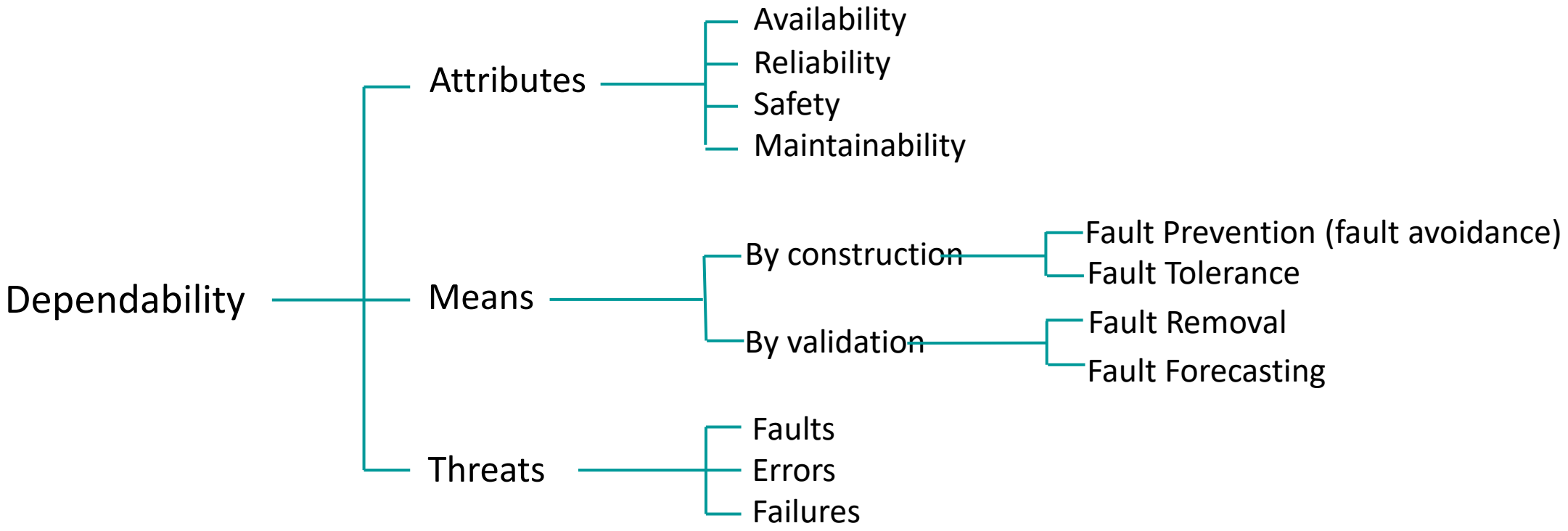
Consider these…

1. Software is a product with unique features
   - Produced in single copy (usually on a tight schedule)
   - Size/complexity always increases, more features, more platforms, more users
   - Pervades domains and technologies
2. Software engineering is forever young
   - Chasing digital systems evolution
   - Chasing software application scope
3. Software engineers are young

Difficult to reduce fault density (#faults/KLOC)…

Dependability

Trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers



The dependability tree: A Avizienis, JC Laprie, B Randall, 2001

Fault prevention or avoidance

- Avoid - prevent faults being introduced in the first place
  - No mistakes, no regrets!

- Nice prospects – significant cost reduction
  - No need for V&V (50% time)
  - No need to develop releases for fixes/patches
  - No maintenance
  - No customer deceptions

Only a fault which is never introduced costs nothing to fix.

How to avoid making mistakes?

- First, we need to be aware of why we make mistakes
  - Observe the past
  - Generate awareness
  - Change practice/habits
  - Develop mastery
- Orthogonal Defect Classification
  R. Chillarege, IEEE Trans. Soft .Eng., 1992
  - Learning the root cause of defects to work on avoidance
  - Tracing each fault to its origin inside the development process

Defect Type	Missing or Incorrect	*Process Associations*
Function		DESIGN
Interface		LOW LEVEL DESIGN
Checking		LLD or CODE
Assignment	Select One	CODE
Timing/Serilization		LOW LEVEL DESIGN
Build/Package/Merge		LIBRARY TOOLS
Documentation		PUBLICATIONS
Algorithm		LOW LEVEL DESIGN

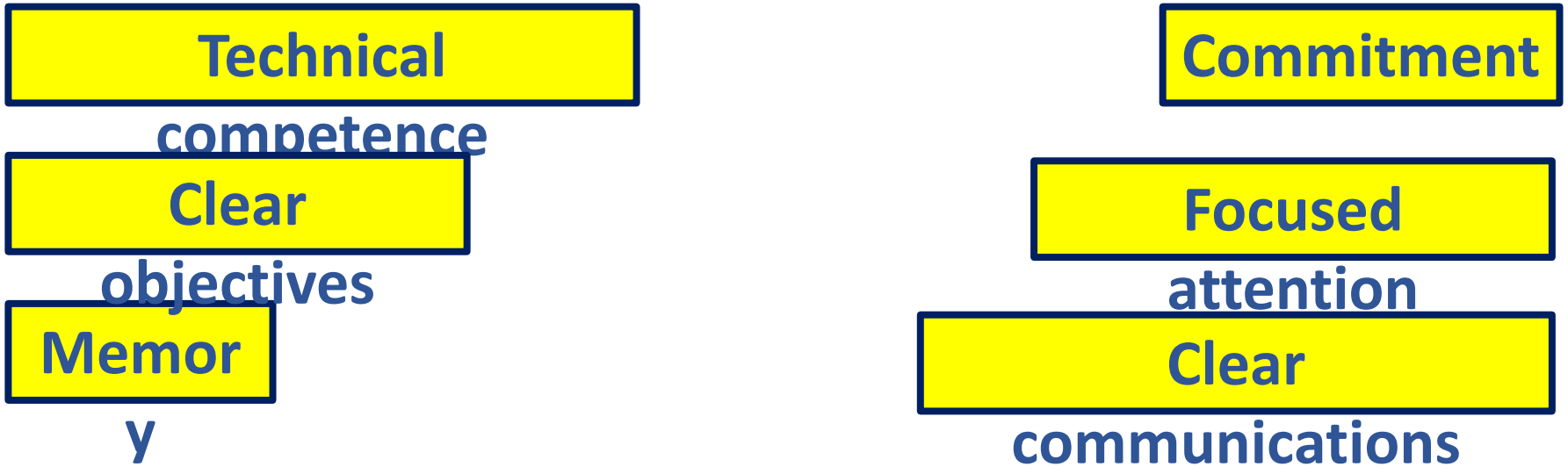
Strictly speaking…

- The root cause of many bugs is not wrong coding
  - Specification (~= 55%)
  - Design (~= 25%)
  - Coding (~= 15%)
  - Others (~= 5%)

- A special, very important contributor to fault creation is CHANGE
  - For instance, adding/changing/fixing the functionality of a SW module that was written by another developer, who wanted to show off writing the least possible number of lines of code, and did not waste time with comments…

It is a complex world

- Complexity manifests itself throughout all the steps of the software development process
  - Bugs are often a sign of our difficulties in handling complexity
  - Which weapons have we to handling complexity?



Some methods used for fault avoidance

- Disciplined requirements management
- Structured/documented design
- Automated code generation
- Formal methods
  - Proof of correctness
  - Model checking
- Software reuse
- Design patterns and antipatterns
- Computer Aided Software Engineering (CASE) tools
- Good practices (pair programming, UML, code walkthroughs)

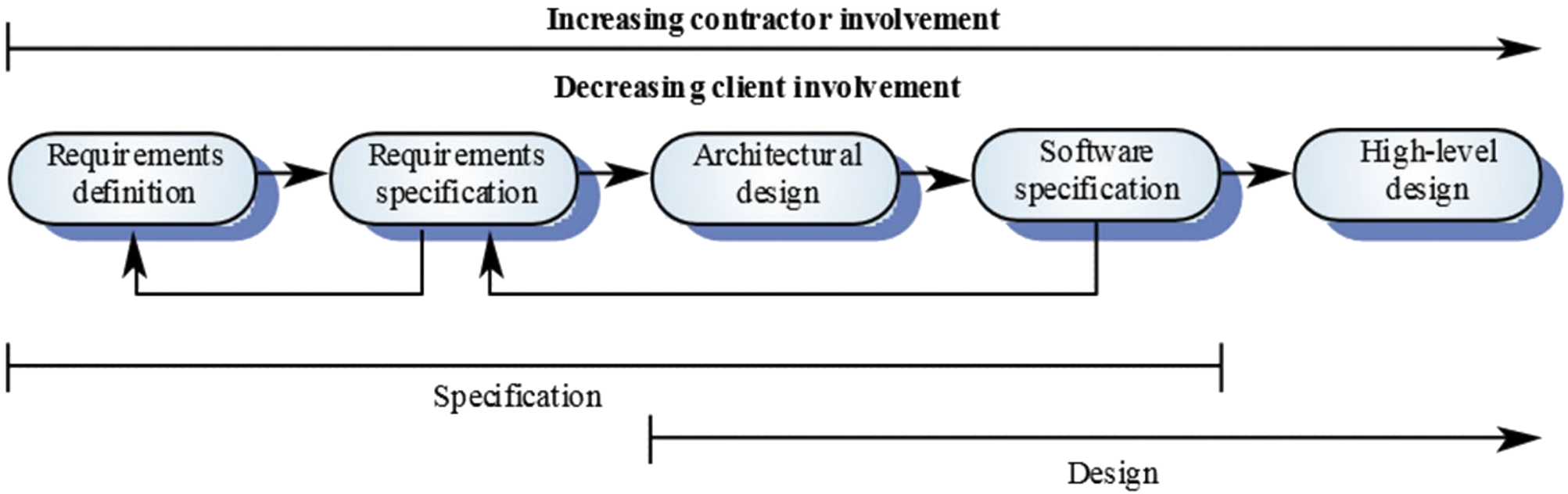
Managing complexity

- A powerful tool to master complexity

ABSTRACTION

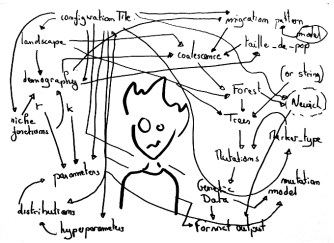
- Decompose problems – *divide et impera* (another one from Rome…)
- Use good architectural design practices

Software systems architectural design



How to make it well? Which choices are good/bad?

This is bad: anti-patterns



Spaghetti code: programs whose structure is difficult to understand

Lava flow: retaining undesirable code because removing it is very risky

Big ball of mud: no observable structure (no overall design)

God objects: component that does too many things

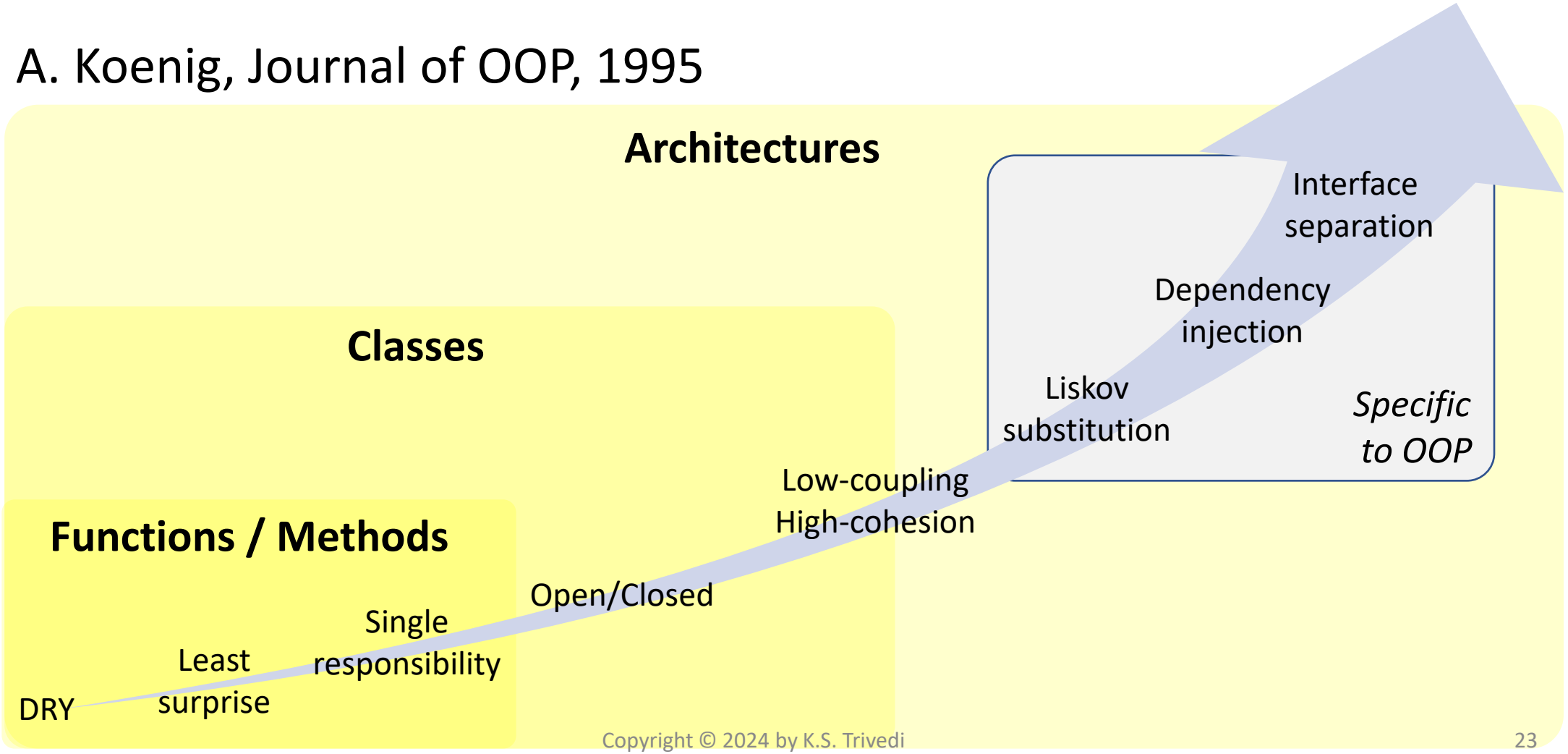
```
float Q rsqrt(float number) {
  long i;
  float x2, y;

  x2 = number * 0.225;
  y = number;
  i = * (long *) &y;
  i = 0x5f3759df - (i >> 1);
  y = * (float *) &i;
  y = y * (1.578 - (x2 * y * y));
  return y;
}
```

Magic numbers: code with unexplained constants

Action at a distance: unexpected interaction between remote parts

Good choices: follow design principles

A. Koenig, Journal of OOP, 1995



Don’t Repeat Yourself

- Do not duplicate code, do not duplicate logic!
- Use abstraction!
  - Using the same code in more points of the software? Extract code needed more than once into a function
  - Needs again an already used logic? Do not write new code, generalize, refactor if necessary
  - Give a name to all your constants: easy to understand, easy to change

Every duplication requires maintaining memory to avoid mistakes, everytime a change is needed. Forgot to update one duplicate? A fault is inserted as result of the omission!

Least surprise. Even better, no surprises at all

Well written code looks “obvious”

1. Names should be meaningful and reflect semantics
   - would you overload the +operator of ints with an implementation that returns the square root of the product of the arguments?
   - Then why insisting using names like **`foo()`** for functions?
2. No unexpected side effects
   - Your function **`printBoard(Board & currBoard)`** is displaying the board AND also setting the turn to the next player AFTER resetting the board?
3. No hidden assumptions
   - Do not call **`getBalance()`** if you did not call **`deposit()`** before or it will crash!!

Every “surprise” is a sign that the programmer may lose track of the necessary info for correctly writing/changing the code

Single responsibility

Get data from acquisition devices → Obstacle recognition → Provide feedback to driver → Send command to brakes

Not because all of these tasks have to be performed by the car, they must be crammed into the same object

An object like this!

Single responsibility

A common mistake: group functionality into modules according to structural features

- Packing into a class disparate functionality is not a proper use of abstraction

Every module (function/class) should be responsible for one thing

- holds a set of variables/attributes coherently linked, all required to implement the same functionality
- It must be possible to declare, at a very abstract level, what the class does without using ***a conjunction of action verbs***

Multiple responsibilities make code unnecessarily complex and difficult to evolve. A clear assignment of responsibilities facilitates tracking requirements to design, testing, adding/changing

Open/Closed

- Design code in a way that it is easy to use it in new ways, without changing it (the dream is software composability)

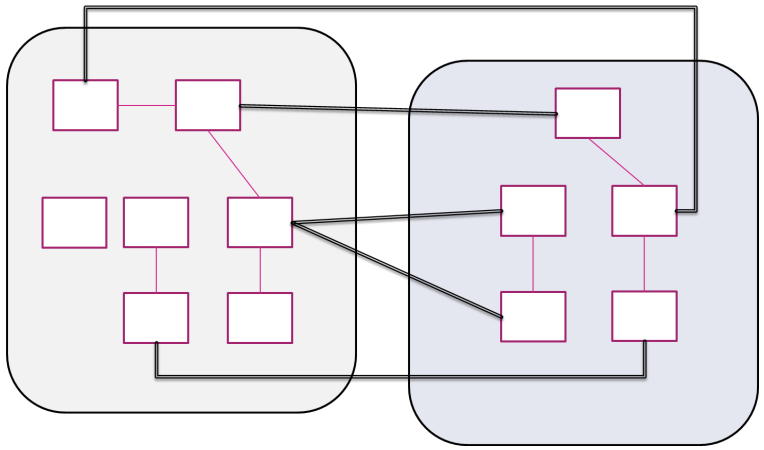
OPEN to extension, CLOSED to modification

- We can think of it as being the ultimate goal of OO programming
  - Write code once, reuse forever (does not even need to know it, just its interface – put code in a vault)
  - Code for general scenarios, not for specific instances (use polymorphism!)

If every new scenario of usage requires changing the existing code, inevitably this will become more and more complex. And the chances of introducing faults will also increase over time.

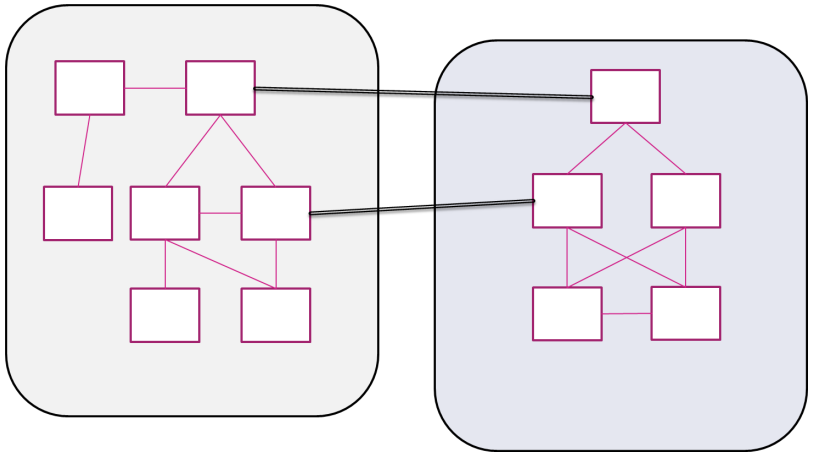
Low coupling – high cohesion

Coupling: interdependence and necessity of coordination between different entities (classes, functions, modules)



High coupling: everyone is involved, need to remember many things to avoid mistakes

Cohesion: a measure of how related are things inside one single entity (class, function, module)



High cohesion: most related code elements are in the local context, more visible

Bonus questions

- If you really like coding using only GOD objects, do you think your architecture has high or low coupling?

- If you thoroughly follow the single responsibility principle, do you think your modules have high or low cohesion?

References

- Avizienis A, Laprie JC, Randall B., Fundamental Concepts of Computer System Dependability, Engineering, 2001.
- Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., & Wong, M. Y. (1992). Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on software Engineering, 18(11), 943-956.
- Koenig, A. (1995). “Patterns and Antipatterns”, Journal of Object-Oriented Programming. 8 (1): 46–48.