



# Generic or specific?

Making sensible software design decisions

Bert Jan Schrijver  
bertjan@openvalue.eu

 @bjschrijver

Let's meet  
Bert Jan Schrijver



● OPENVALUE

● .nl.  
jug

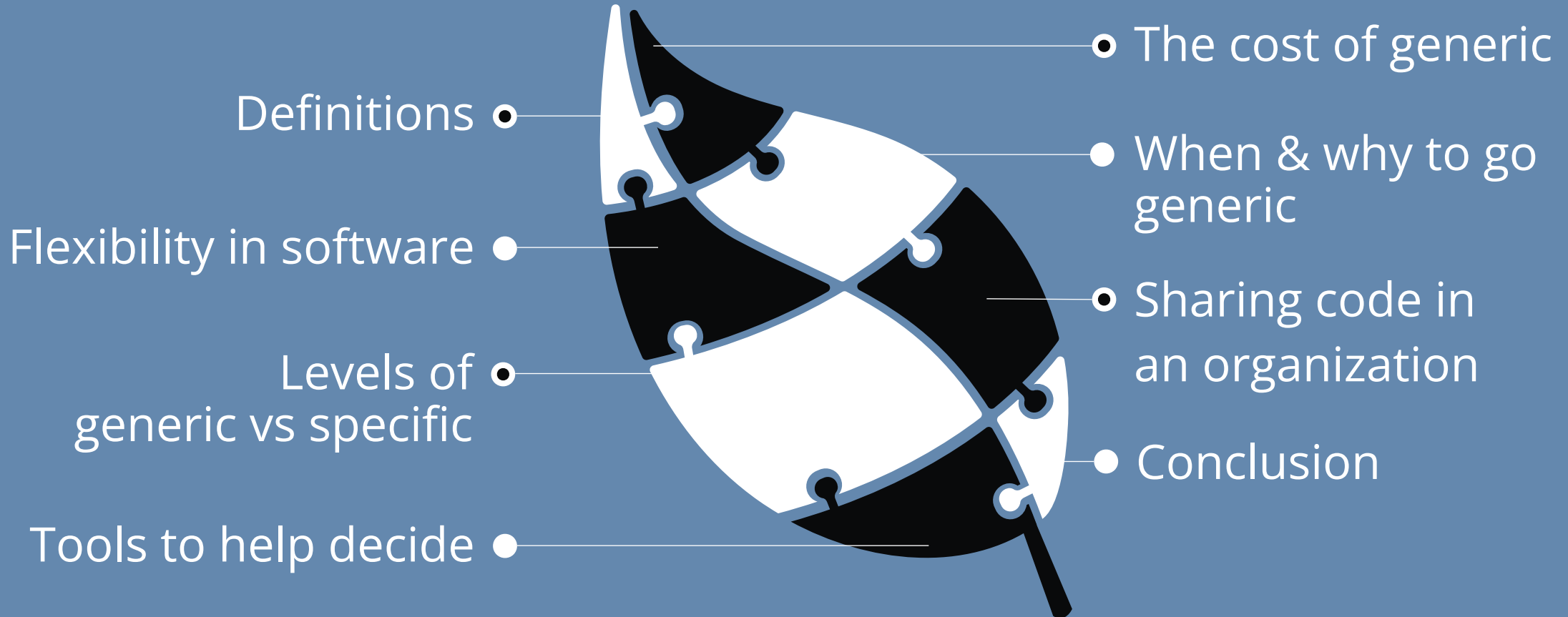


 @bjschrijver

What's next?

# Outline

---

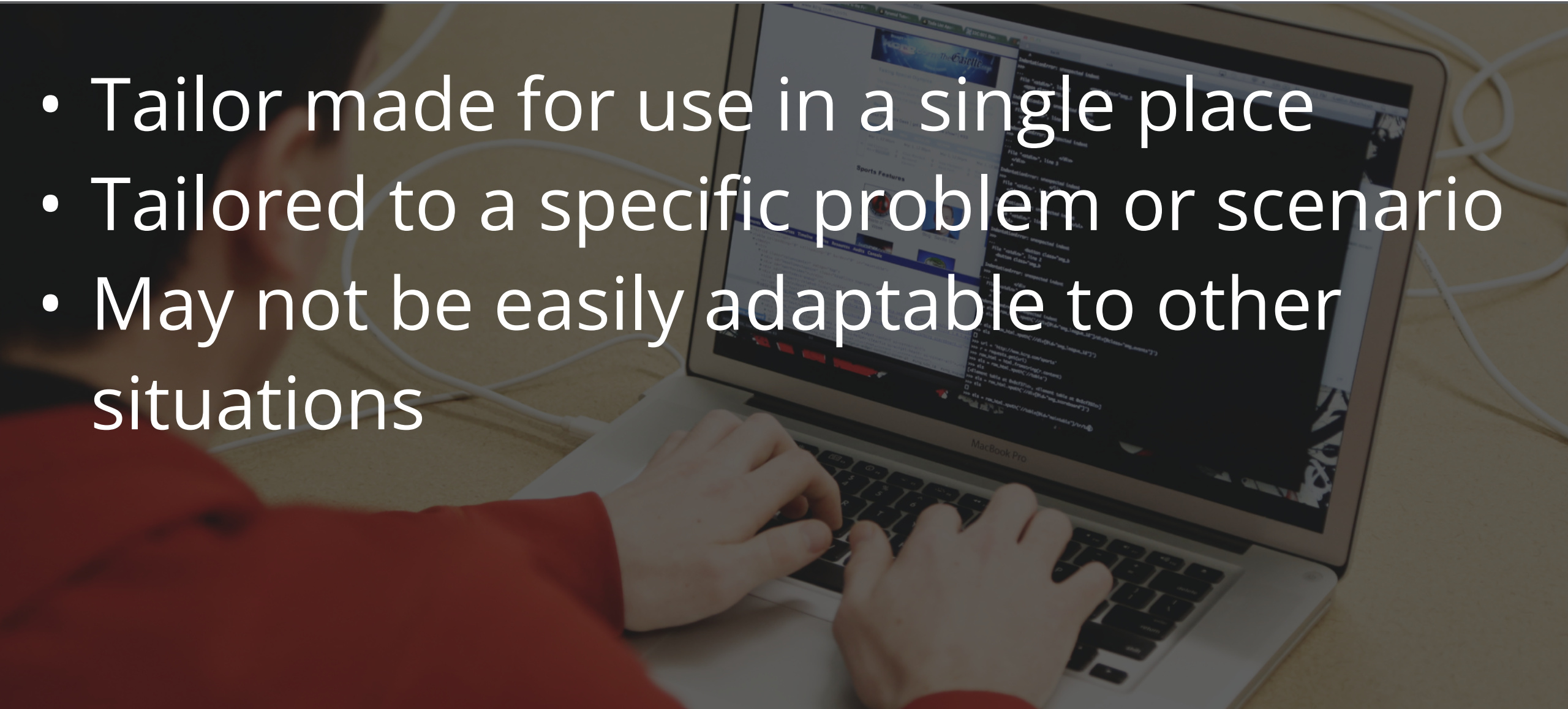




What is software design?

# Specific solution (or design)

- Tailor made for use in a single place
- Tailored to a specific problem or scenario
- May not be easily adaptable to other situations

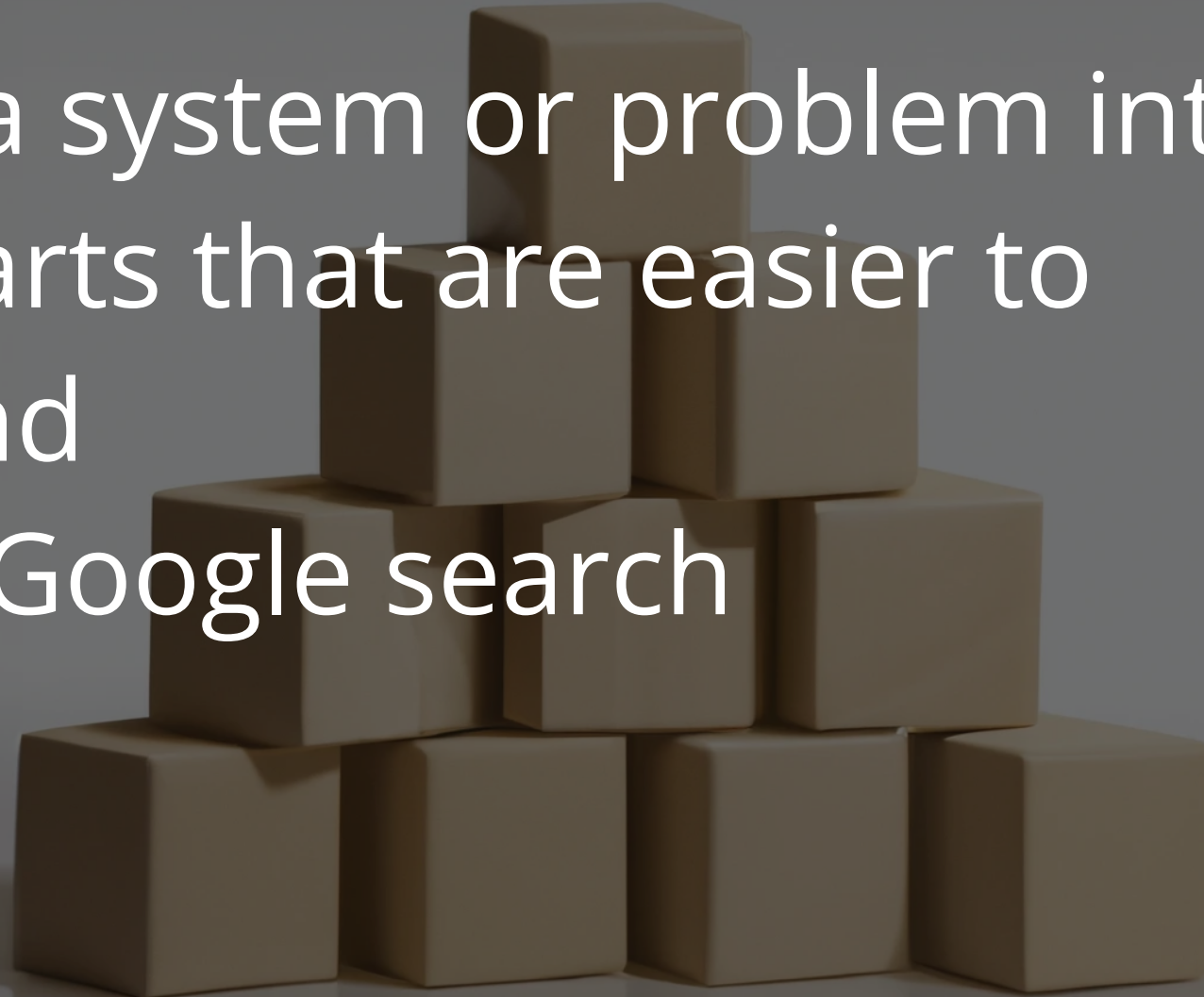


# Generic solution (or design)

- More flexible and reusable solution
- Solution can be applied to a wide range of problems or scenarios
- Generifield solution that can be used in more than 1 place

# Hierarchical decomposition

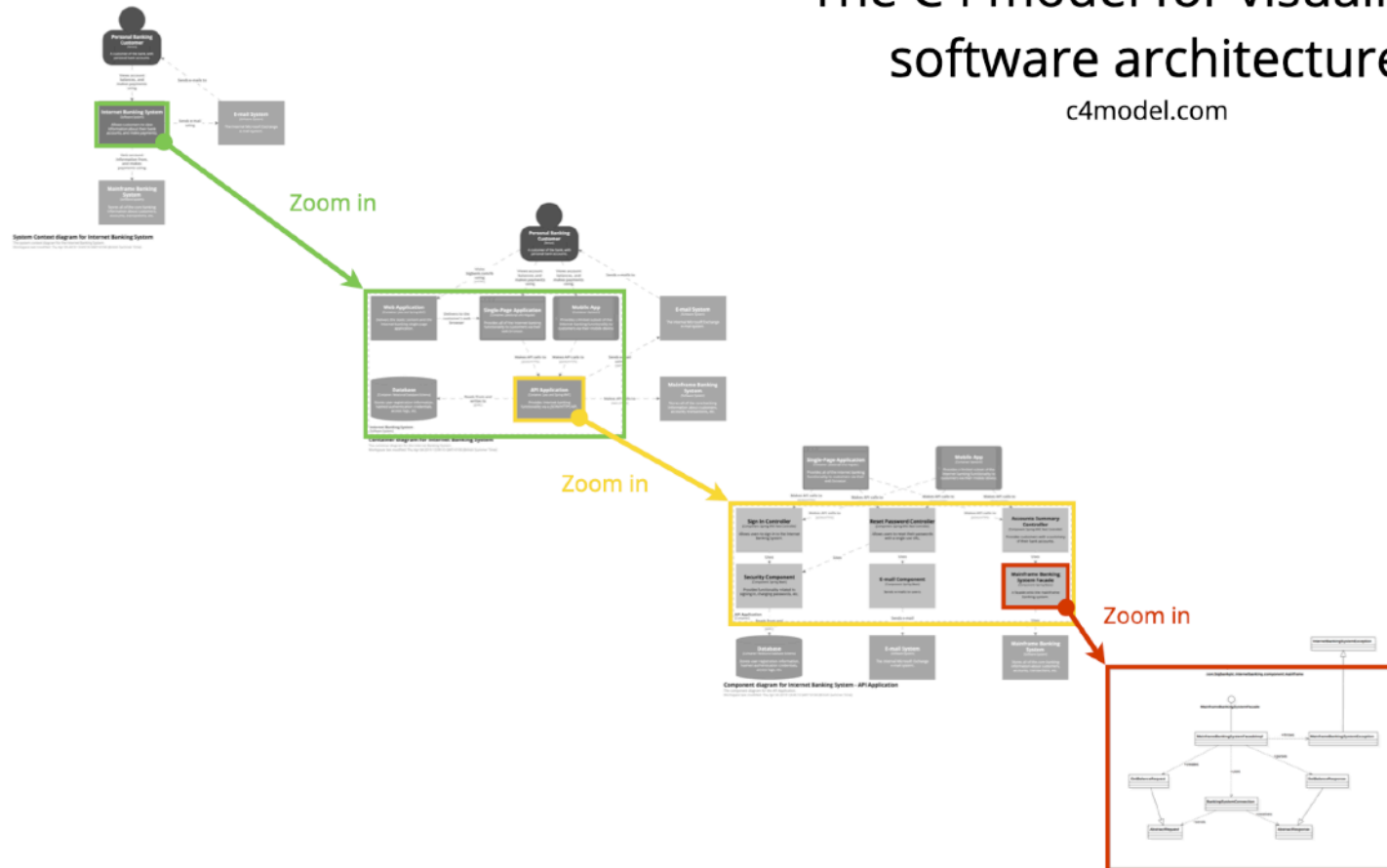
- Breaking a system or problem into smaller parts that are easier to understand
- Example: Google search



# Hierarchical decomposition

## The C4 model for visualising software architecture

c4model.com



Level 1  
Context

Level 2  
Containers

Level 3  
Components

Level 4  
Code



# Coupling

- Kind and degree of interdependence between building blocks of software
- Measure of how closely connected two components are
- Usually contrasted with cohesion (low coupling -> high cohesion)

# Types of coupling

- Inheritance
- Messages or events
- Temporal
- Data types
- Data
- Code / API (binary or source)

A pair of metal handcuffs is shown in an open position, with the two circular cuffs facing outwards. The central locking mechanism is visible. Below the handcuffs, a metal key is shown, which is used to unlock the cuffs. The entire scene is set against a dark, textured background.

Be careful with coupling!

The image shows three metal pipe fittings arranged horizontally against a dark grey background. From left to right: a hex nut with a threaded hole, a hex plug with a threaded end, and a hex coupling with a threaded hole on one end and a smooth bore on the other. The text "Generic solution = coupling!" is overlaid in white, centered across the middle of the fittings.

Generic solution = coupling!

The image features four rolled-up towels of different colors: a dark red one at the top center, a medium blue one on the left, a light blue one in the middle, and a large, thick blue one on the right. The towels are set against a dark grey background. The text "The risk of DRY" is overlaid in white, centered horizontally across the middle of the towels.

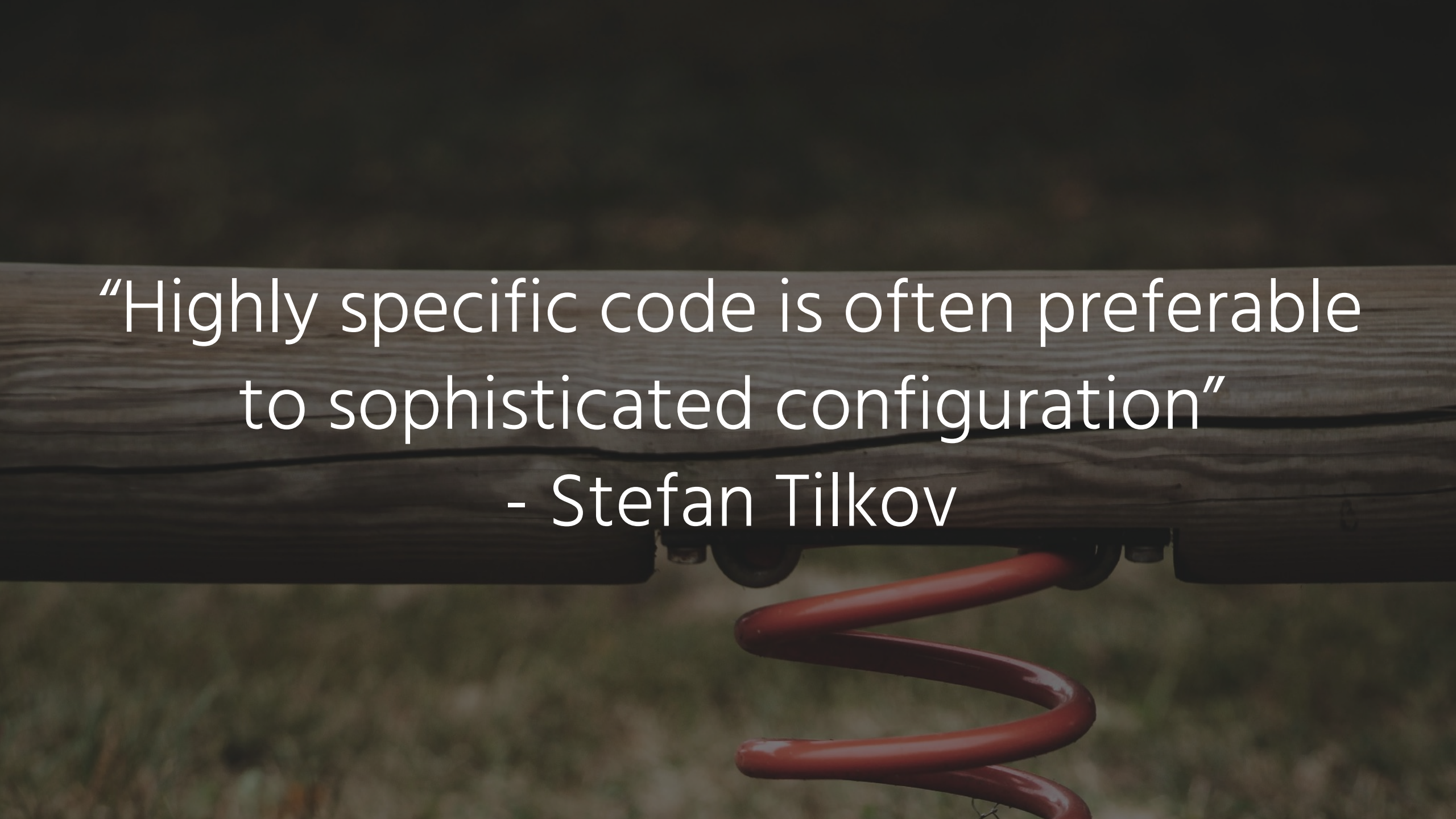
The risk of DRY

# “Future proof” design

- Should we be prepared for future changes?
- Design should be structured to accommodate change
- Risk management: risk of wrong decision

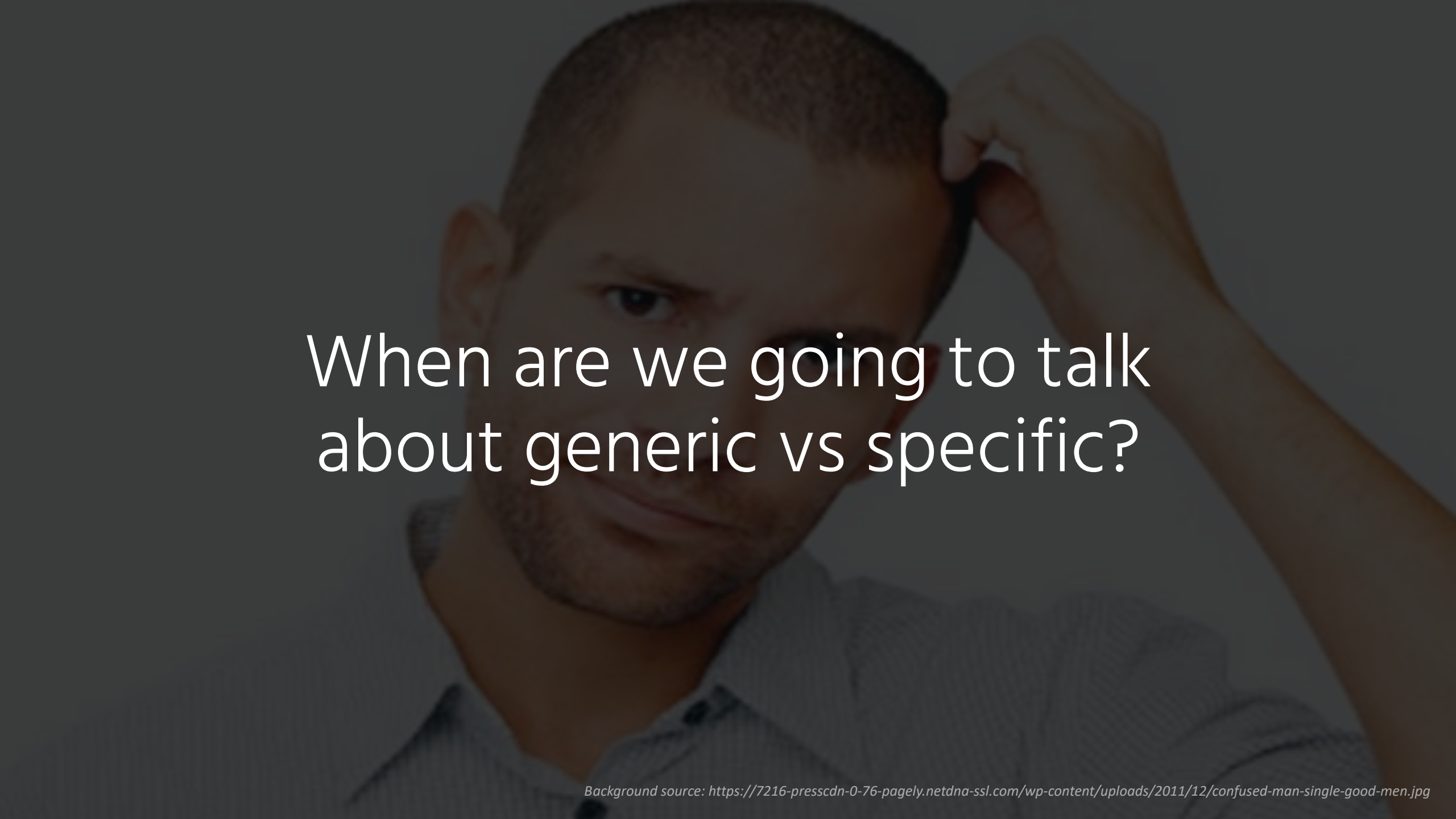
A close-up photograph of a weathered wooden beam. A red metal spring is attached to the underside of the beam, extending downwards. The background is a blurred, natural setting, possibly outdoors. The text "About flexibility in software..." is overlaid in white on the wooden beam.

About flexibility in software...

A red metal spring is suspended from a dark wooden beam. The background is a blurred outdoor setting with green grass and a dark sky. The text is overlaid on the wooden beam.

“Highly specific code is often preferable  
to sophisticated configuration”  
- Stefan Tilkov



A man with a short haircut and a light beard is shown from the chest up. He is looking slightly to the right of the camera with a thoughtful or confused expression. His right hand is raised to his forehead, with his fingers resting on his hair. He is wearing a light-colored, short-sleeved button-down shirt. The background is a plain, light color.

When are we going to talk  
about generic vs specific?

# Generic vs specific: levels

- Code / class level
- Library level
- Data level
- (Micro)service level
- Organisation level



Generic or specific?

# Tools to help decide

- Do we really need this now? (YAGNI)
- Time/effort for generic vs specific
- Myth of “first time right”
- Complexity and scope
- Future needs and evolution
- The rule of three

# The rule of three

- When reusing code, copy it once, and only abstract the third time
  - Avoid writing the wrong abstraction
  - It's easier to make a good abstraction from duplicated code than to refactor the wrong abstraction
- "Three strikes and you refactor"

# The rule of three

- First case: Just build it, don't genericise at all. Solve the problem in front of you (YAGNI)
- Second case: Duplicate the original, redesign and extract common behaviour while you change
- Third case: examine the lessons from the first two passes, design a generic solution that will make it easy to add your third case

# Design heuristics

- Pass 1: YAGNI / rule of three: as simple and specific as possible
- Pass 2: based on solution domain knowledge: is a generic solution less work?
- Pass 3: based on problem domain knowledge: is the easiest solution actually correct?
- Pass 4: looking at customer behaviour or other non technical considerations, does this change your decision?

# Strategic design

- Concept from Domain Driven Design
- Tool to help decide for generic vs specific
- But more about building yourself or not
- Subdomains:
  - Core domain
  - Supporting subdomain
  - Generic subdomain



# Conway's law

- Organizations design systems that mirror their own communication structure
- Don't force a solution that goes against the organisation structure
- Be careful to go generic when teams don't want to work together

A blurred photograph of a classroom. In the foreground, the backs of several children's heads and shoulders are visible as they sit at desks. One child has a bright pink backpack. In the background, a teacher is standing at the front of the room, facing the class. The overall image is dimly lit and out of focus, with a dark overlay. A small white circle is visible on the left side of the text.

# Conway's law in action

# The cost of a generic solution

- Going generic may save time in the long run, but at which price?
- Another rule of three: building reusable components is 3x as difficult as single use
- The price you pay is coupling
  - Both on code level and people/team level (communication overhead)

A photograph of two skateboarders on a desert road. The skateboarder on the left is wearing a pink t-shirt and dark pants, riding smoothly. The skateboarder on the right is wearing a yellow t-shirt and blue jeans, falling off their board. The background is a vast, arid landscape with rolling hills and a clear sky. The text "What if you get it wrong?" is overlaid in white in the center of the image.

What if you get it wrong?

# The cost of abstractions

- There are no zero cost abstractions
- Efficiency gains of a generic solution are typically clear, but how about:
  - Onboarding new people
  - Readability
  - Coupling

# The cost of abstractions

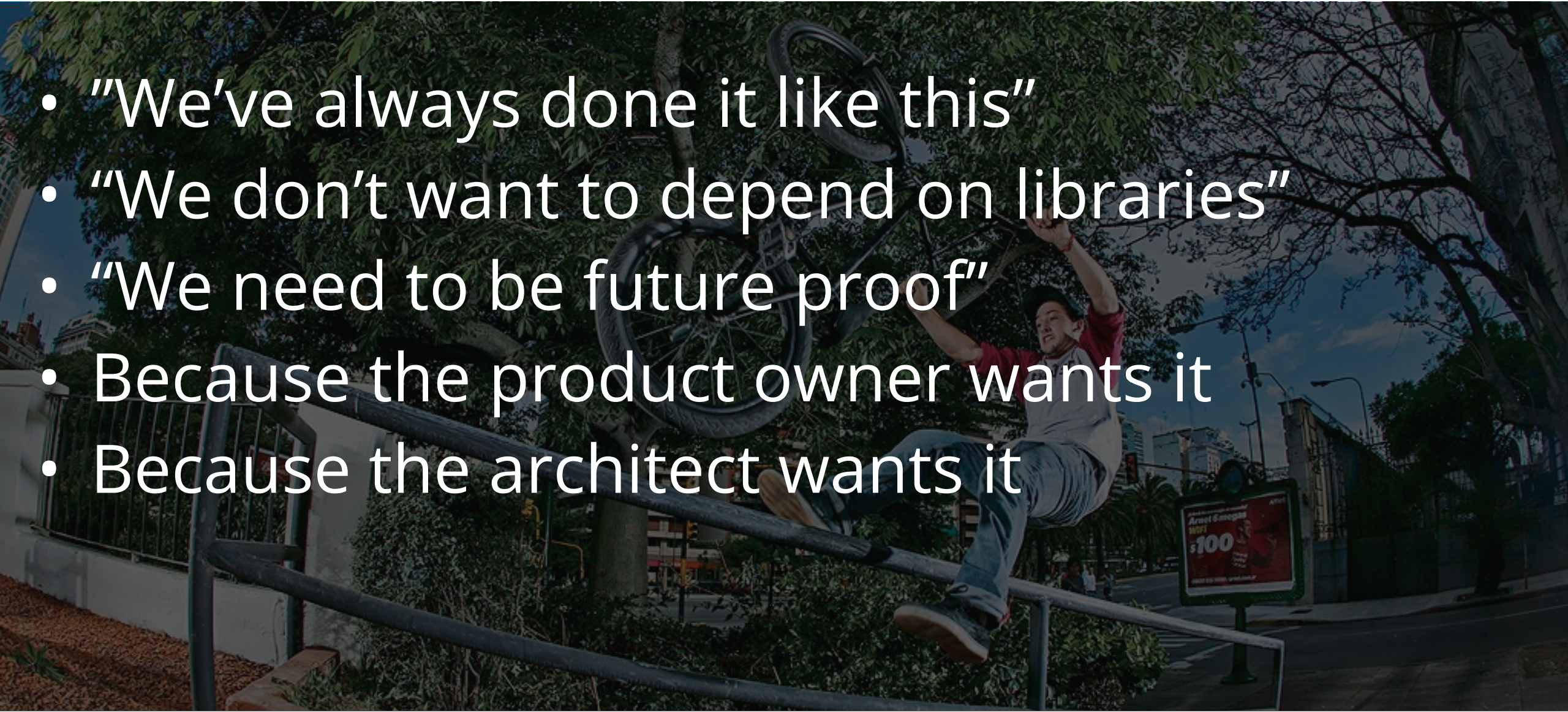
- Writing bad abstractions
  - Writing unnecessary reusable code
  - Introducing unnecessary coupling
- Maintaining bad abstractions
  - Hard to see
  - Hard to understand
  - Hard to extend



When / why to go generic

# Bad reasons to go generic

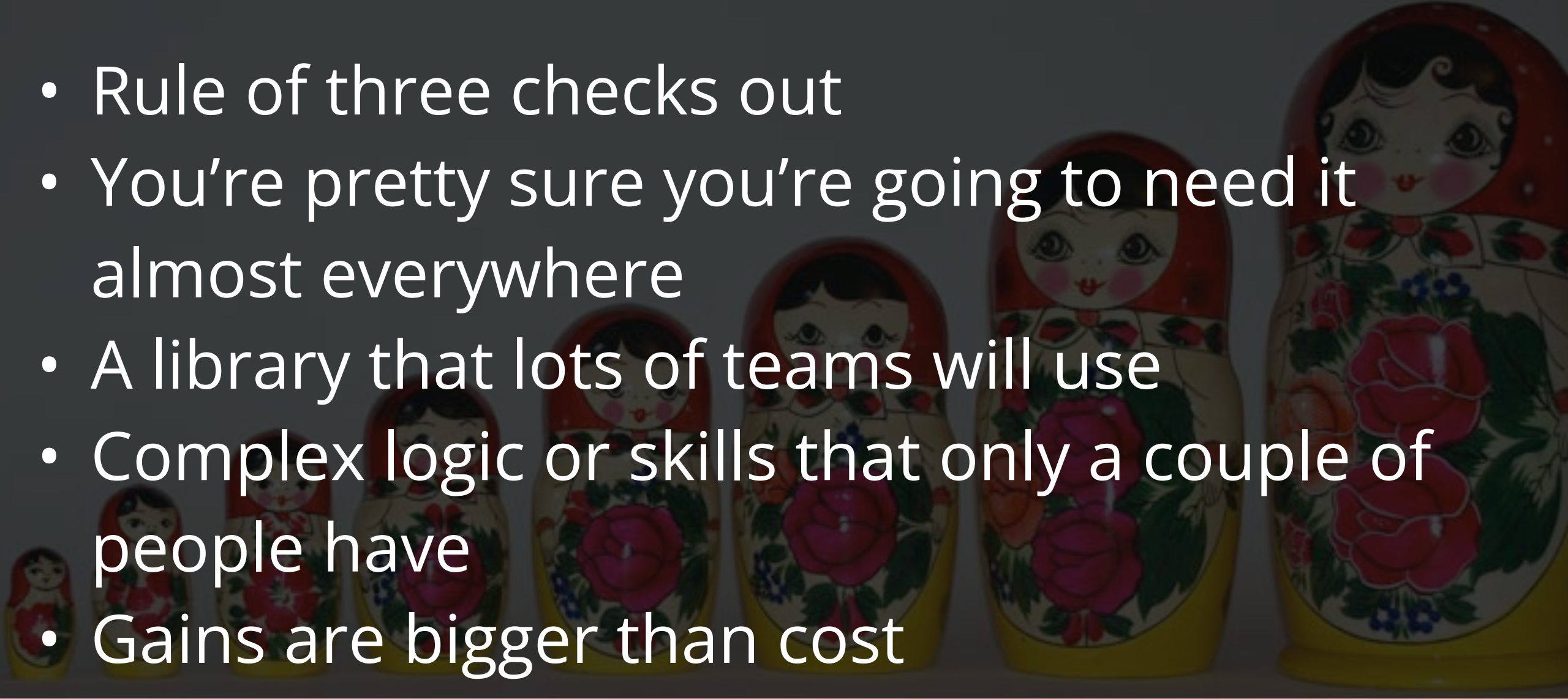
- "We've always done it like this"
- "We don't want to depend on libraries"
- "We need to be future proof"
- Because the product owner wants it
- Because the architect wants it





# Valid reasons to go generic

- Rule of three checks out
- You're pretty sure you're going to need it almost everywhere
- A library that lots of teams will use
- Complex logic or skills that only a couple of people have
- Gains are bigger than cost



# Generic vs specific in different scopes

- Think back about the layers in hierarchical decomposition of a system
- Code vs component vs service
- Are the considerations for generic vs specific the same on every level?
- Risk when getting it wrong is higher when the level is higher
- Don't confuse generification with standardization!



Why specific is often faster

A golfer in a bright green polo shirt, dark trousers, and a grey cap is captured in the middle of a golf swing on a lush green course. The golfer is wearing a light-colored glove on their left hand and a dark glove on their right. The background features a line of trees and a distant hill under a dramatic, overcast sky with heavy grey clouds. The text "Code golf" is overlaid in the center of the image.

Code golf

# Advent of code



### --- Day 5: Supply Stacks ---

The expedition can depart as soon as the final supplies have been unloaded from the ships. Supplies are stored in stacks of marked **crates**, but because the needed supplies are buried under many other crates, the crates need to be rearranged.

The ship has a **giant cargo crane** capable of moving crates between stacks. To ensure none of the crates get crushed or fall over, the crane operator will rearrange them in a series of carefully-planned steps. After the crates are rearranged, the desired crates will be at the top of each stack.

The Elves don't want to interrupt the crane operator during this delicate procedure, but they forgot to ask her **which** crate will end up where, and they want to be ready to unload them as soon as possible so they can embark.

They do, however, have a drawing of the starting stacks of crates **and** the rearrangement procedure (your puzzle input). For example:

```

    [D]
[N] [C]
[Z] [M] [P]
 1   2   3

move 1 from 2 to 1
move 3 from 1 to 3
move 2 from 2 to 1
move 1 from 1 to 2
```

[Q]	[J]							[H]
[G]	[S]	[Q]		[Z]				[P]
[P]	[F]	[M]		[F]		[F]		[S]
[R]	[R]	[P]	[F]	[V]		[D]		[L]
[L]	[W]	[W]	[D]	[W]	[S]	[V]		[G]
[C]	[H]	[H]	[T]	[D]	[L]	[M]	[B]	[B]
[T]	[Q]	[B]	[S]	[L]	[C]	[B]	[J]	[N]
[F]	[N]	[F]	[V]	[Q]	[Z]	[Z]	[T]	[Q]
1	2	3	4	5	6	7	8	9

move 1 from 8 to 1  
 move 1 from 6 to 1  
 move 3 from 7 to 4  
 move 3 from 2 to 9  
 move 11 from 9 to 3  
 move 1 from 6 to 9  
 move 15 from 3 to 9  
 move 5 from 2 to 3  
 move 3 from 7 to 5  
 move 6 from 9 to 3  
 move 6 from 1 to 6  
 move 2 from 3 to 7

```
Scanner scanner = new Scanner(input);
List<String> lines = new ArrayList<>();
List<String> instructions = new ArrayList<>();

// determine initial matrix width/height dimensions
int maxLineLength = 0, initialMatrixHeight = 0;
while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    lines.add(line);
    if (line.contains("[")) { initialMatrixHeight++; }
    if (line.endsWith("]") && line.length() > maxLineLength) { maxLineLength = line.length(); }
}
int initialMatrixWidth = (maxLineLength + 1) / 4;
Matrix matrix = new Matrix(initialMatrixWidth, initialMatrixHeight);

// init matrix and instruction
int y = 0;
for (String line: lines) {
    if (line.contains("[")) { // matrix line
        y++;
        for (int x=1; x<initialMatrixWidth+1; x++) {
            matrix.put(x, y: initialMatrixHeight-y+1, line.charAt(4*(x-1)+1));
        }
    } else if (line.startsWith("move")) { instructions.add(line); }
}
}
```



```
ArrayList<String> stack1 = new ArrayList<>(Arrays.asList("F", "D", "B", "Z", "T", "J", "R", "N"));
ArrayList<String> stack2 = new ArrayList<>(Arrays.asList("R", "S", "N", "J", "H"));
ArrayList<String> stack3 = new ArrayList<>(Arrays.asList("C", "R", "N", "J", "G", "Z", "F", "Q"));
ArrayList<String> stack4 = new ArrayList<>(Arrays.asList("F", "V", "N", "G", "R", "T", "Q"));
ArrayList<String> stack5 = new ArrayList<>(Arrays.asList("L", "T", "Q", "F"));
ArrayList<String> stack6 = new ArrayList<>(Arrays.asList("Q", "C", "W", "Z", "B", "R", "G", "N"));
ArrayList<String> stack7 = new ArrayList<>(Arrays.asList("F", "C", "L", "S", "N", "H", "M"));
ArrayList<String> stack8 = new ArrayList<>(Arrays.asList("D", "N", "Q", "M", "T", "J"));
ArrayList<String> stack9 = new ArrayList<>(Arrays.asList("P", "G", "S"));
```

An overhead, top-down view of a modern office environment. The office is furnished with several round, light-colored wooden tables. People are seated around these tables, many of whom are using laptops. The individuals are dressed in casual to business-casual attire. The floor is a light-colored, polished surface. In the background, there are some office supplies, a blue bag, and a bulletin board with various papers and colorful sticky notes pinned to it. The overall atmosphere is one of a collaborative and active workspace.

Generic solutions on organization level

# Sharing code within an organization

- Sharing code efficiently at scale is hard
- Sharing code at scale means:
  - Multiple modules that share code
  - Multiple team members
  - High rate of change
  - Little to no loss of individual productivity

# Sharing code within an organization

- Challenges:
  - Refactoring
  - Versioning
  - Reviewing
  - Builds and codebase size - monorepo?

# Monorepositories



- Monorepo: 1 large repository for a group of projects (possible all projects)
- Good: easy to make changes across projects
- Bad: dependencies & build times

# Considerations on sharing code in an org

- Discovery: what code / libraries exist?
- Distribution: binary or source dependency?
- Import: well defined API's or chaos?
- Versioning, upgrades and lifecycle management
- Who maintains it?
- Possible approach: inner source culture

Success Ln

Summary

Failure Dr

# Generic or specific?

- Consider:
  - YAGNI / Rule of three
  - Cost of generic
  - Scope / level
  - Conway's law
  - Organization



Success Ln

Generic or specific?

It depends.

Failure Dr

Write. simple. code.





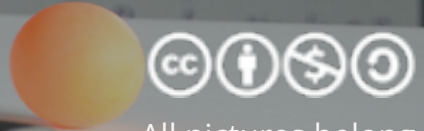
**THAT'S IT.  
NOW GO KICK SOME ASS!**



Questions?

Thanks for your time.

*Got feedback? Tweet it!*



All pictures belong to their respective authors

