Software Engineering

Understanding the Problem and Dividing the Work

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with
a subject like: SE-*id-Andy: On What …*

Download from c.program@yeah.net
/文件中心/网盘/SoftwareEngineering2024

Topics

- Labor Division vs. Expertise Division
- Reducing Inter-team Communication
- Decomposition by Partitioning vs. Projection

Key message:

Divide the work by the problems that you want to solve, not by the expertise in software tools:

> The problem provides the purpose to the expert knowledge.
> Expert knowledge is a resource to be managed, based on the problem needs.

Why We Want To Subdivide Problems/Projects

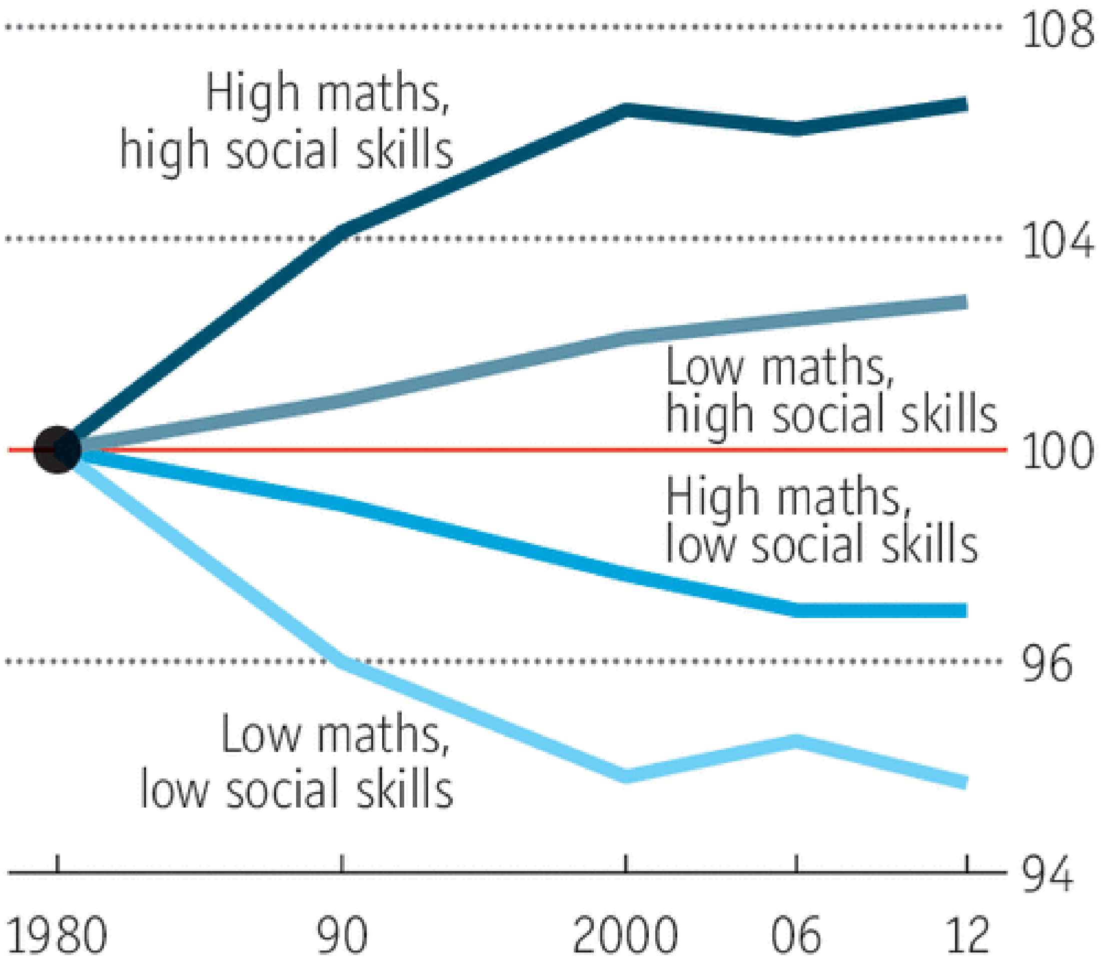
- "…main value [of social skills] lies in the relationship between colleagues: *people who can divide up tasks quickly and effectively between them form more productive teams*."
  —David Deming (Harvard U. & NBER)
- Common problem-solving strategy:
  "divide-and-conquer"
- Division of labor vs. division of the knowledge within the developers team
  - It may be that a solution cannot be made by a single person ("hands limitation") or cannot be comprehended by a single person ("mind limitation")
  - Different team members can focus on different sub-problems and not worry about everything
  - Different ways of dividing the work—which are better or best?
    - This depends on the problem, which can be discovered only by iterating creative and evaluative steps
      - Therefore the developer team must be **agile**—ready to reconfigure and reassign the work as insights are gained from the experience of testing their ideas in practice
- Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately ("separation of concerns")

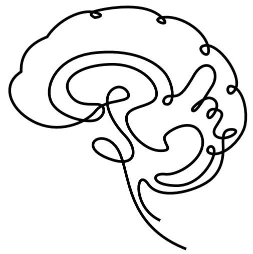
"When we think what we might have accomplished if we had been able to devote this time to experiments, we feel very sad, but **it is always easier to deal with things than with men**, and no one can direct his life entirely as he would choose."
—Wilbur Wright

Getting along and getting on

US, change in employment share

By skills required, 1980=100



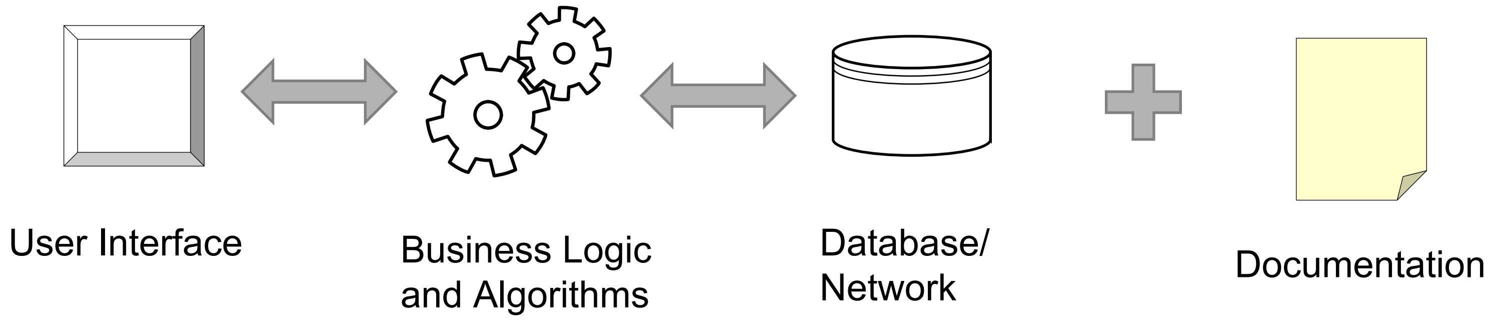


Source: "The Growing Importance of Social Skills in the Labor Market", by David Deming, Aug 2016

Economist.com

How Beginners Divide the Work

- Contrive a solution and divide the work by parts of the solution (based on team knowledge)
- Each sub-group of ≥1 is assigned a different part
  - One sub-group is assigned to “write documentation”



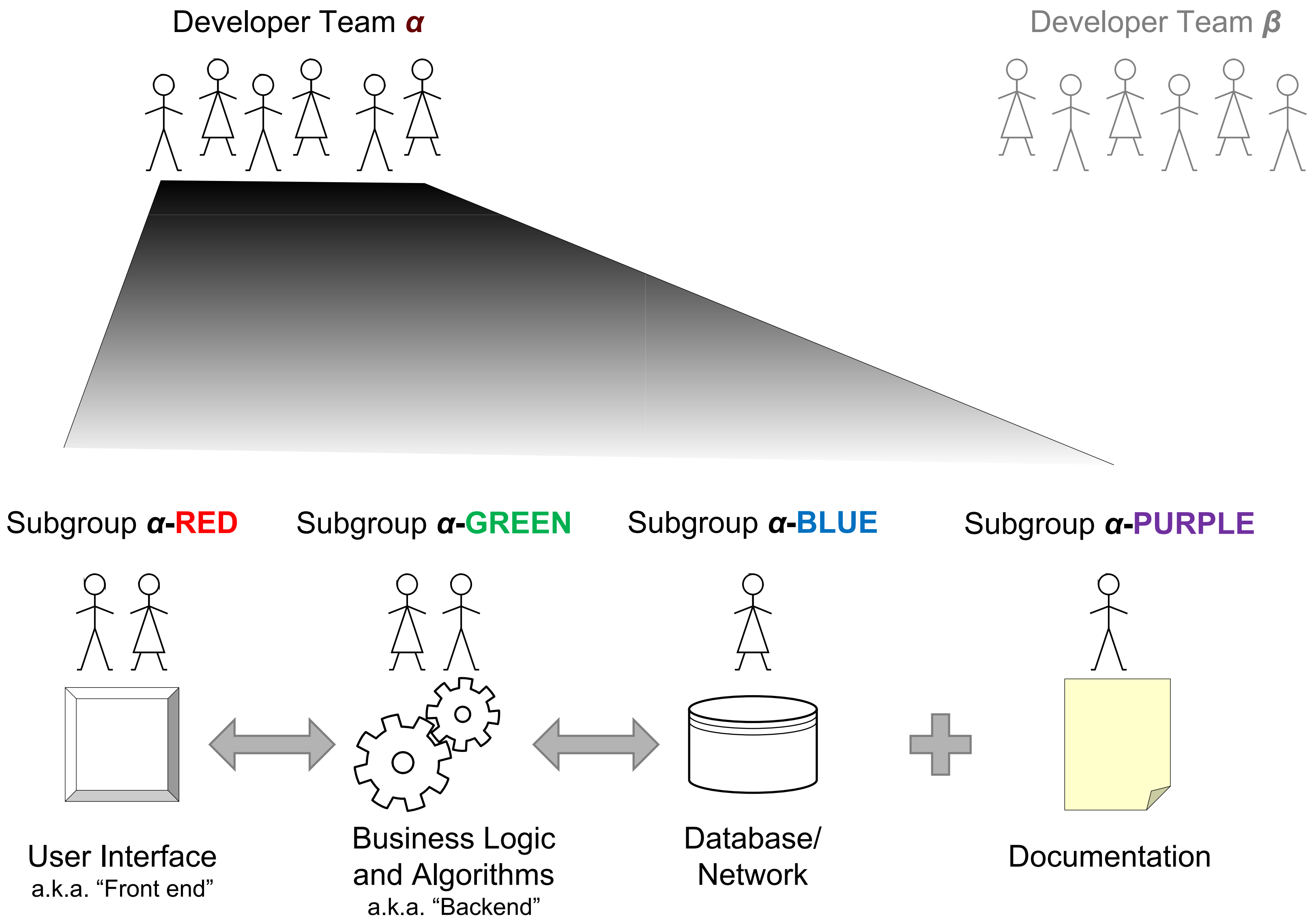
- “Chain” organization—every link is critical
  If any link fails, the whole project fails
  - Separation of concerns is impossible

Example: Restaurant Automation Requirements

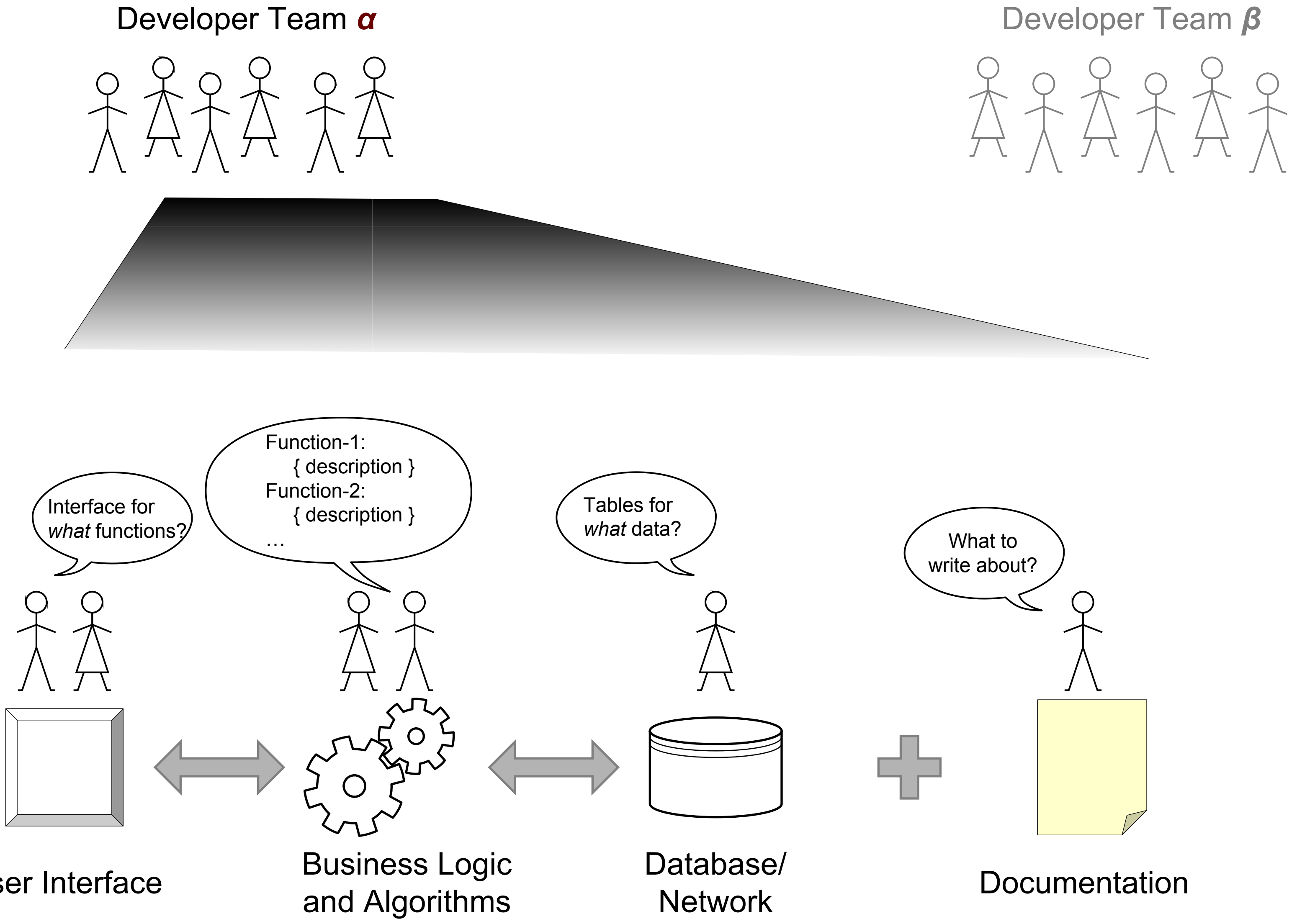
- REQ-1: Support the work of **wait & floor staff**
- REQ-2: Support the work of **kitchen staff**
- REQ-3: Support the work of **management**

Example: Restaurant Automation

Decomposition by Partitioning the Solution

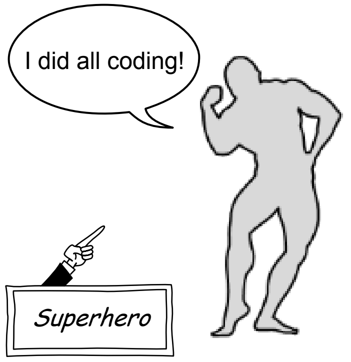


Example: Restaurant Automation
Communication Overhead

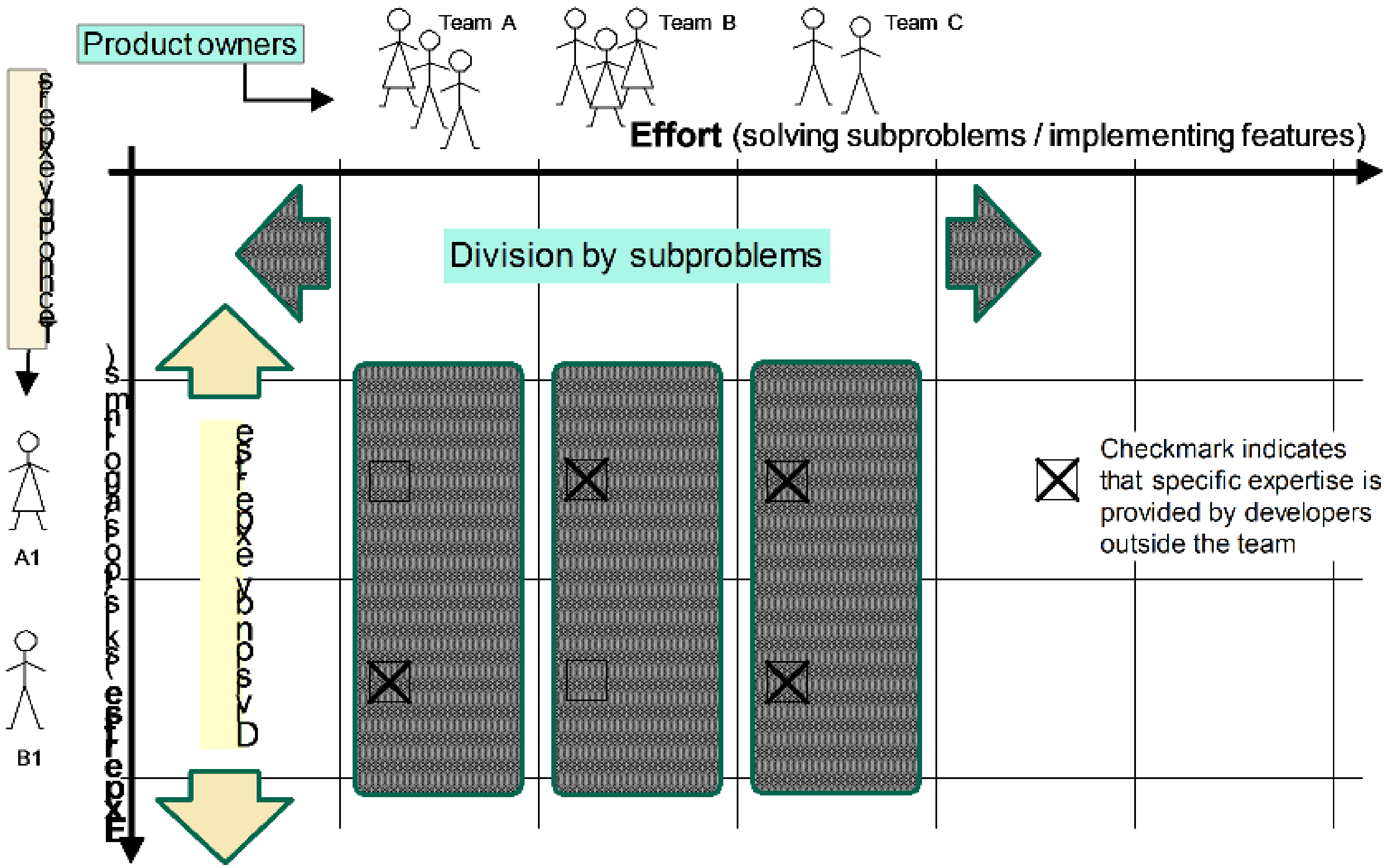


Example: Restaurant Automation
Drawbacks of Team α Approach

- ❑ Original problem still undivided:
  only the "tools" have been distributed with which to attack the unbroken monolith of the problem
- ❑ Their "architecture" is hastily contrived to divide the work, rather than carefully designed based on problem analysis
- ❑ Long feedback loop:
  - The time to implementation depends on the slowest sub-group
- ❑ Communication overhead:
  - Most time spent in communication and "documentation writing"
  - No time for creative software development
- ❑ Failure-prone:
  - If any sub-group doesn't deliver, the whole project fails
- ❑ Each student learns only one aspect of software development
- ❑ Ownership fuzzy or one "leader"
  claims all the credit

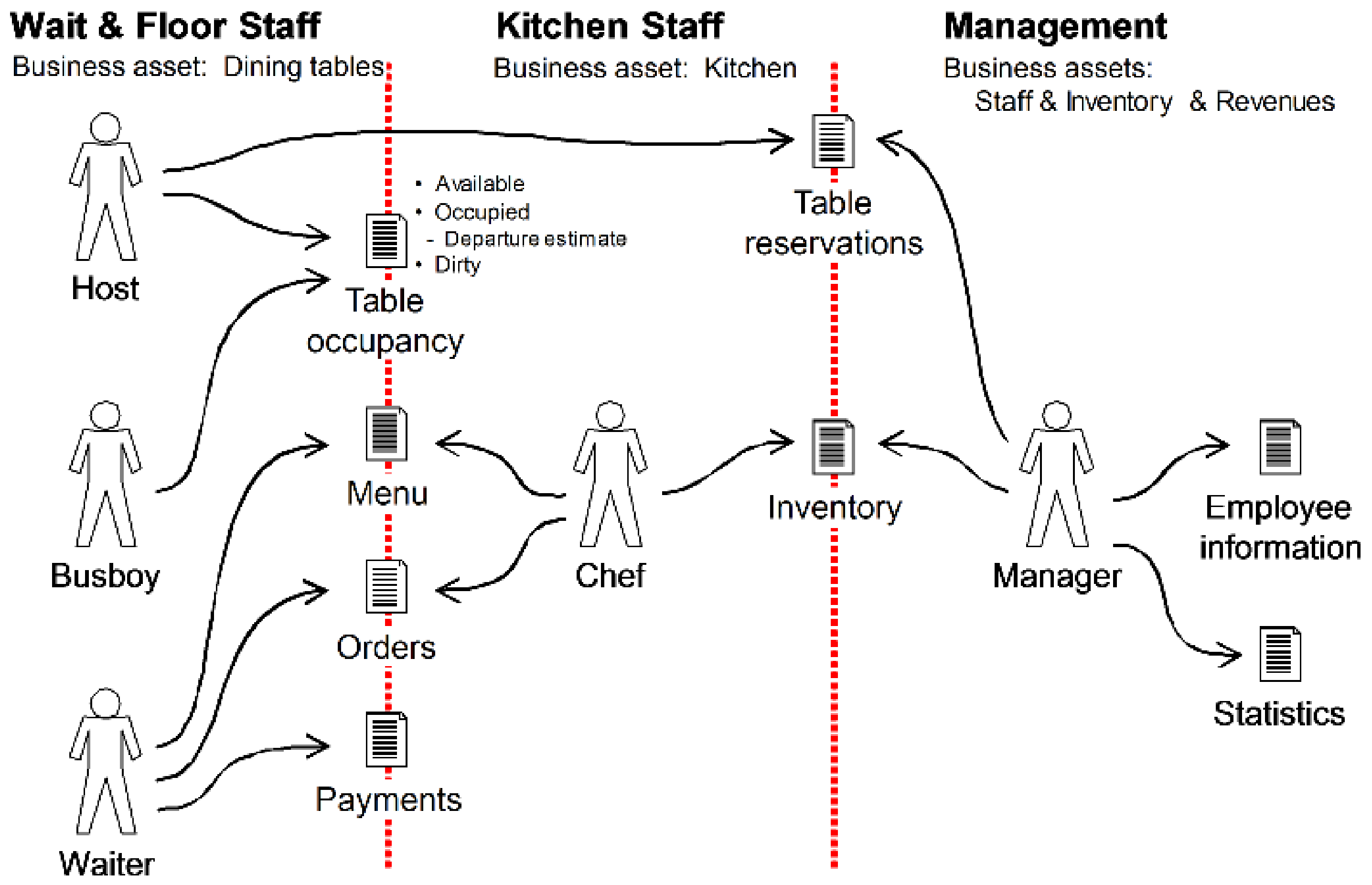


Effort Division vs. Expertise Division


Product owners
Team A
Team B
Team C
Effort (solving subproblems / implementing features)
Division by subproblems
Division by expertise
A1
B1
Checkmark indicates that specific expertise is provided by developers outside the team

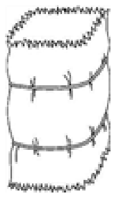
Identifying Subproblems by Managed Business Assets

Example: Restaurant Automation



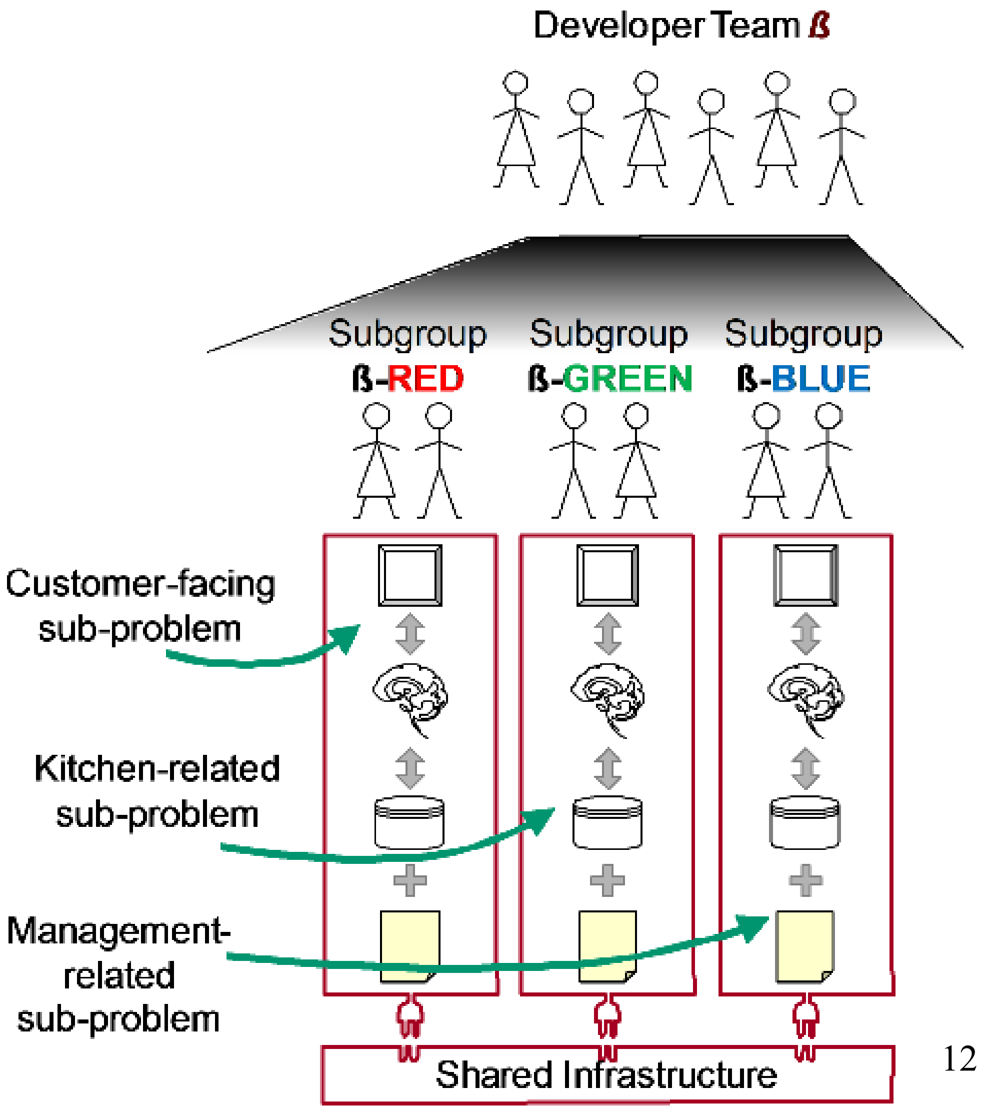
Example: Restaurant Automation
Decomposition by Projection to Sub-problems

- Different way of "slicing" the project:
  - Instead of horizontal slicing (by solution), we have vertical slicing— "stacks" (by sub-problems)



- Each "stack" can be done independently of other stacks, as a mini-project, *because it solves a different sub-problem*!

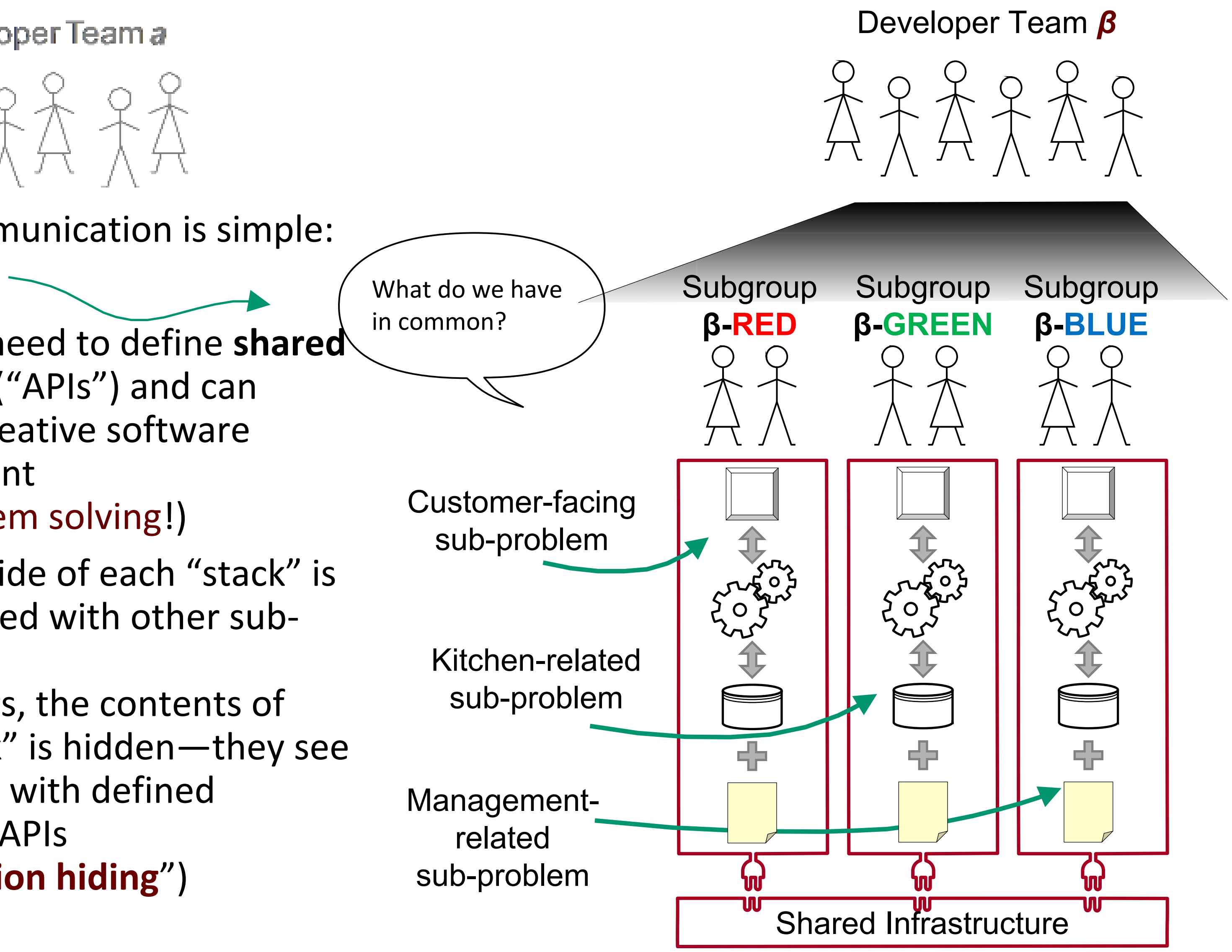
➔ Separation of concerns!



Developer Team ß
Subgroup ß-RED
Subgroup ß-GREEN
Subgroup ß-BLUE
Customer-facing sub-problem
Kitchen-related sub-problem
Management-related sub-problem
Shared Infrastructure

Example: Restaurant Automation
Decomposition by Projection

- Team communication is simple:
- They only need to define **shared interfaces** ("APIs") and can focus on creative software development (sub-problem solving!)
- What is inside of each "stack" is not discussed with other sub-teams —for others, the contents of your "stack" is hidden—they see a black box with defined interface / APIs ("**information hiding**")


What do we have in common?
Developer Team β
Subgroup β-RED
Subgroup β-GREEN
Subgroup β-BLUE
Customer-facing sub-problem
Kitchen-related sub-problem
Management-related sub-problem
Shared Infrastructure

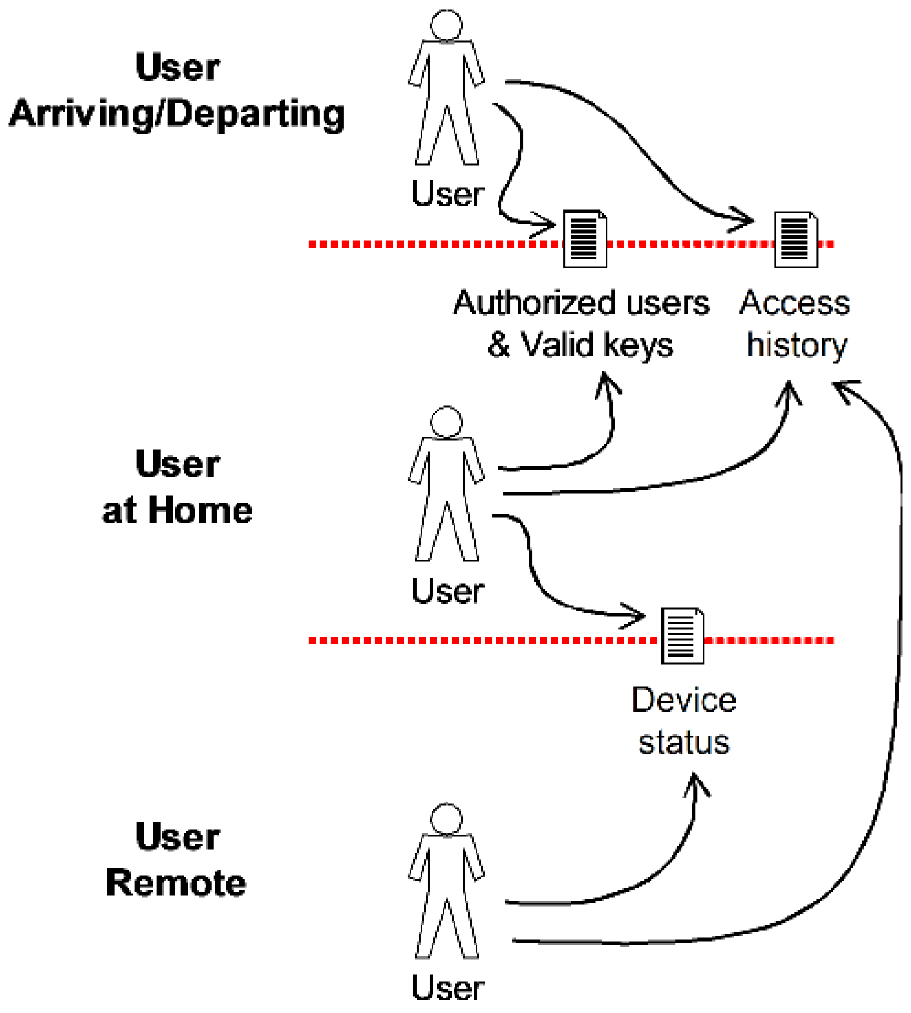
Example: Restaurant Automation
Benefits of Team β Approach

Organizing a team by smaller problems that will be solved (features developed)
rather than by the solution (implementation technologies):

- The problems are more likely to *remain stable* than the solutions
- Short feedback times
  - Every sub-group can have implementation feedback as soon as their part is working
- Increased productivity:
Minimizes the amount of communication needed to coordinate the work
  - → more time to focus on creative software development
- Failure resilience:
Reduces the dependency on the rest of the team—if they fail to deliver, you can still succeed and demonstrate your mini-project
(Unlike chain-organization, where a link failure leads to project failure, each “stack” can succeed independently of others, *because it solves customer’s sub-problem*)
- Each student learns **all** aspects of software development
- Ownership visibility:
Each mini-project can be demonstrated alone (because it solves customer’s sub-problem)
(If the work is organized around solution expertise, who demonstrates UI, or dBase, or …?)
- Customer friendly:
The customer (or class instructor) knows whom to talk about every aspect of a given functional feature (i.e., about a sub-problem that is of interest to them)
- Requirements traceability, from inception to implementation

Identifying Subproblems by User Context

Example: Safe Home Access



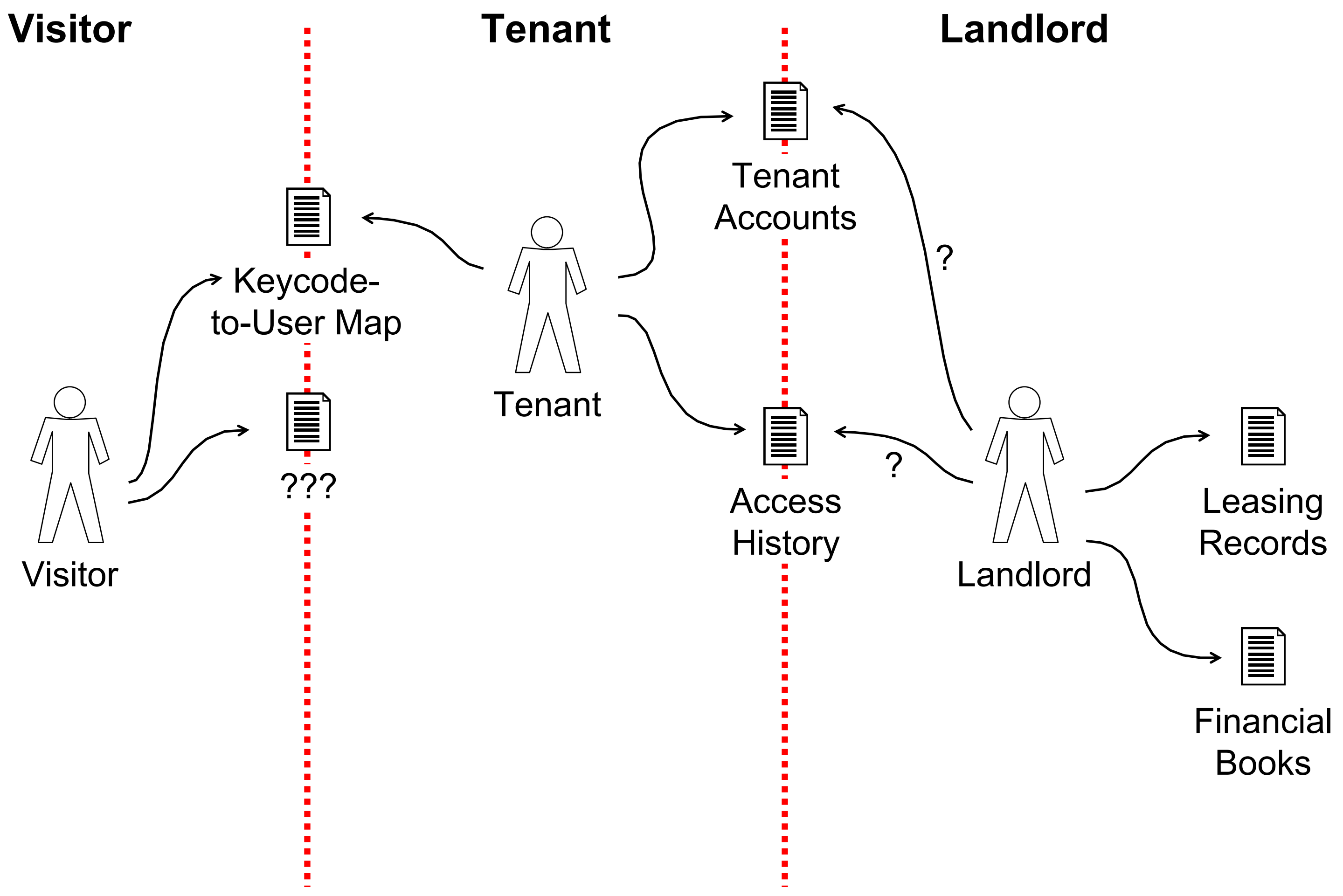
Rationale:

If the developer team's structure is mirroring the product structure,

then the product structure should also be mirroring the structure of the user contexts whose work this product will support

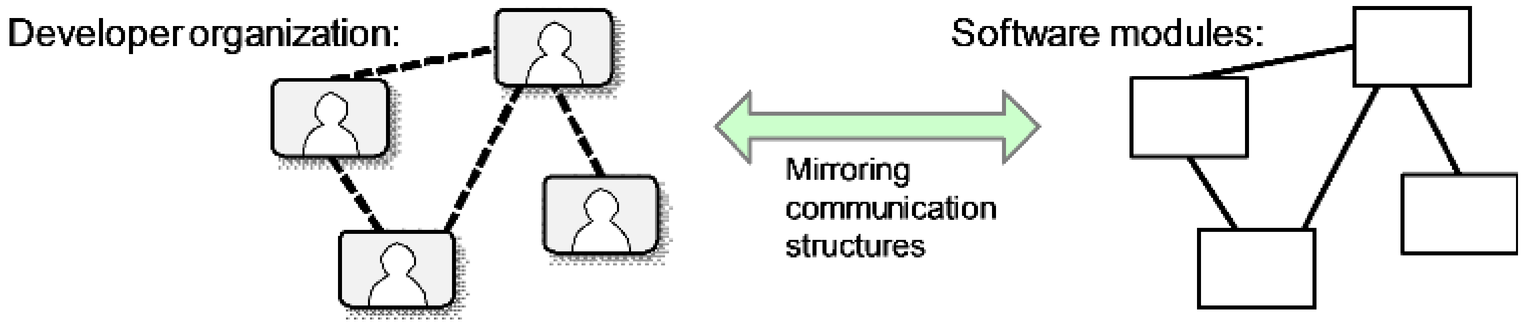
Identifying Subproblems by User Roles

Example: Safe Home Access



Conway's Law a.k.a. Mirroring Hypothesis

- ❑ M. Conway (1968):
  - "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

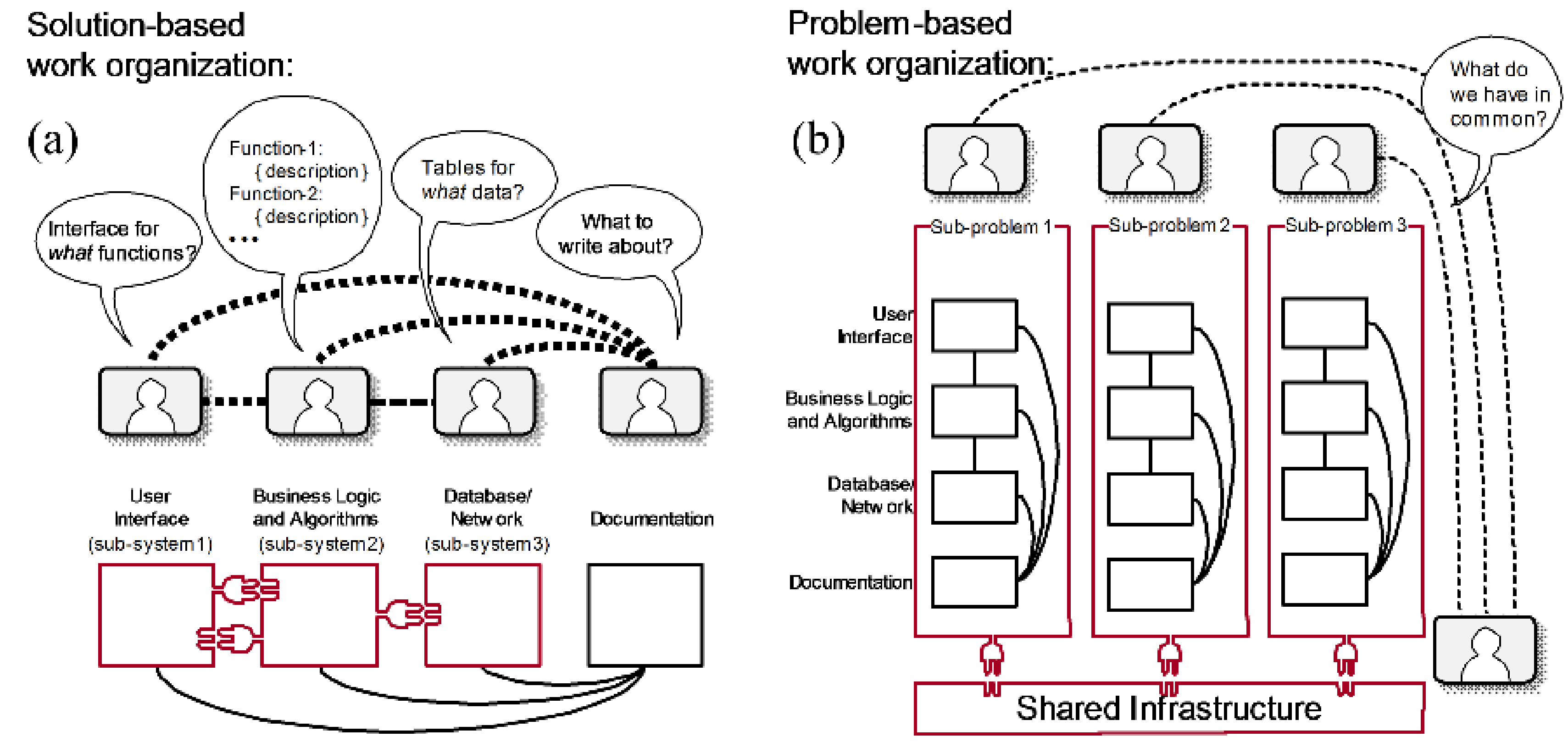


- ❑ Corollary for identifying subproblems/modules:
  - If the developer team's structure is mirroring the product structure, then the product structure should also be mirroring the structure of the user group that will use this product
    - Spatial structure: different user roles and their relationships
    - Temporal structure: user (or users) in different contexts over time

Why Communication Required by Problem-based < Solution-based ?

Recall **Conway's Law**:

developer's communication patterns and system modules' communication patterns will mirror each other



Solution-based work organization:
(a)
Interface for *what* functions?
Function-1: {description} Function-2: {description} ...
Tables for *what* data?
What to write about?
User Interface (sub-system1)
Business Logic and Algorithms (sub-system2)
Database/ Network (sub-system3)
Documentation
Problem-based work organization:
(b)
What do we have in common?
Sub-problem 1
Sub-problem 2
Sub-problem 3
User Interface
Business Logic and Algorithms
Database/ Network
Documentation
Shared Infrastructure

What is "Shared Infrastructure"?

- Needs to be discovered, as we will learn in subsequent lectures
- Examples of shared infrastructure:
  - A relational database, so the whole team needs to design the shared tables
  - Data items communicated via messages, so the whole team needs to discuss the communication protocol and message format
  - "Services" accessed by method calls, e.g., data analysis, pattern recognition, user authentication, etc.
  - Software configuration management (SCM) system for the entire project

Divide Work by Problem vs. by Solution

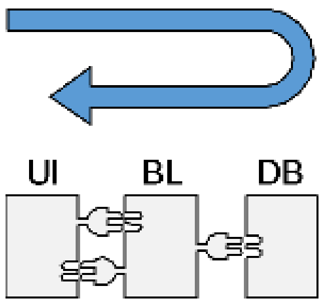
- **By problem:** Different team members responsible to solve different sub-problems or develop different "features" of the system
  - **Suitable for any level of expertise, particularly if uncertain about other team members' skills and ability to deliver on time (➔ highly recommended for novices/students)**
  - ***Advantages*: Short time to feedback from implementation, mutually independent between the sub-groups Dependency on other team members is reduced—the development of different features can progress independently, at their own pace**
  - ***Drawback*: Potential redundancies because different teammates may need to develop same/similar parts to make their "feature" work independently of others**

- **By solution:** Different team members responsible for different parts of the system
  - ***Advantage*: everyone can focus on one *technology (area of expertise)***
  - ***Drawbacks*:**
    - **The original problem is still an undivided whole; the "modules" are just different aspects of the monolith**
    - **Long time to implementation and lack of early feedback**
    - **Developers need to know a lot about the solution (i.e., parts of the "system" and their relationships) before detailed analysis of the sub-problems.**
    - **If solution parts are not precisely specified, there will be a great uncertainty about interfacing the parts and integrating into the whole system.**
  - **Suitable for highly-experienced specialists, not for beginners (i.e., not for students), working on a familiar problem**
  - **Always watch for fallback solutions in case some parts are not ready on time or problematic**
  - **Right now, you have no "area of expertise"—otherwise you wouldn't be a student in this class!**

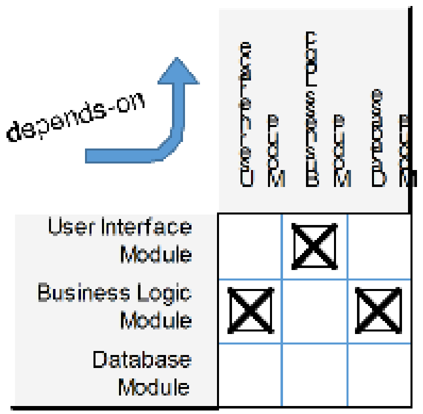
Design Parameter Dependencies

Design Structure Matrix

- Design Structure Matrix (DSM)
  - Shows which modules depend on (or, are allowed to use) which other modules
- Any change in the choices for design parameters can potentially affect the dependent parameters
  - It may or may not, but because different sub-teams cannot be sure, they need to communicate and find out

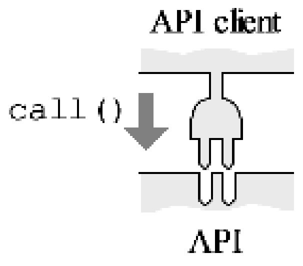


(a)

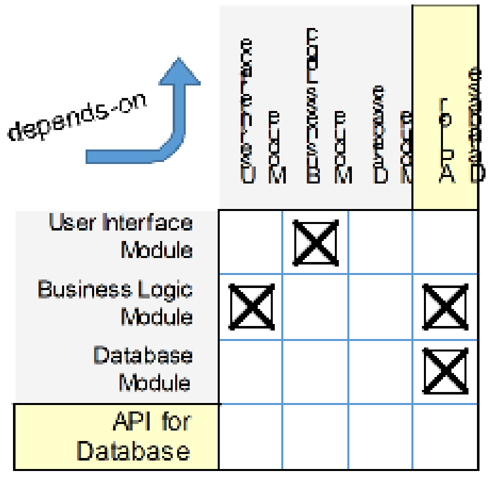


User Interface Module		☒	
Business Logic Module	☒		☒
Database Module			

(b)



(c)

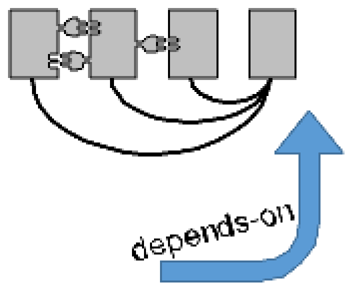


User Interface Module		☒		
Business Logic Module	☒			☒
Database Module				☒
API for Database				

(b)

Design Parameter Dependencies

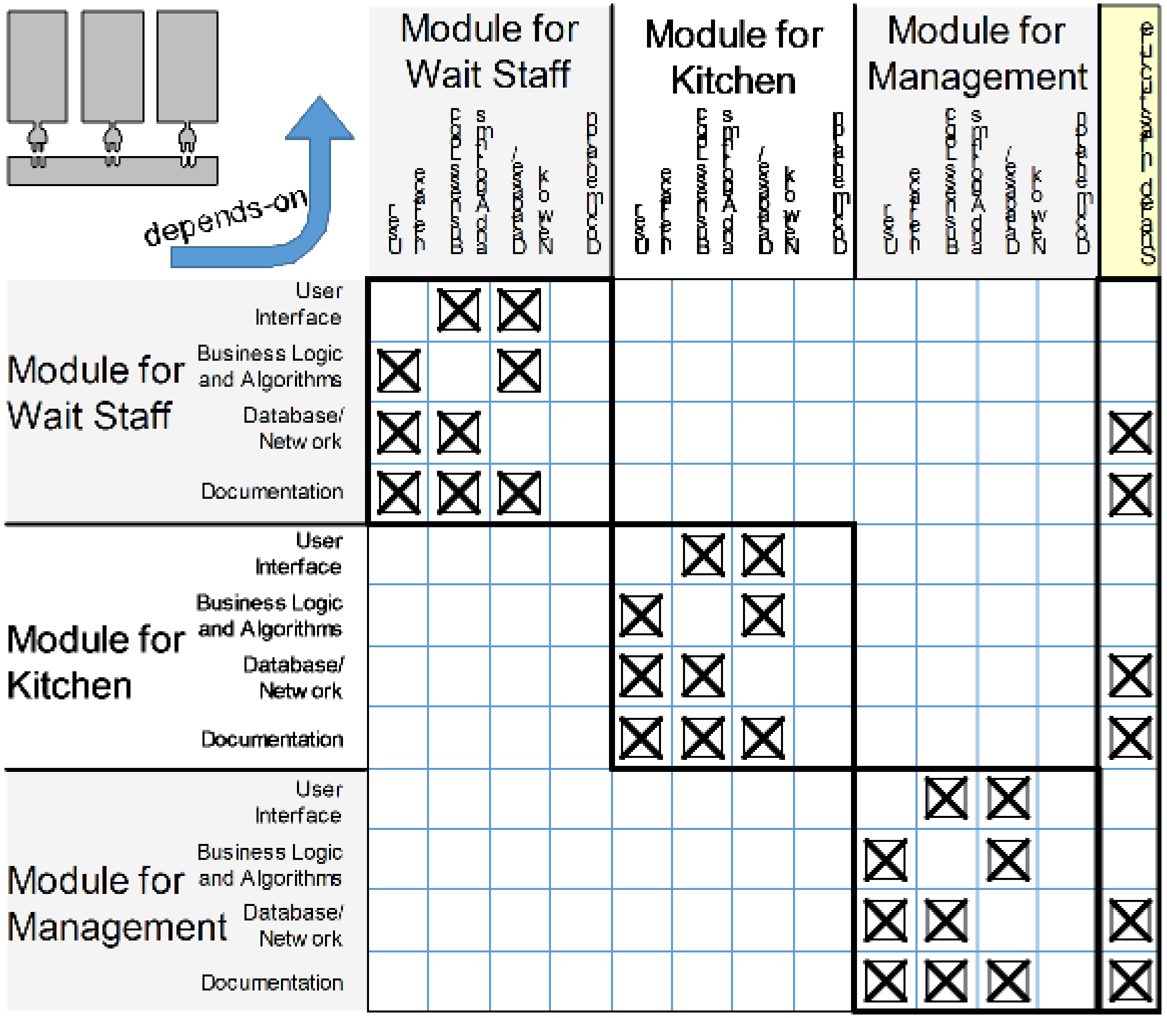
Work Assignment by *Solution* Parts



		User Interface Module			Business Logic Module			Database Module			APIs for Modules		
		UI for Wait Staff	UI for Kitchen	UI for Management	BL for Wait Staff	BL for Kitchen	BL for Management	DB for Wait Staff	DB for Kitchen	DB for Management	API for User Interface Module	API for Business Logic Module	API for Database Module
User Interface Module	UI for Wait Staff		☒	☒							☒	☒	
	UI for Kitchen	☒		☒							☒	☒	
	UI for Management	☒	☒								☒	☒	
Business Logic Module	BL for Wait Staff					☒	☒				☒	☒	☒
	BL for Kitchen				☒		☒				☒	☒	☒
	BL for Management				☒	☒					☒	☒	☒
Database Module	DB for Wait Staff								☒	☒			☒
	DB for Kitchen							☒		☒			☒
	DB for Management							☒	☒				☒
	API for User Interface Module												
	API for Business Logic Module												
	API for Database Module												
Documentation		☒	☒	☒	☒	☒	☒	☒	☒	☒	☒	☒	☒

Design Parameter Dependencies

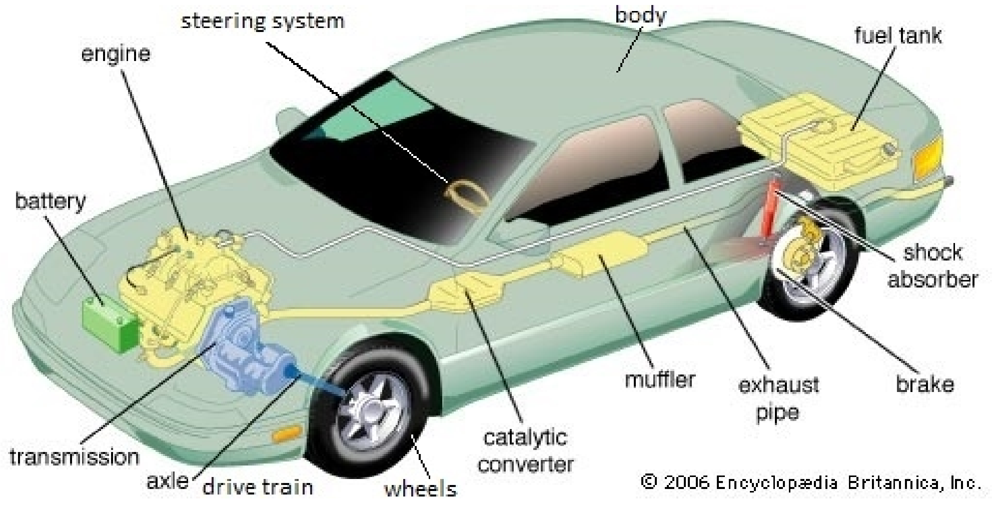
Work Assignment by *Problem* Parts



		Module for Wait Staff: User Interface	Module for Wait Staff: Business Logic and Algorithms	Module for Wait Staff: Database/Network	Module for Wait Staff: Documentation	Module for Kitchen: User Interface	Module for Kitchen: Business Logic and Algorithms	Module for Kitchen: Database/Network	Module for Kitchen: Documentation	Module for Management: User Interface	Module for Management: Business Logic and Algorithms	Module for Management: Database/Network	Module for Management: Documentation	Search Engine
Module for Wait Staff	User Interface		☒	☒										
	Business Logic and Algorithms	☒		☒										
	Database/Network	☒	☒											☒
	Documentation	☒	☒	☒										☒
Module for Kitchen	User Interface						☒	☒						
	Business Logic and Algorithms					☒		☒						
	Database/Network					☒	☒							☒
	Documentation					☒	☒	☒						☒
Module for Management	User Interface										☒	☒		
	Business Logic and Algorithms									☒		☒		
	Database/Network									☒	☒			☒
	Documentation									☒	☒	☒		☒

Counterexample?

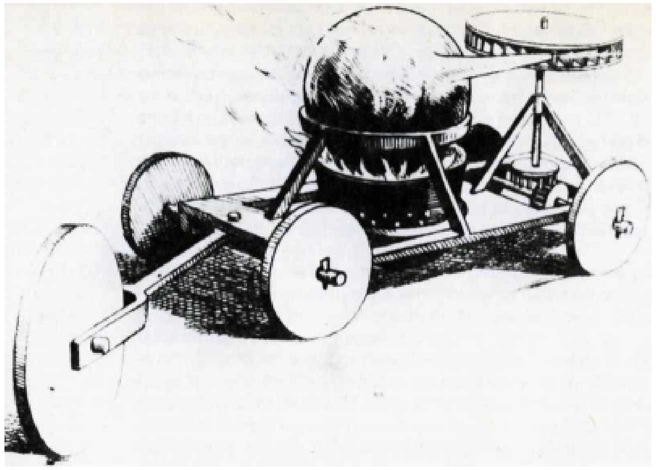
- Car subsystems -- different companies produce different subsystems!
- Why cannot we do the same?!

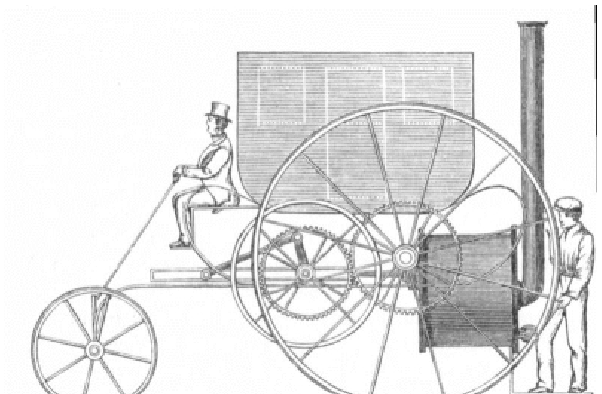


- Because you are not at this level of understanding of your problem
  - They did all the work of analysis and design and arrived at commonly-accepted standards of subsystem designs!

Counterexample Rebuttal (1)

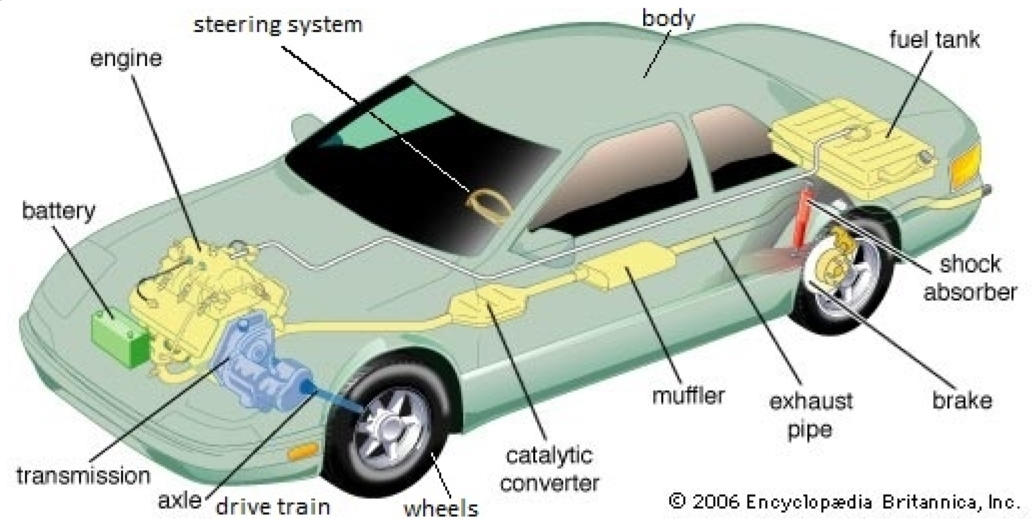
- Actually, you are at this level of understanding of your problem:





Counterexample Rebuttal (2)

Before we could do this…



…there was a lot of trial-and-error
and each subsystem ***evolved independently***
to solve a different sub-problem

➔ Interfaces between the parts defined by precise industry standards
➔ Reliable 3rd-party providers of standard-defined subsystems!