

Lesson 4

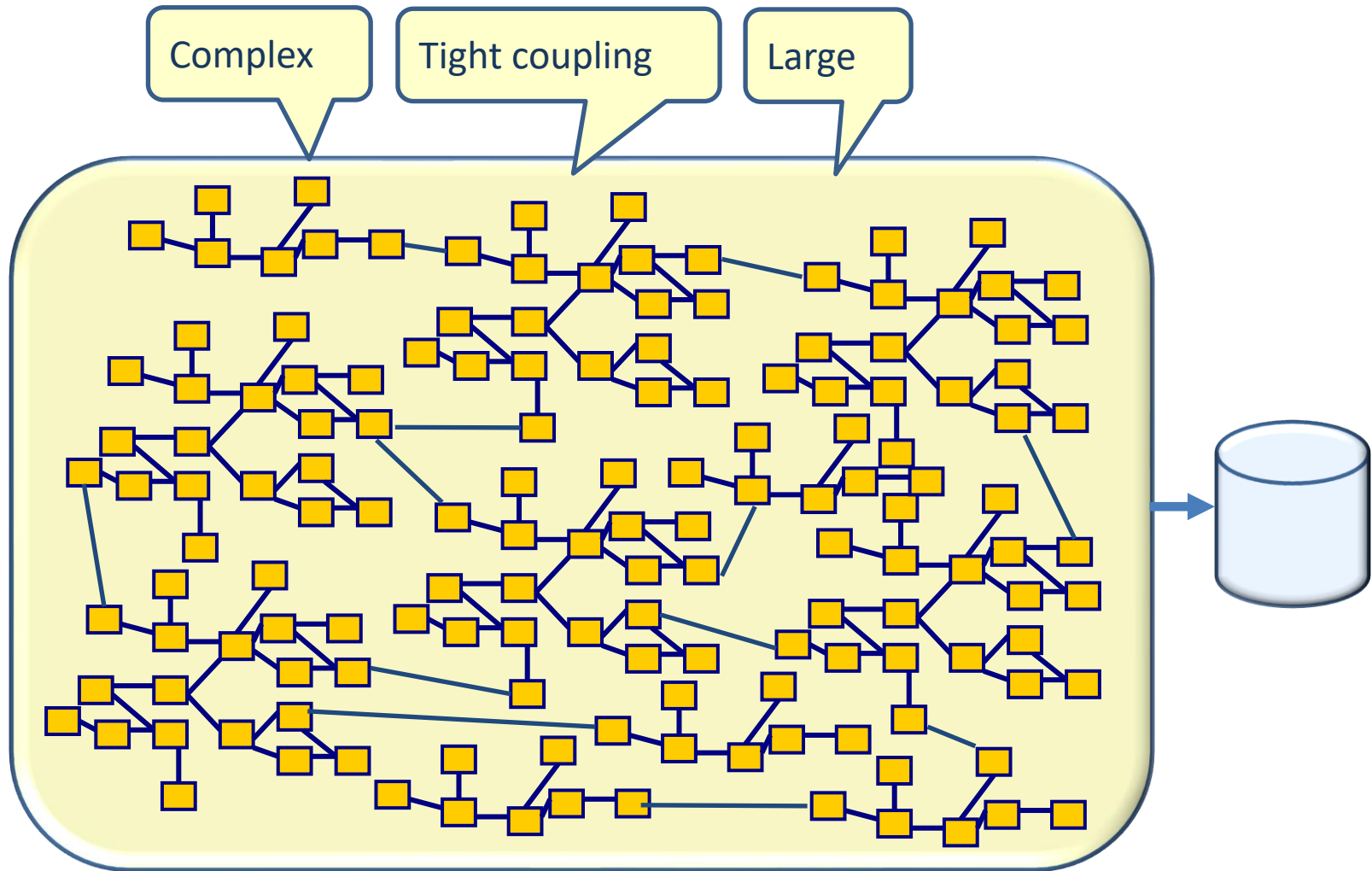
COMPONENT BASED DESIGN



PROBLEM OF 1 LARGE OO APPLICATION

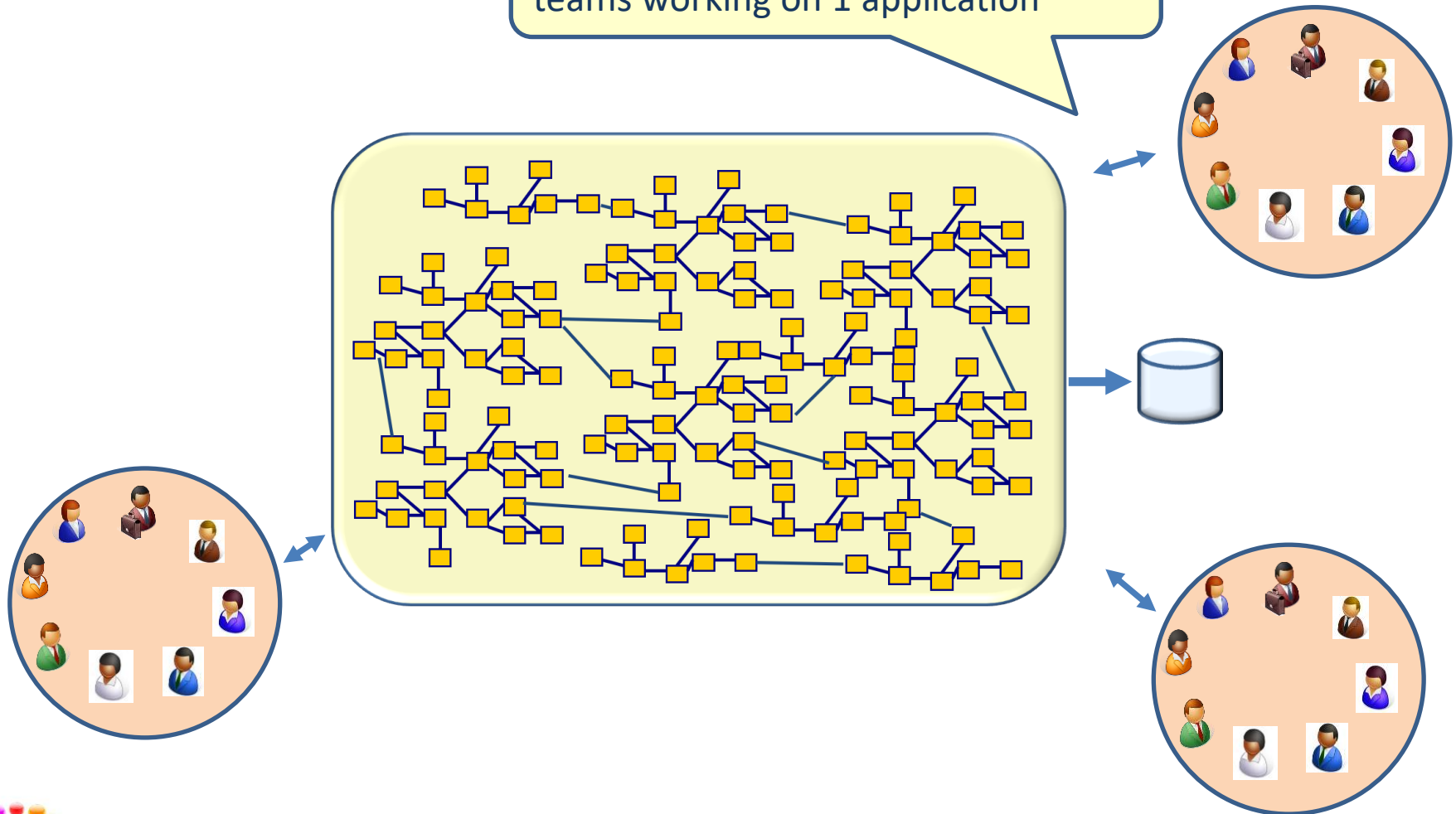


Object orientation

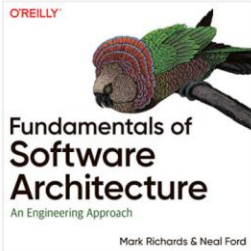


1 large monolith application

Very difficult to have multiple scrum teams working on 1 application



Amazon.com



Fundamentals of Software Architecture
An Engineering Approach
Mark Richards & Neal Ford

Audible Sample

Fundamentals of Software Architecture: An Engineering Approach Audible Audiobook – Unabridged

Mark Richards (Author), Neal Ford (Author), Benjamin Lange (Narrator), Upfront Books (Publisher)

★★★★☆ 251 ratings

See all formats and editions

Kindle \$40.51	Audiobook \$0.00	Paperback \$53.41
-------------------	----------------------------	----------------------

Read with Our **Free App** Free with your Audible trial 16 Used from \$44.10 24 New from \$42.56

Salary surveys worldwide regularly place software architect in the top 10 best jobs, yet no real guide exists to help developers become architects. Until now. This book provides the first comprehensive overview of software architecture's many aspects. Aspiring and existing architects alike will examine architectural characteristics, architectural patterns, component determination, diagramming and presenting architecture, evolutionary architecture, and many other topics.

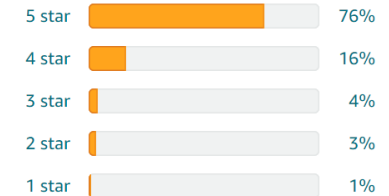
Mark Richards and Neal Ford - hands-on practitioners who have taught software architecture classes professionally for years - focus on

[Read more](#)

Customer reviews

★★★★☆ 4.6 out of 5

251 global ratings



How are ratings calculated?

People who viewed this also viewed

Page 1 of 8



Software Engineering at Google: Lessons Learned from Programming Over Time
Titus Winters
★★★★☆ 284
Audible Audiobook
\$0.00 Free with Audible trial



Monolith to Microservices: Evolutionary Patterns to Modernize Your Applications
Sam Newman
★★★★☆ 295
Audible Audiobook
\$0.00 Free with Audible trial



Clean Architecture: A Craftsman's Guide to Software Structure and Design
Robert C. Martin
★★★★☆ 1445
#1 Best Seller in Industrial Quality Control
Audible Audiobook
\$0.00 Free with Audible trial



Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
Martin Kleppmann
★★★★☆ 1822
#1 Best Seller in Enterprise Data Computing
Audible Audiobook
\$0.00 Free with Audible trial



Clean Agile: Back to Basics
Robert C. Martin
★★★★☆ 258
Audible Audiobook
\$0.00 Free with Audible trial




The Clean Coder: A Code of Conduct for Professional Programmers
Robert C. Martin
★★★★☆ 844
Audible Audiobook
\$0.00 Free with Audible trial




Clean Code: A Handbook of Agile Software Craftsmanship
Robert C. Martin
★★★★☆ 1200
Audible Audiobook
\$0.00 Free with Audible trial

People who bought this also bought


Page 1 of 5



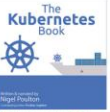
Grokking Algorithms
Aditya Bhargava
★★★★☆ 612
Audible Audiobook
\$0.00 Free with Audible trial



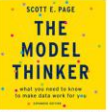
Staff Engineer: Leadership Beyond the Management Track
Will Larson
★★★★☆ 188
Audible Audiobook
\$0.00 Free with Audible trial




Domain-Driven Design: Tackling Complexity in the Heart of Software
Eric Evans
★★★★☆ 585
Audible Audiobook
\$0.00 Free with Audible trial




The Kubernetes Book
Nigel Poulton
★★★★☆ 592
Audible Audiobook
\$0.00 Free with Audible trial



The Model Thinker: What You Need to Know to Make Data Work for You
Scott E. Page
★★★★☆ 523
#1 Best Seller in Statistical Modeling
Audible Audiobook
\$0.00 Free with Audible trial



Microservices Security in Action
Prabath Siriwardena
★★★★☆ 13
Audible Audiobook
\$0.00 Free with Audible trial

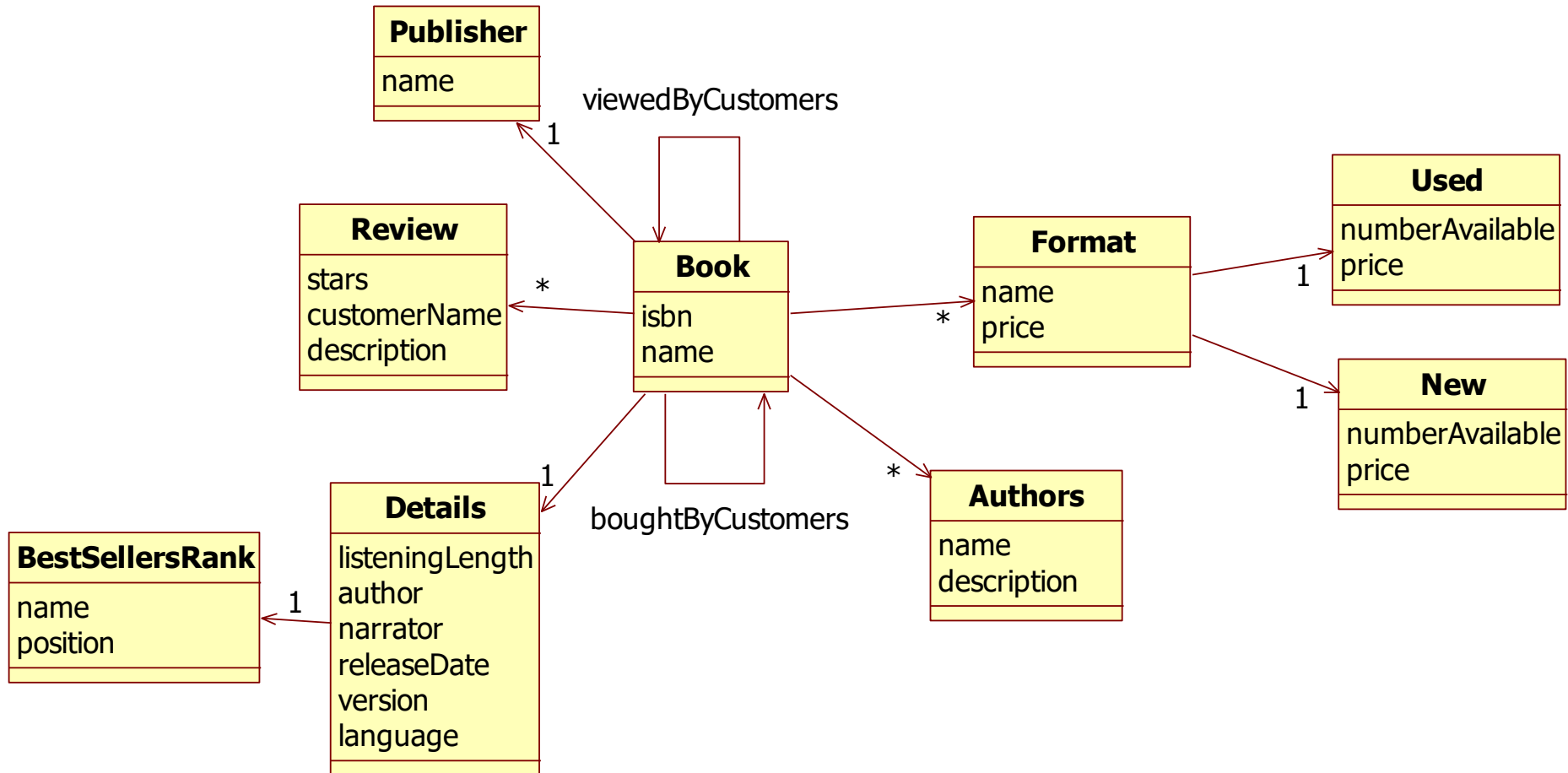


The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations
Gene Kim
★★★★☆ 1592
Audible Audiobook
\$0.00 Free with Audible trial

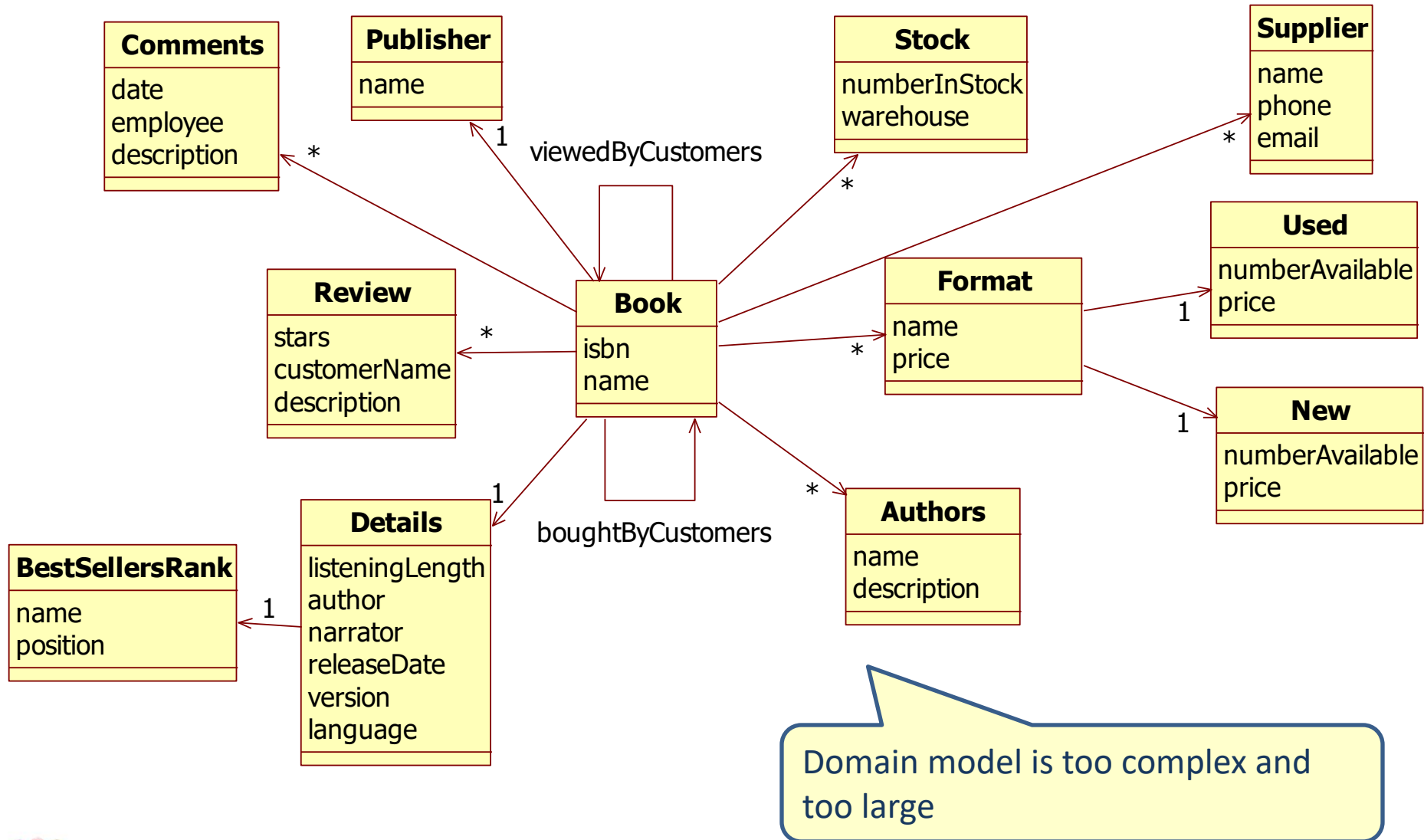
Product details

Listening Length	13 hours and 10 minutes
Author	Mark Richards, Neal Ford
Narrator	Benjamin Lange
Audible.com Release Date	February 27, 2021
Publisher	Upfront Books
Program Type	Audiobook
Version	Unabridged
Language	English
ASIN	B08X8H15BW
Best Sellers Rank	#5,062 in Audible Books & Originals (See Top 100 in Audible Books & Originals) #2 in Computer Systems Analysis & Design (Books) #3 in Software Design Tools #26 in Computers & Technology (Audible Books & Originals)

Amazon.com book



Amazon.com book

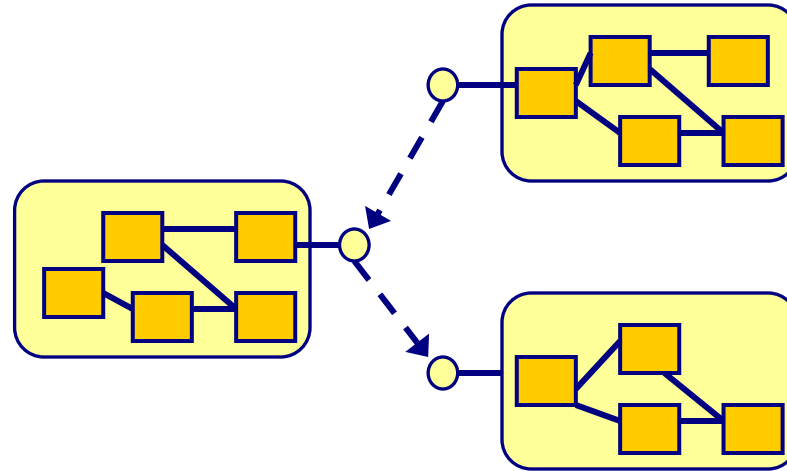


COMPONENT BASED DESIGN



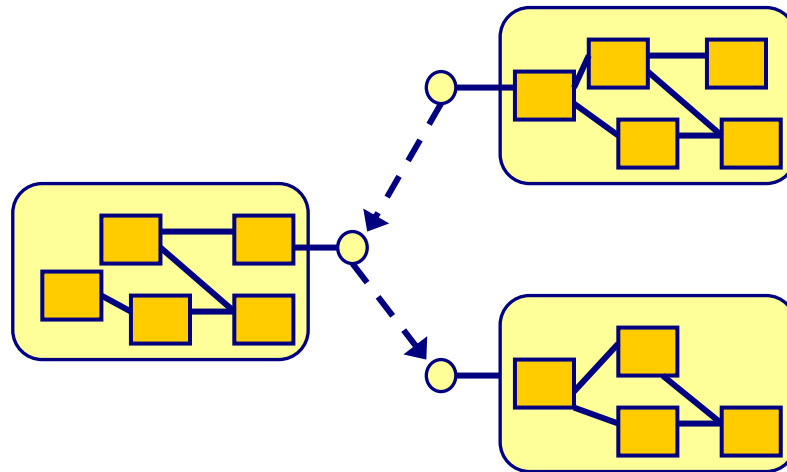
Component Based Development

- Decompose the domain in functional components



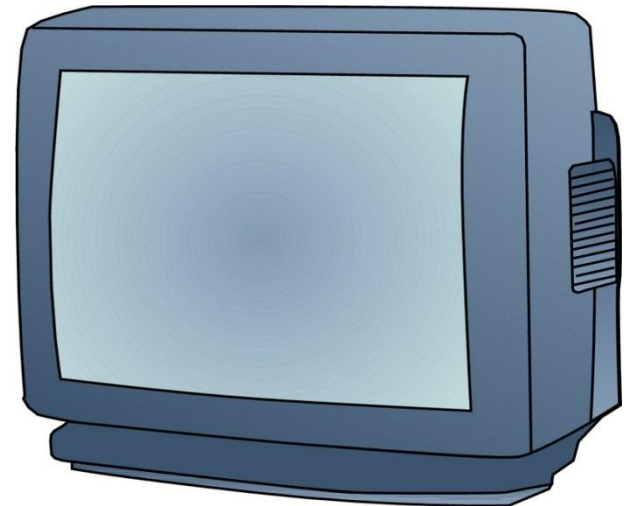
What is a component?

- There is no definition
- What we agree upon:
 1. A component has an **interface**
 2. A component is **encapsulated**
- Plug-and-play
- A component can be a single unit of deployment



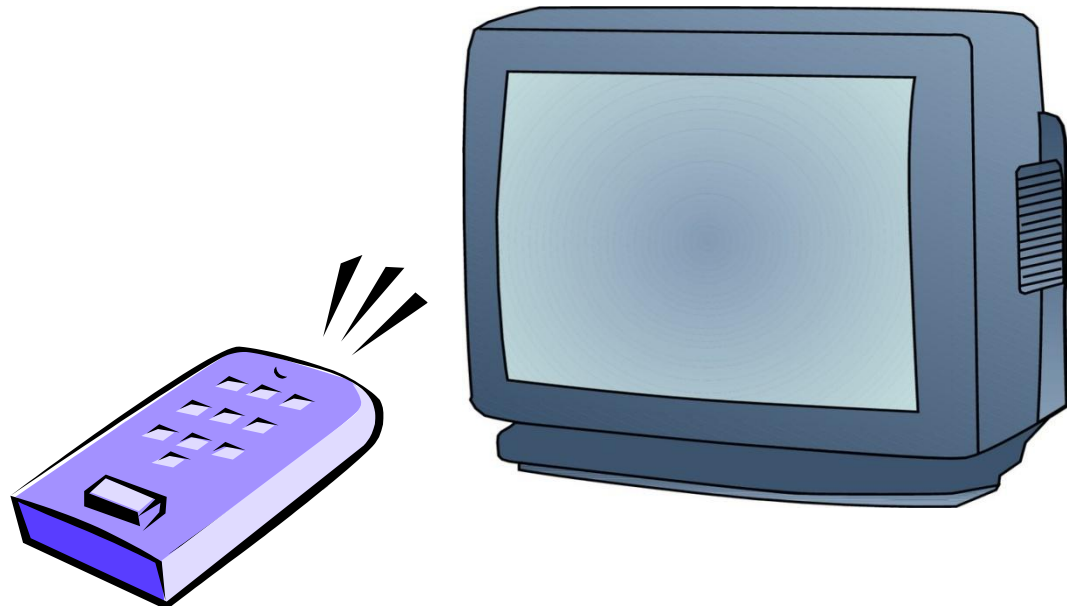
Encapsulation

- The implementation details are hidden



Interface

- The interface tells what you can do (but not how)



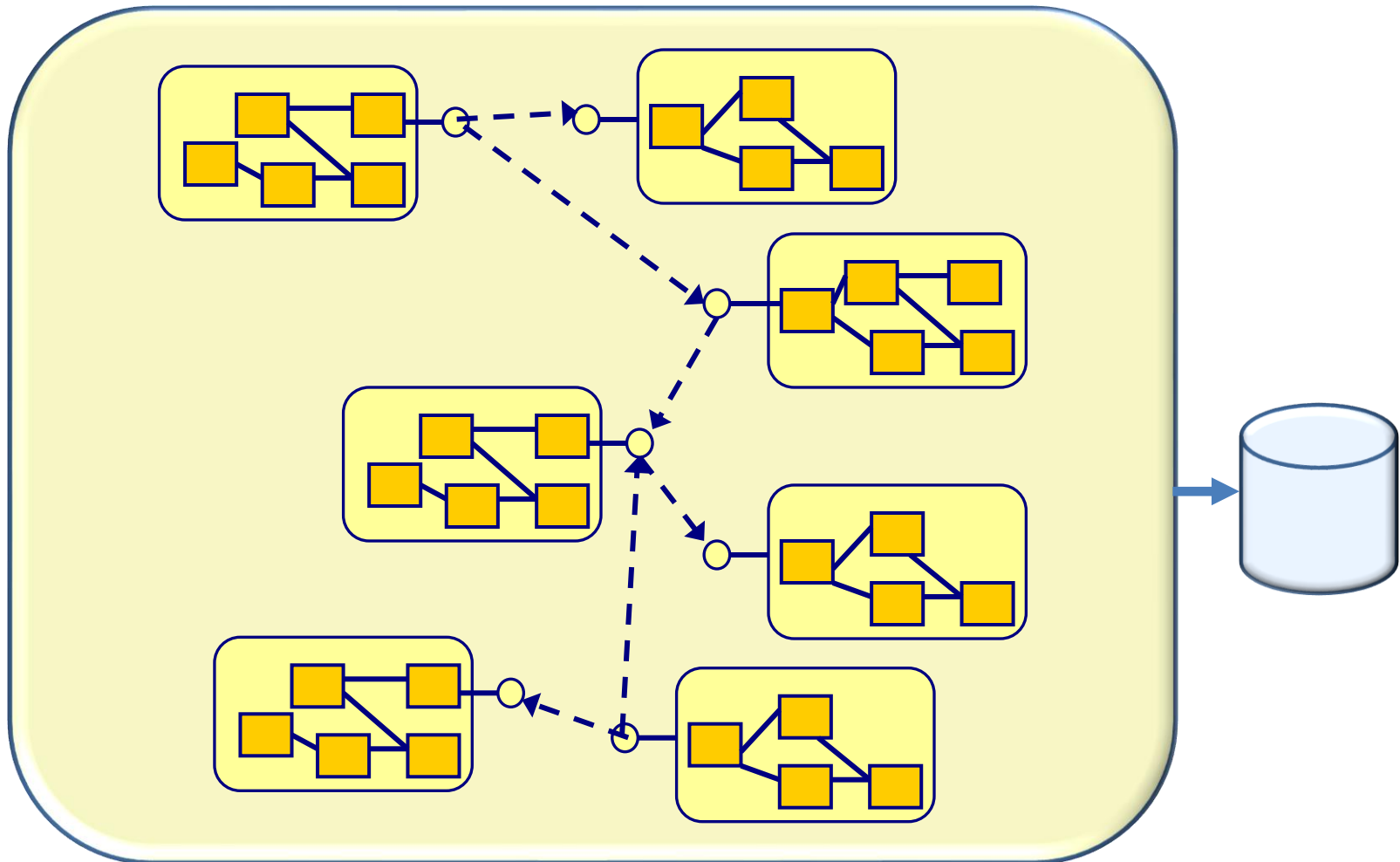
Advantages of components

- High cohesion, low coupling
- Flexibility
- Reuse ?

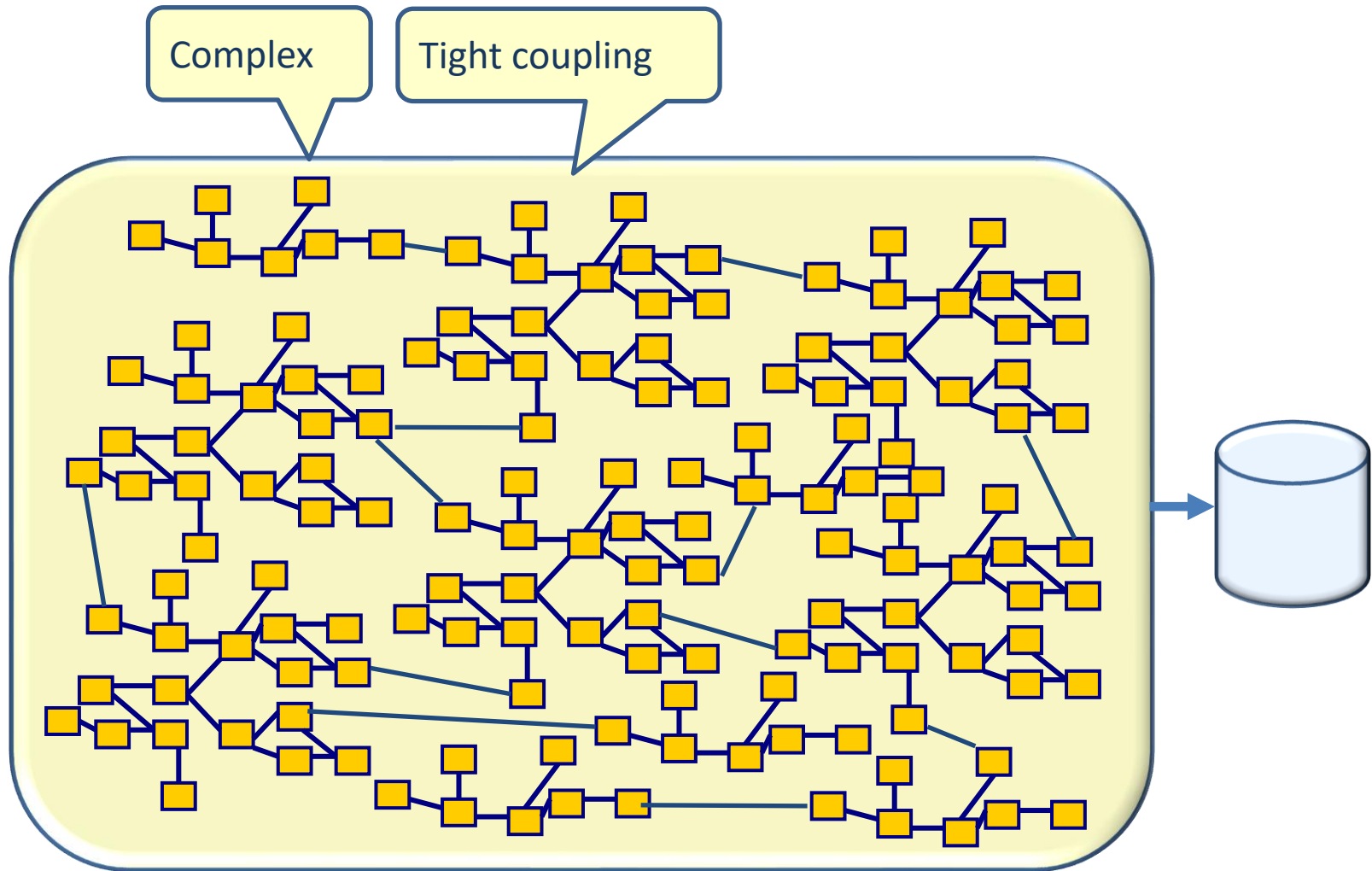


Component Based Development

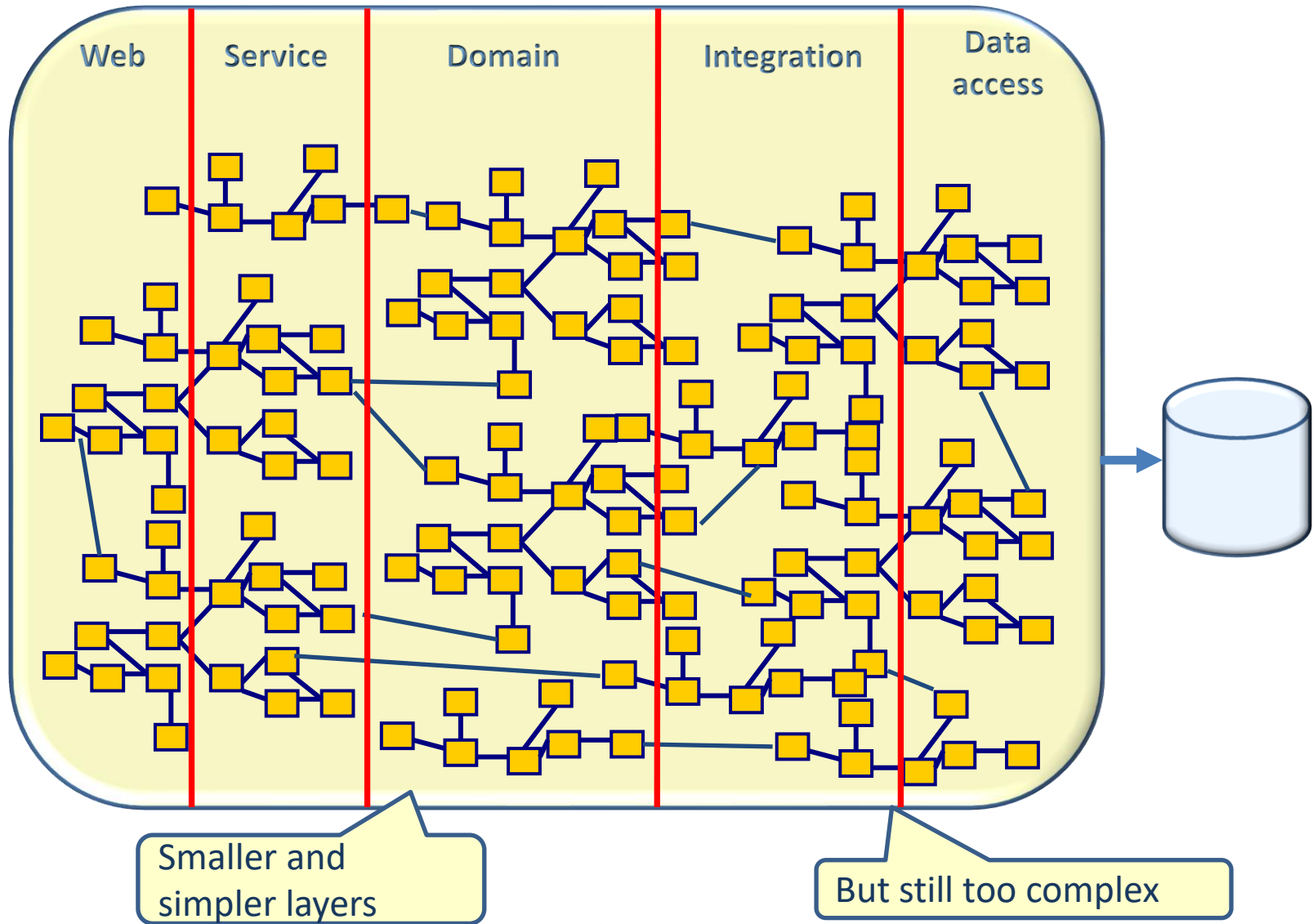
- Decompose the domain in functional components



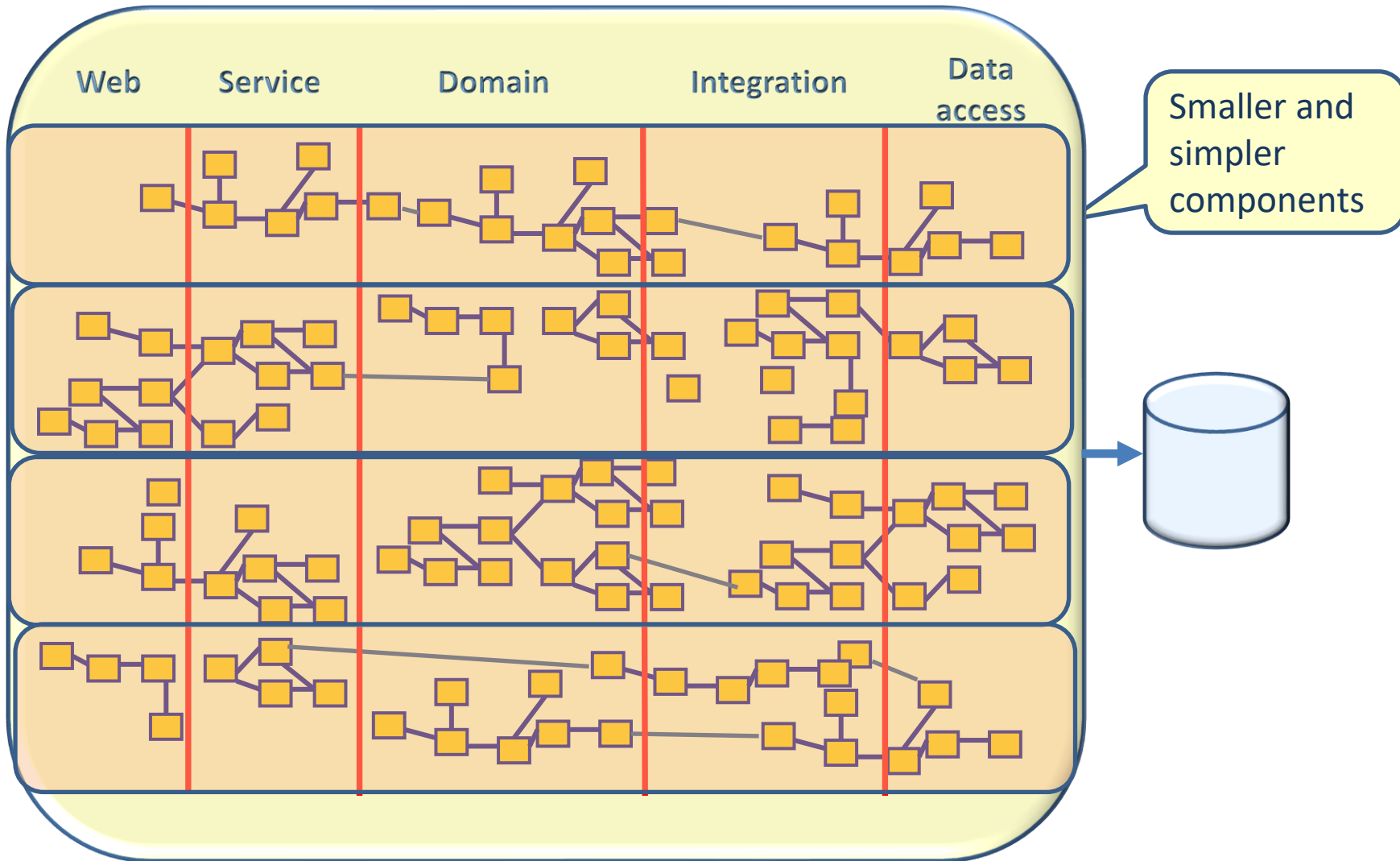
Object orientation



Layering



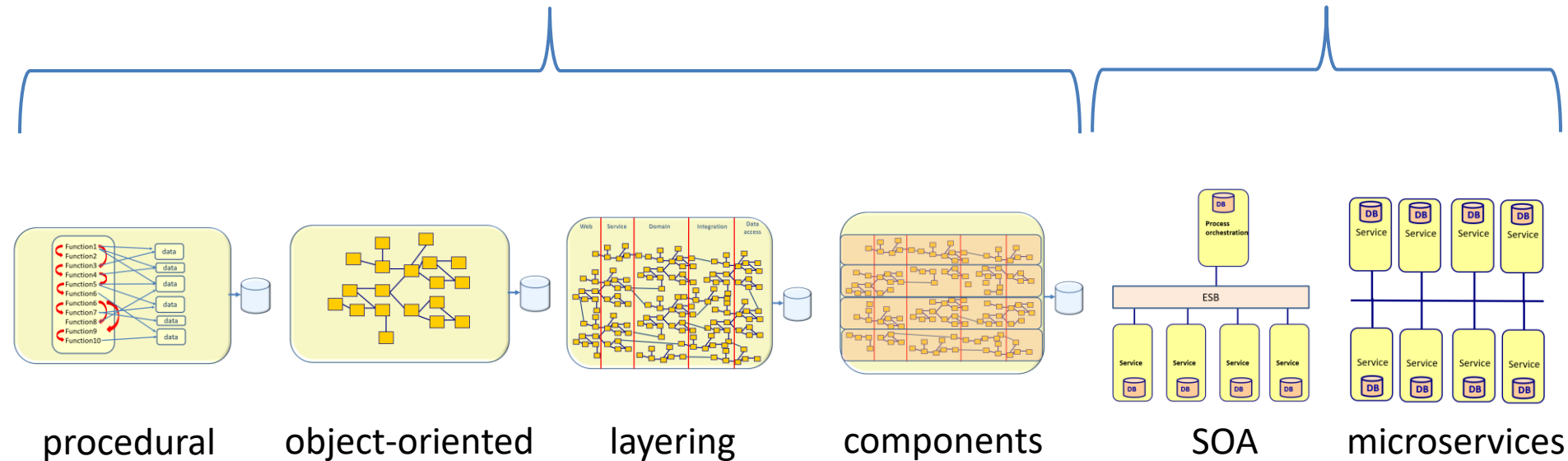
Components



Architecture evolution

Monolith

Distributed system



- Smaller and simpler parts
- More separation of concern
- More abstraction
- Less dependencies

Main point

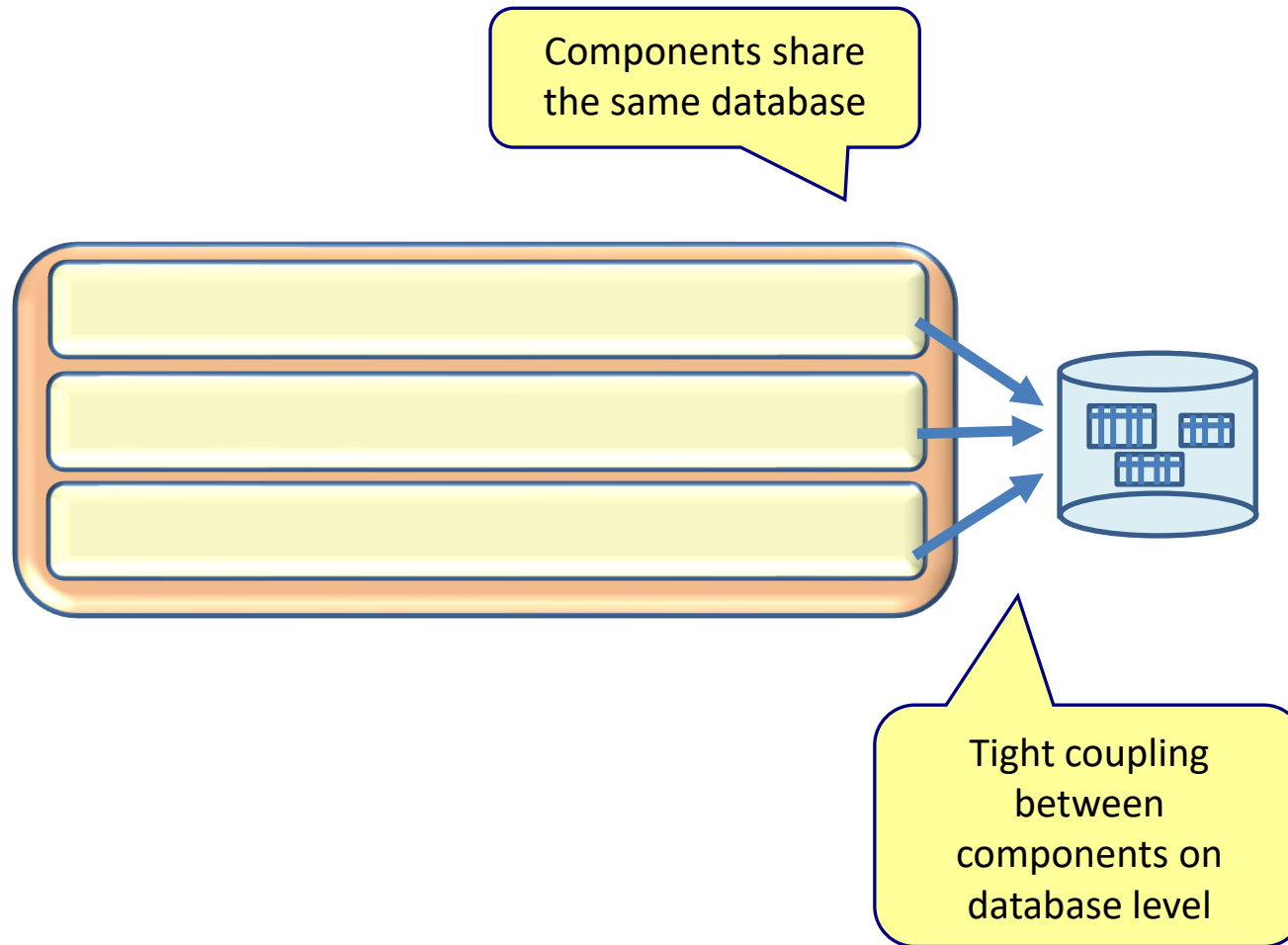
- Components are encapsulated and completely autonomous plug-and-play elements.
- The human nervous system is capable to transcend to that abstract field of pure consciousness which lies at the basis of the whole creation.



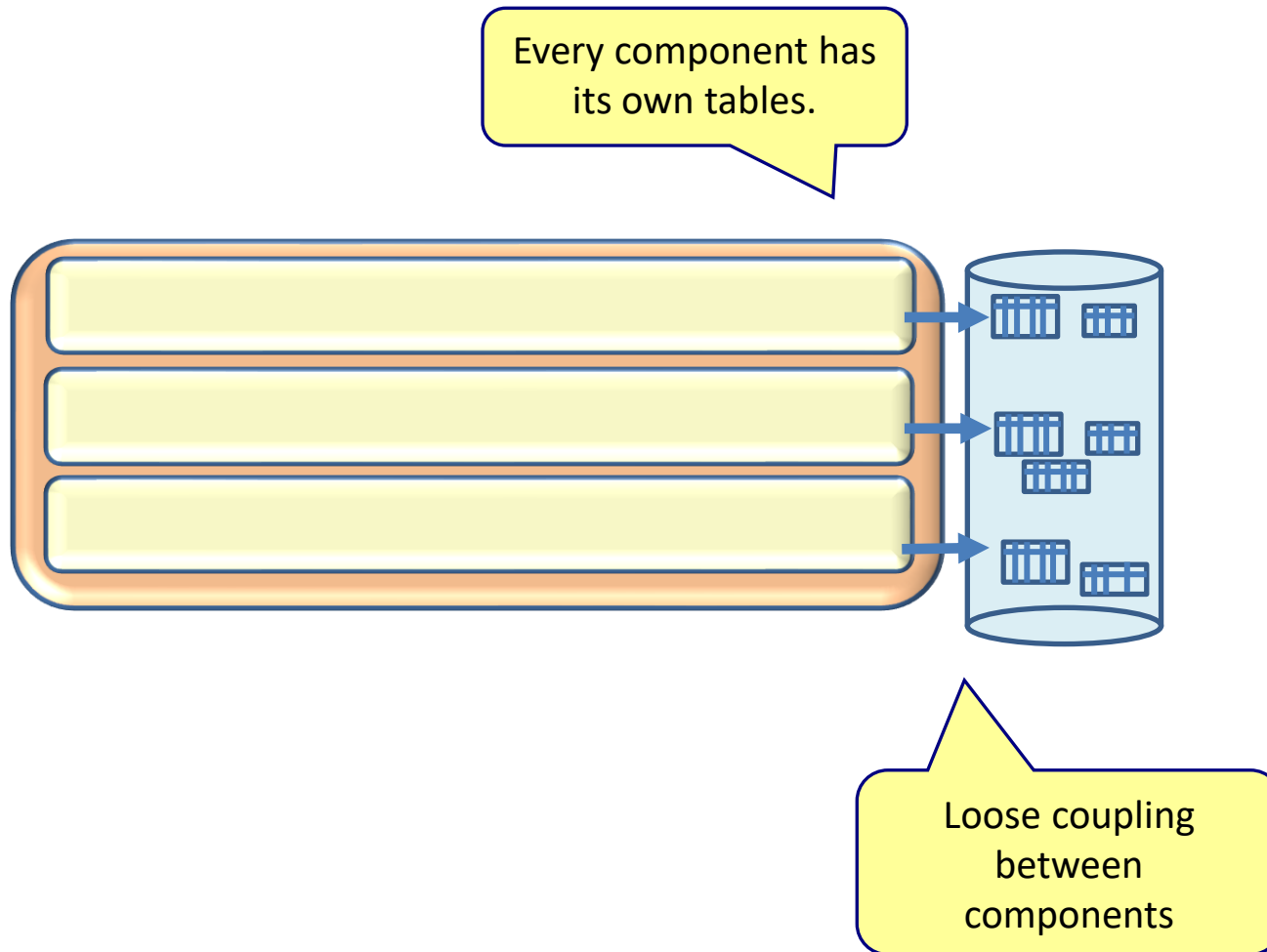
COMPONENT DESIGN



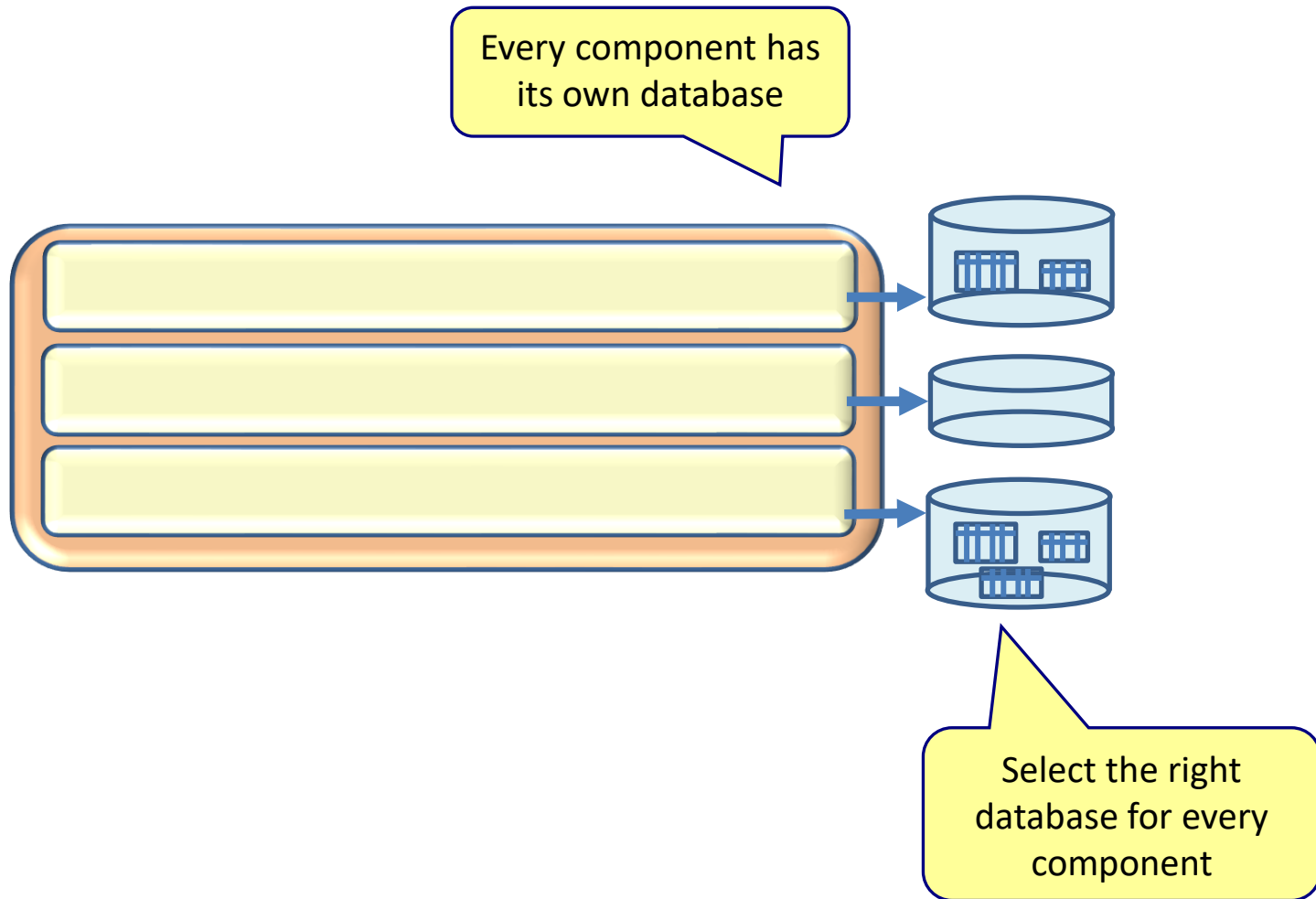
Components and database



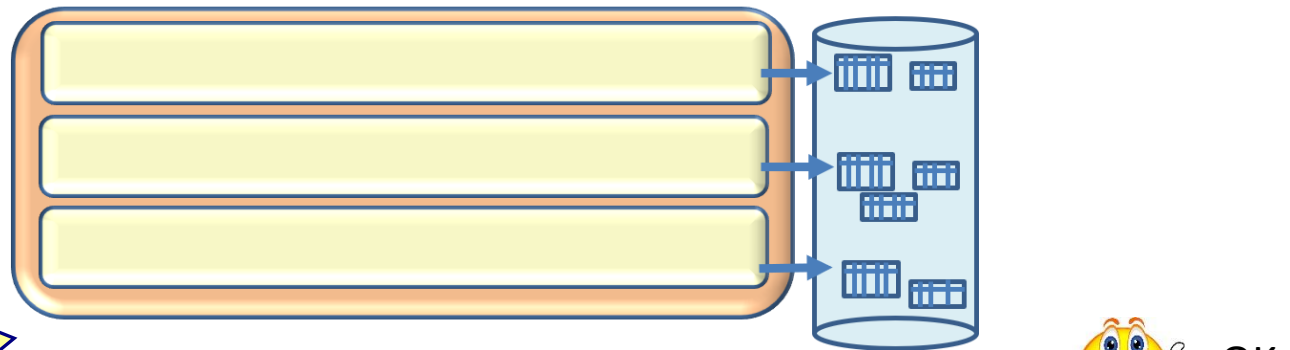
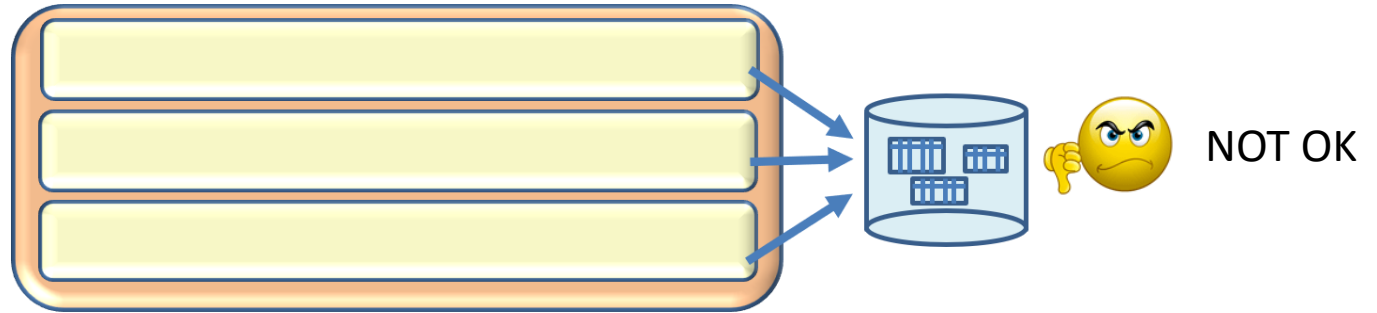
Components and database



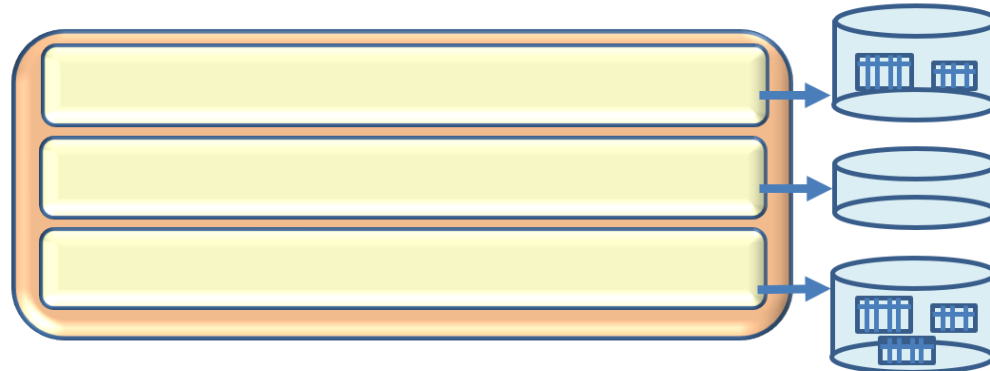
Components and database



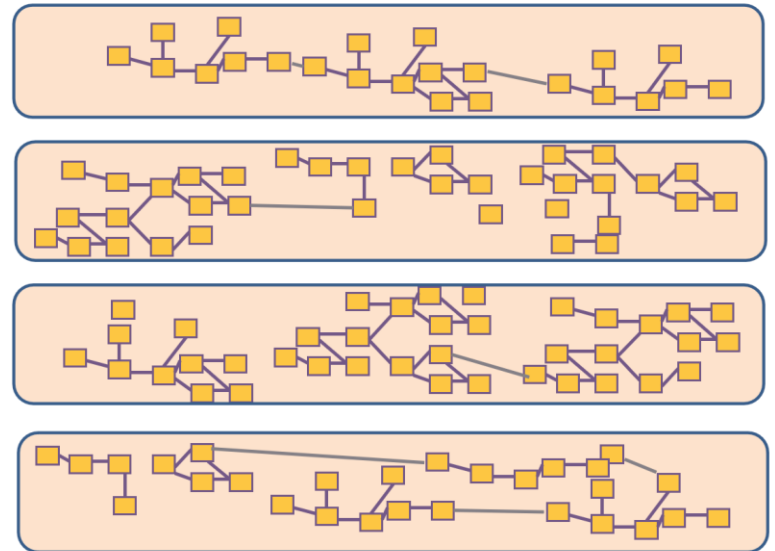
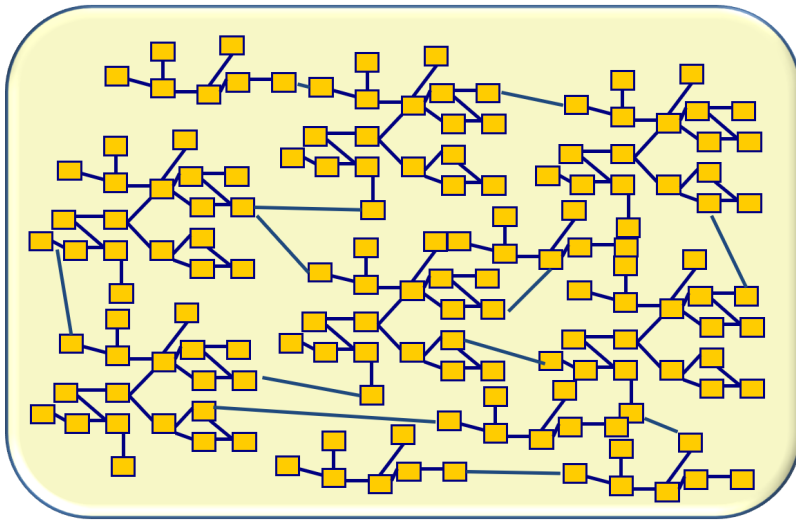
Components and database



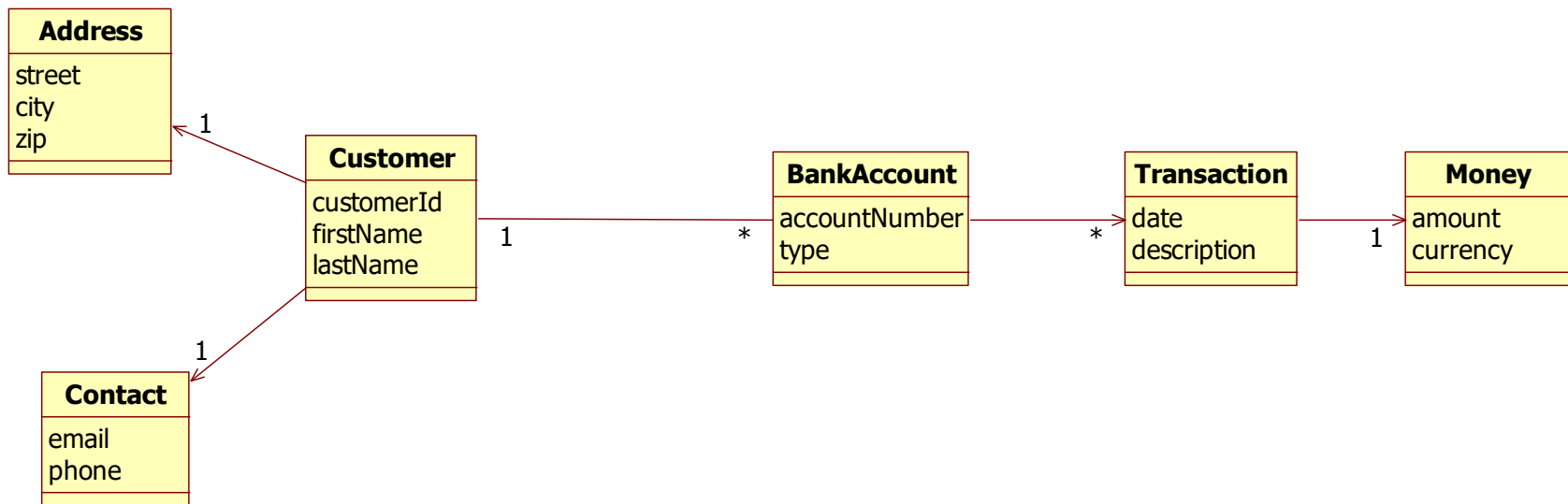
Loose coupling is the most important aspect of component based design



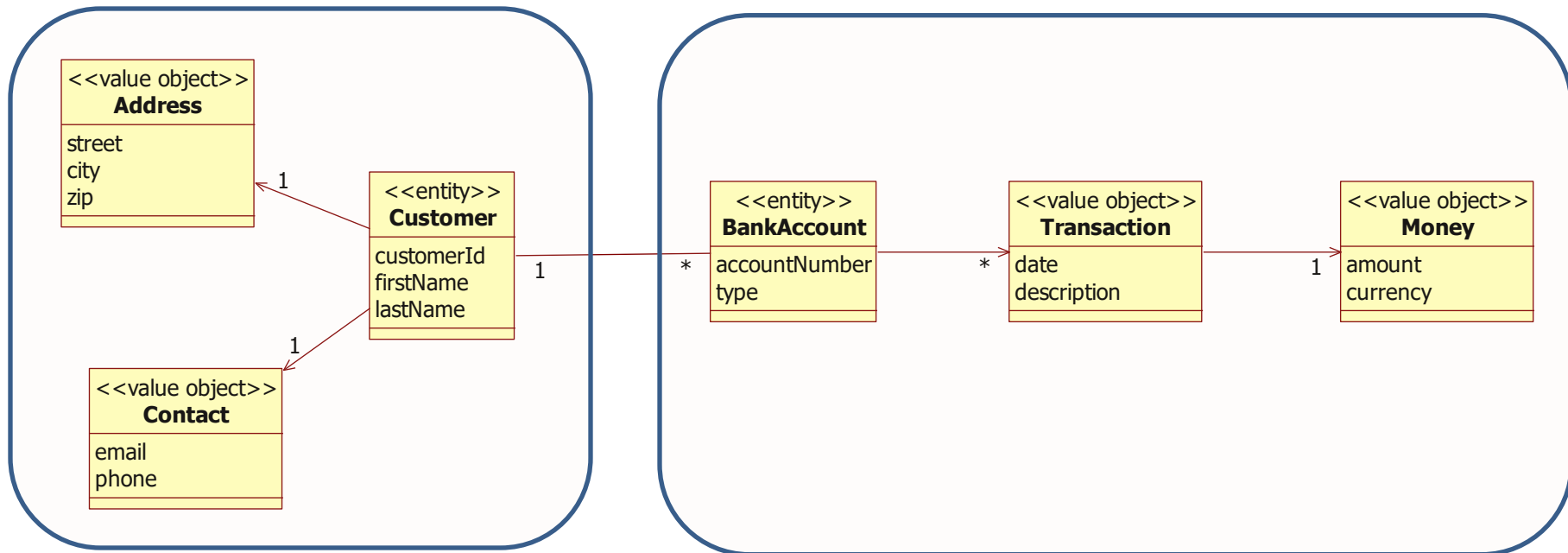
How do we split classes into components?



How do we split classes?

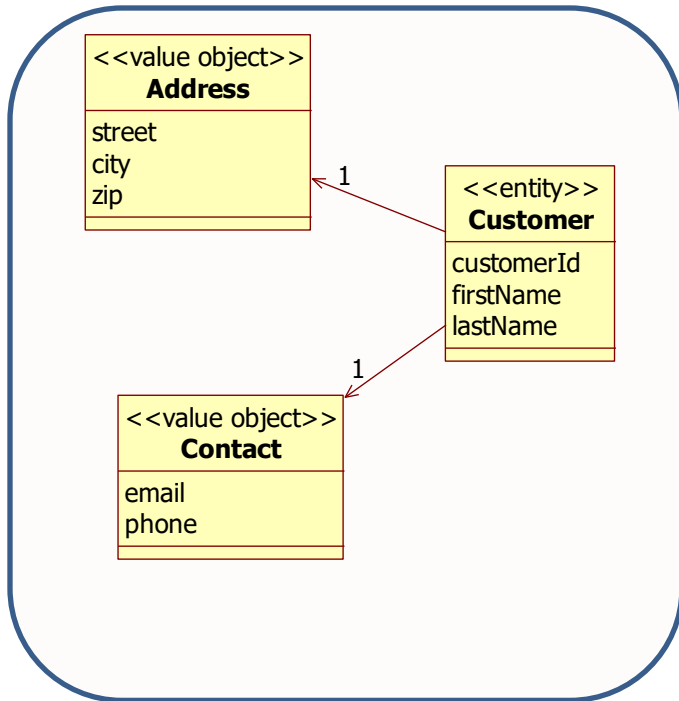


Aggregates

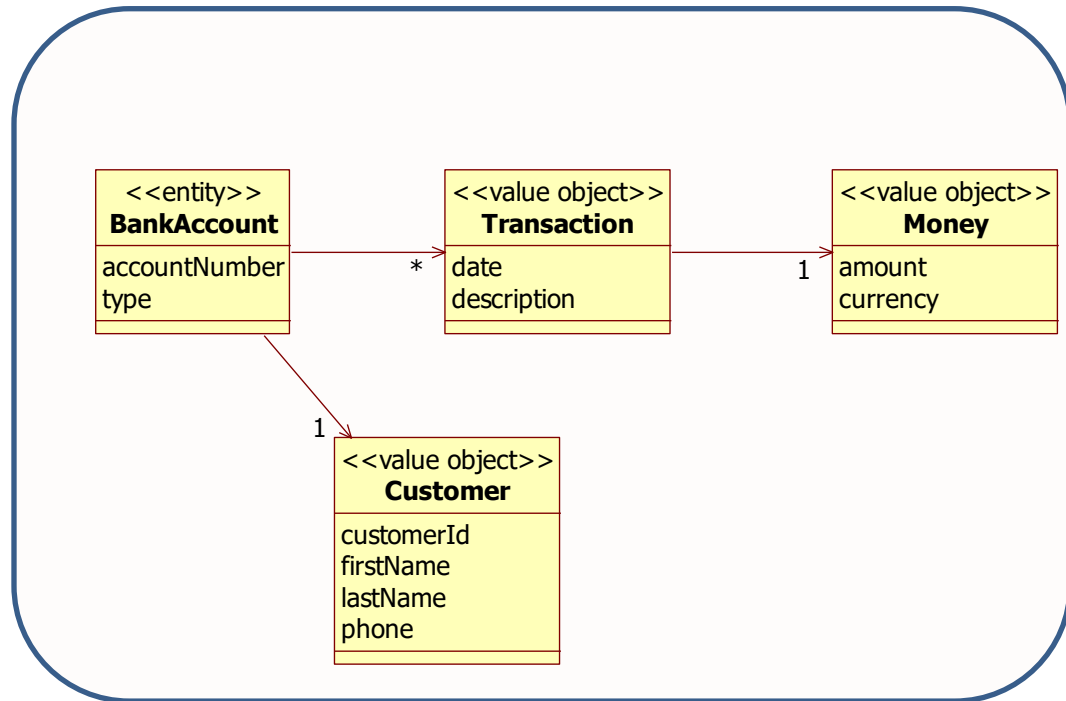


Aggregates

Customer component domain model



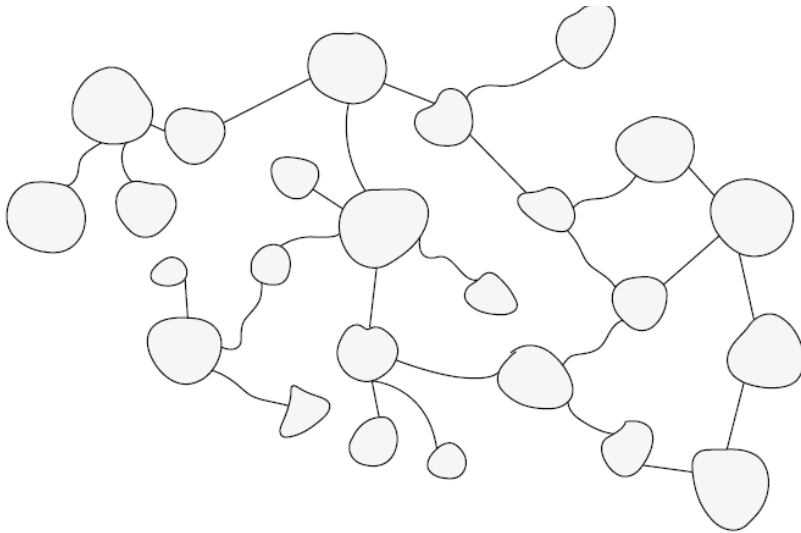
Account component domain model



Simplicity of the domain model



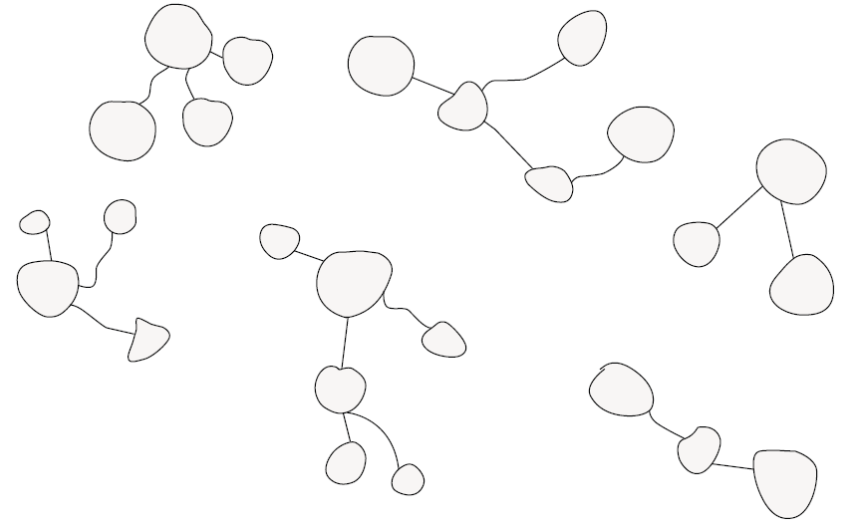
NOT OK



Complex domain model with many unnecessary associations

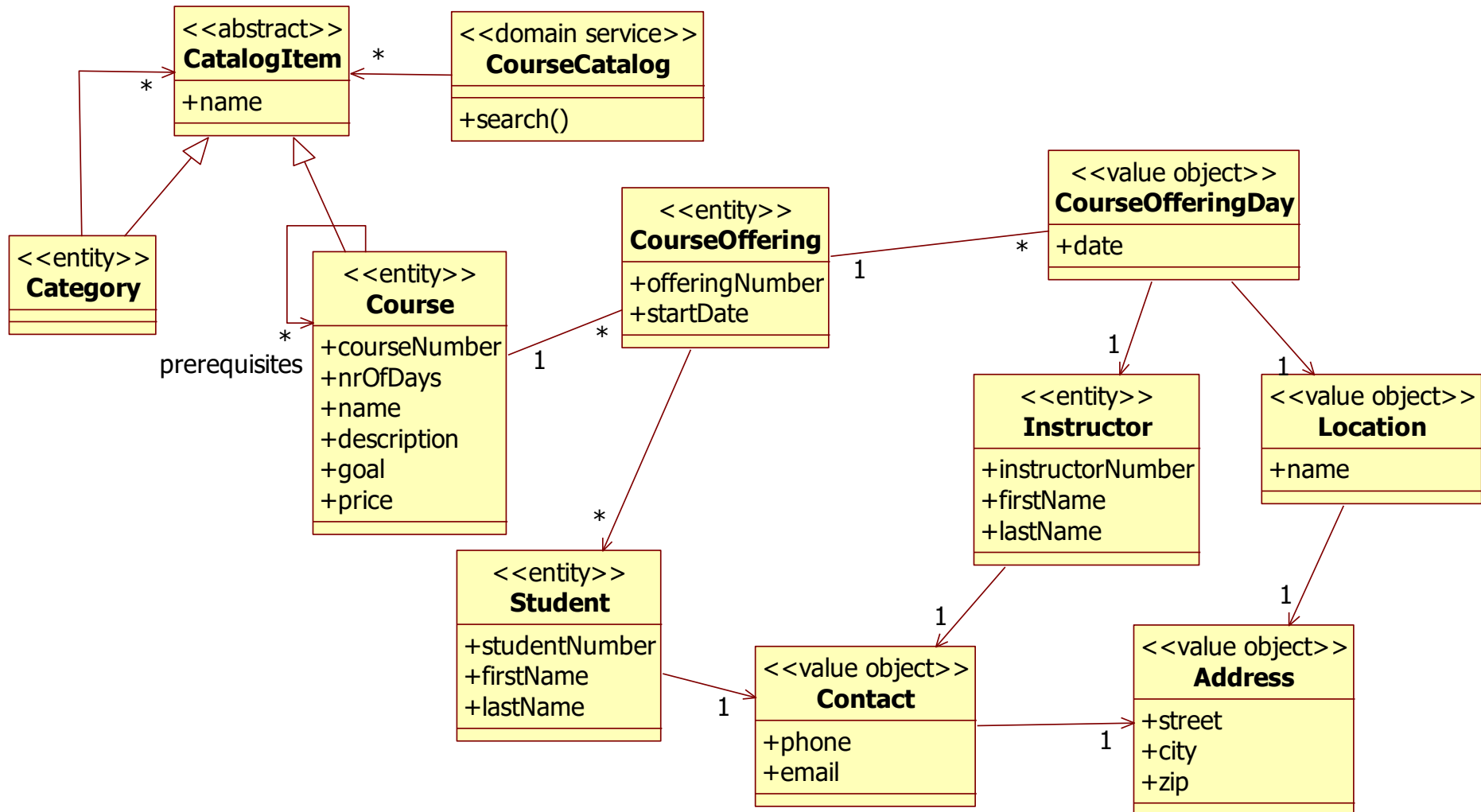


OK

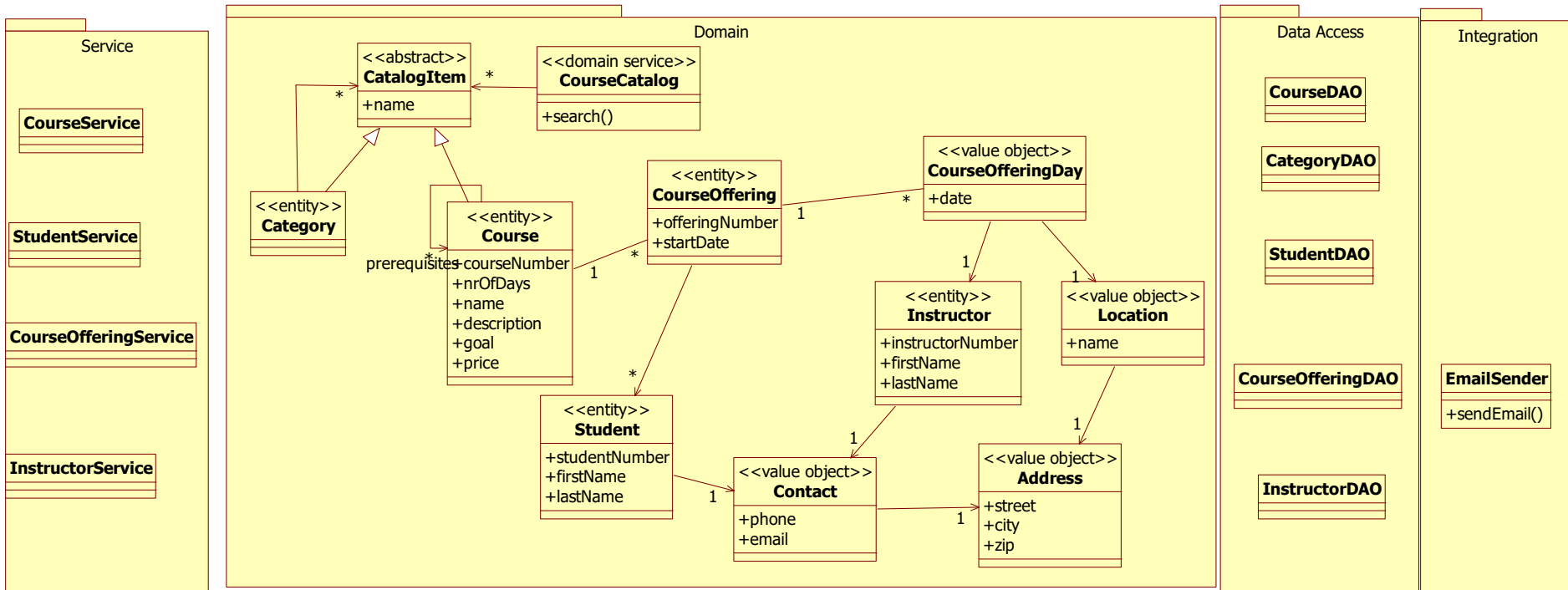


Clearer domain model with only the essential associations

Course Registration Domain Model



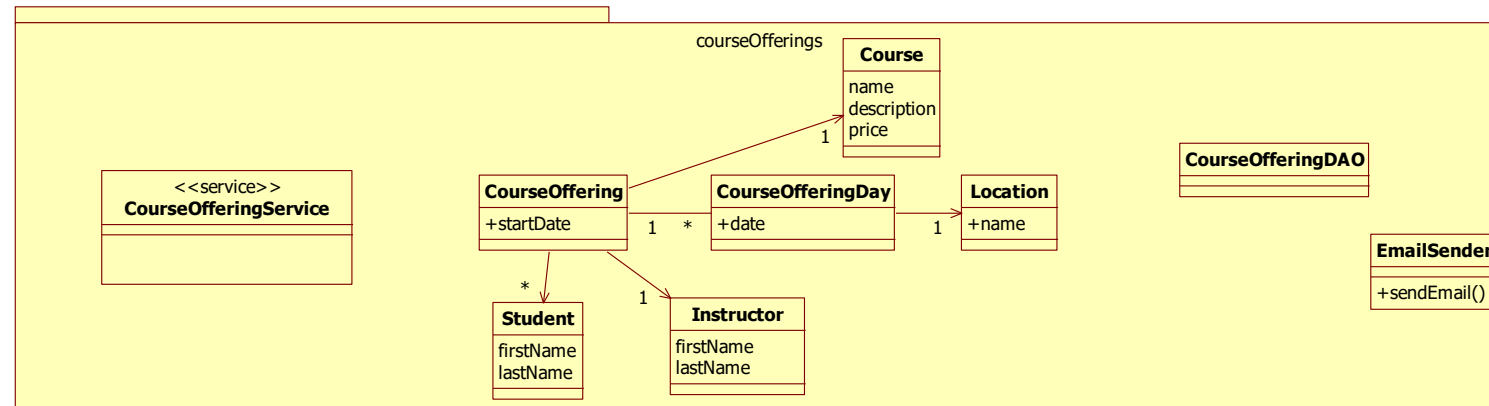
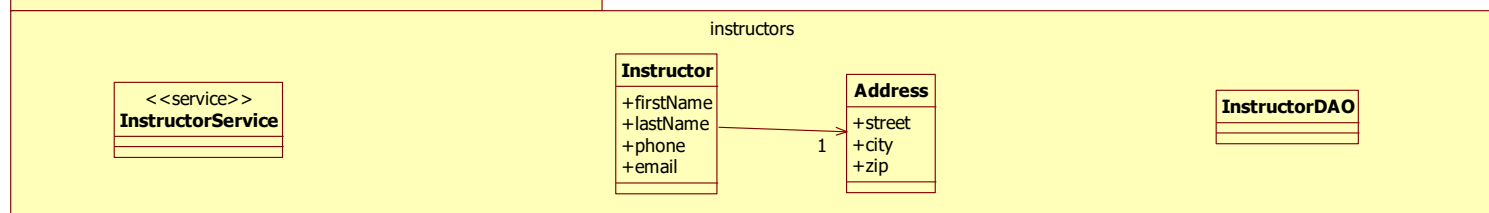
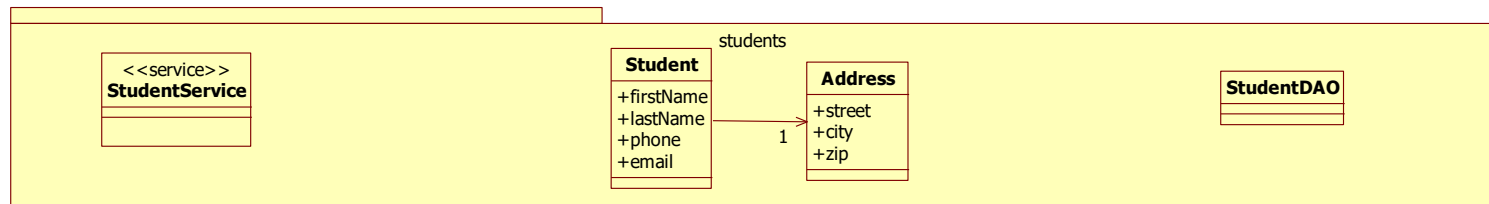
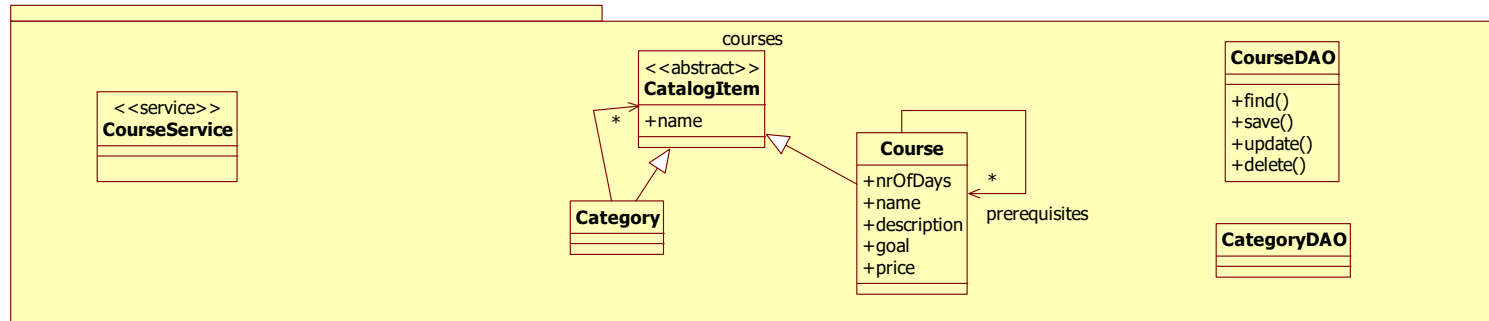
Course Registration Design



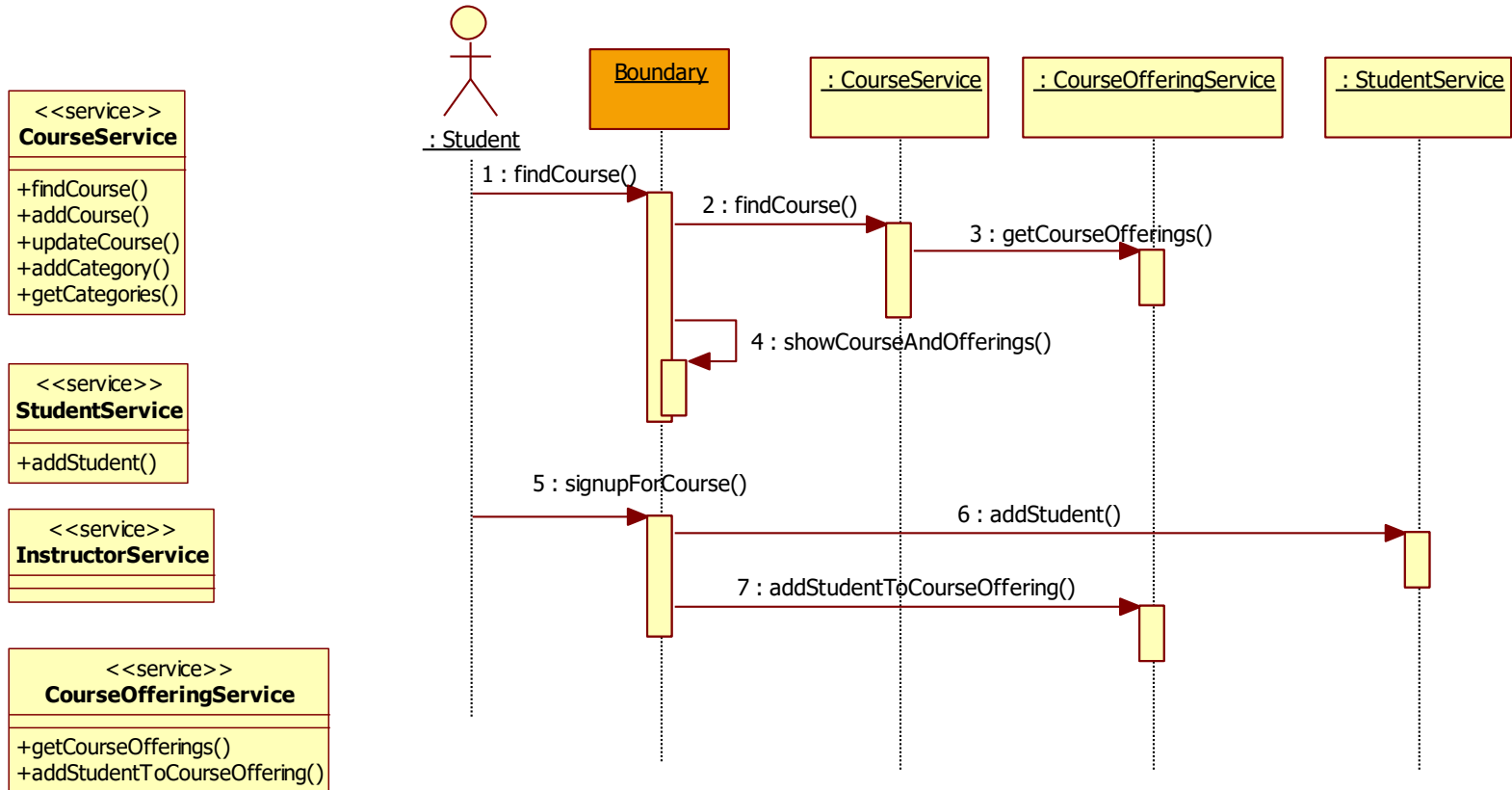
Course Registration with components



Course Registration with components



Course Registration with components

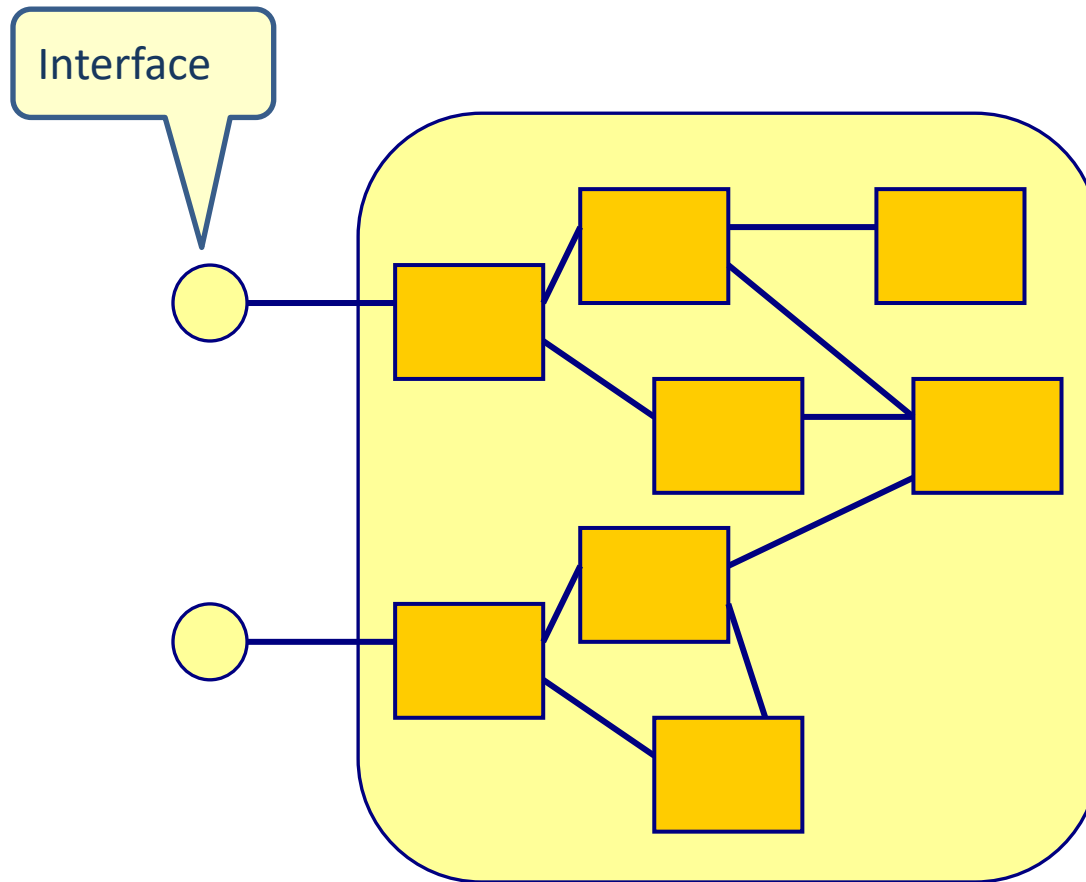


Lesson 5

API DESIGN

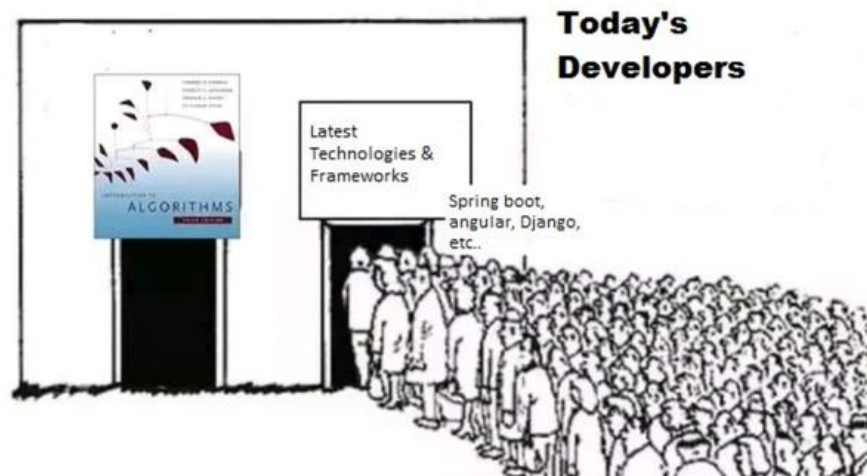


Interface design



Why is API design important?

- APIs play a bigger role in software development than ever before
- Software development has shifted from
 - Designing algorithms and data structures
 - To
 - Choosing, learning, and using an ever-growing set of API's



Why is API design important?

- Good APIs increase the quality of our applications
 - Bad API design infects our complete application
- The API we use shapes our code
 - API authors are responsible for the quality of the client code

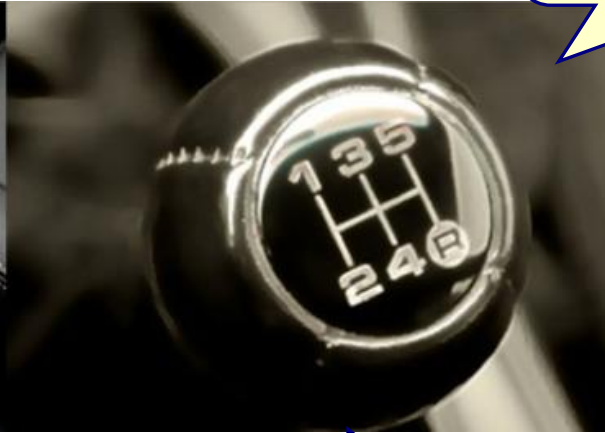


Manual versus automatic transmission

Transmission complexity is hidden



Transmission complexity is exposed to the driver



- + Simple for the driver
- + Less change on mistakes
- + Hard to misuse
- Driver has less control

- More complex for the driver
- Easier to make mistakes
- Easy to misuse
- + Driver has more control

JDBC API client

```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET firstname= ?,
                                                    lastname= ? WHERE employeenumber=?");

        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            System.out.println("Exception in closing jdbc connection in EmployeeDAO" + e1);
        }
    }
}
```

Open connection

Start transaction

Send the SQL

Commit transaction

Rollback transaction

Exception handling

Close connection

JDBC Template client

```
public void save(Product product) {  
    NamedParameterJdbcTemplate jdbcTempl = new NamedParameterJdbcTemplate (dataSource);  
    Map<String,Object> namedParameters = new HashMap<String,Object>();  
    namedParameters.put("productnumber", product.getProductnumber());  
    namedParameters.put("name", product.getName());  
    namedParameters.put("price", product.getPrice());  
    int updateresult = jdbcTempl.update("INSERT INTO product VALUES ( :productnumber,  
                                       :name, :price)",namedParameters);  
}
```

The template takes care of connection,
transaction and exception handling



Why is API design important?

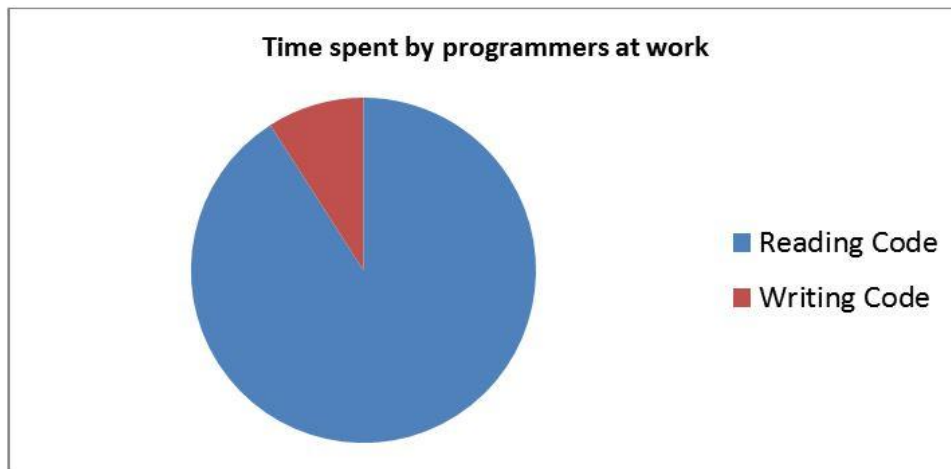
- Public APIs are forever
 - One chance to get it right



Why is API design important?

- Usually write-once, read/learn many times by many different people
 - Developers spend most of their time learning APIs, using APIs and debugging code

Does it run? Just leave it alone.



Writing Code that
Nobody Else Can Read

The Definitive Guide

Why is API design important to you?

- If you program, you are an API designer
 - Good code is modular
 - Each module has an API
- Useful modules tend to get reused
 - Once a module has users, can't change API at will

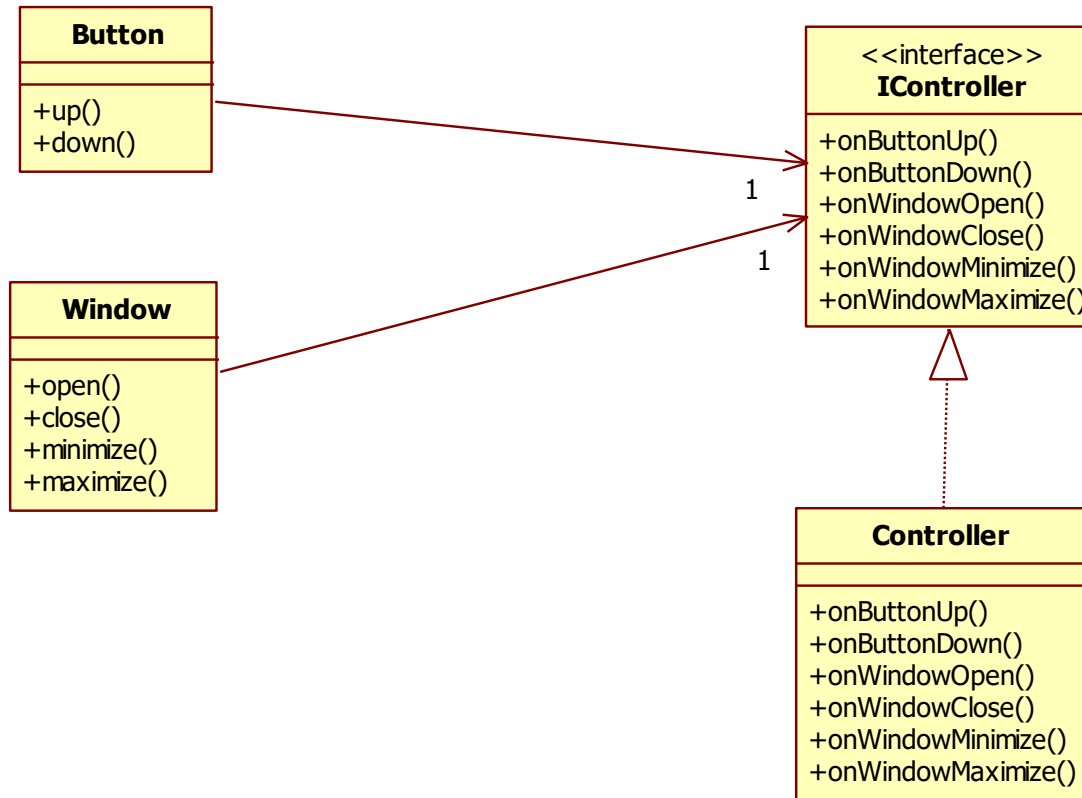


Interface design best practices

- Start with the client first
 - Single responsibility principle
 - Interface segregation principle
- Easy to use
- Easy to learn
- Hide as much as possible
- When in doubt, leave it out

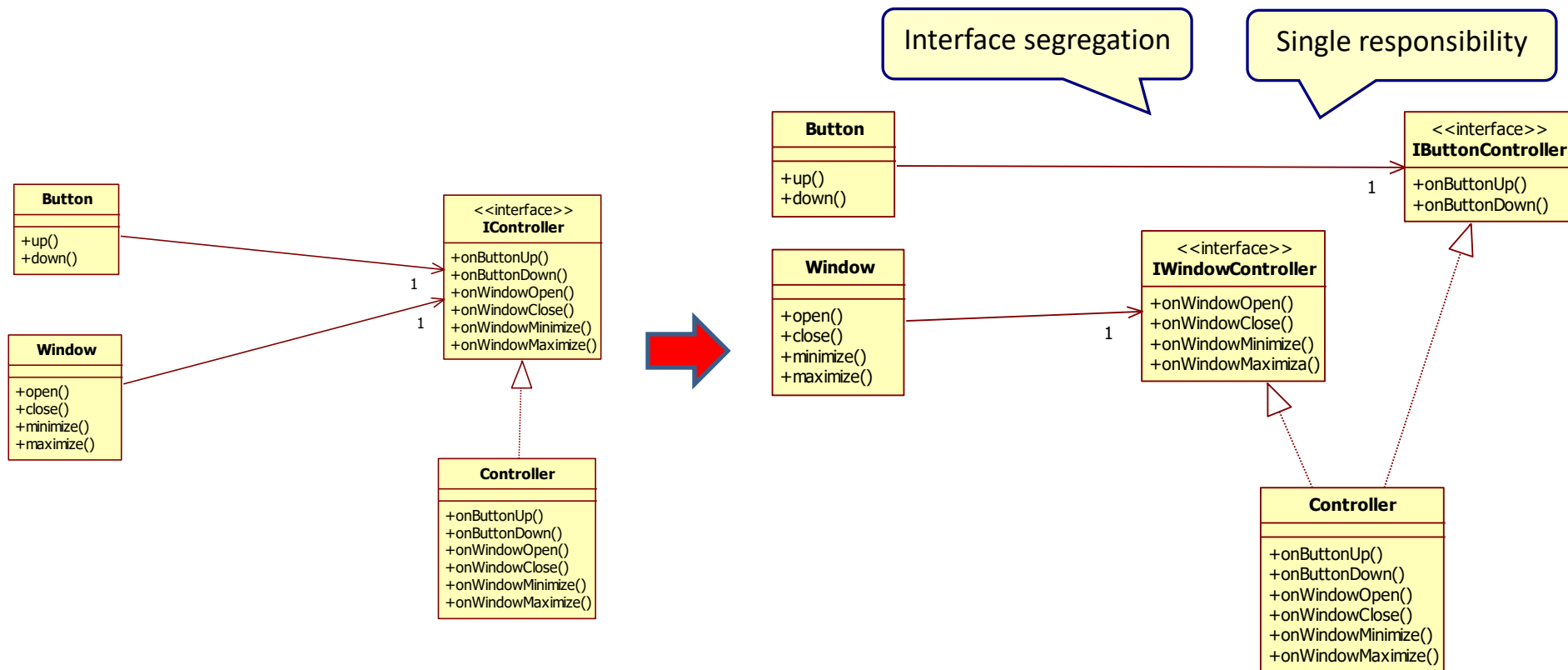


What is wrong with this API?

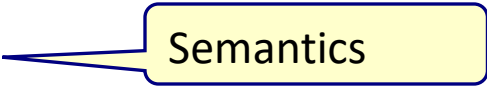


Interface segregation principle

- Clients should not be forced to depend on methods (and data) they do not use



The interface should be easy to use

- And hard to misuse.
 - Easy to do simple things
 - Possible to do complex things
 - Impossible (or at least difficult) to do wrong things
- The interface should be simple
- No surprising behavior 
 - Don't do anything that is relevant to the client and that you cannot derive from the names of the functions and parameters (or comments)



Easy to use



NOT OK

```
String findString(String text,  
                 bool search_forward,  
                 bool case_sensitive);
```

```
findString("text", true, false);
```

- Order is clear
- Self-descriptive.

```
findString("text",  
SearchDirection.FORWARD,  
CaseSensitivity.CASE_INSENSITIVE);
```

```
enum SearchDirection {  
    FORWARD, BACKWARD  
};  
enum CaseSensitivity {  
    CASE_SENSITIVE, CASE_INSENSITIVE  
};  
String findString(String text,  
                 SearchDirection direction,  
                 CaseSensitivity case_sensitive);
```

Prefer enums over
booleans to improve
code readability.



OK



The interface should be easy to learn

- Well documented
- Names matter (classes, functions, variables)
 - Clear code

`bucket.empty()`



NOT OK

`getCurrValue()
getCurValue()
getCurVal()`

`bucket.isEmpty()`

`bucket.makeEmpty()`



OK

`getCurrentValue()`



Do not expose implementation details

- The client should be independent of implementation details
 - You can change the implementation without changing the client
- Don't let implementation details “leak” into the API
 - An API method that throws a SQL exception
 - An API method that returns a hash table



Minimally complete

- But no smaller
- API should satisfy its requirement
- **When in doubt, leave it out**
 - You can always add
 - You can never remove
- API should do one thing, and do it well
- Do not add extra levels of generality for the future
 - You might never need it
 - When you need it, you have a better understandability of what is needed
 - Its easier to add to a simple API than a complex API

It is tempting to expose all possible functionality you can think of.

If it is not required, then leave it out.

The API should be as small as possible

Use immutable objects on the interface

- Immutable object
 - Object whose internal state cannot be changed after its creation
- Advantages
 - Immutable objects are simple.
 - Immutable objects are inherently thread-safe; they require no synchronization.
 - Immutable objects can be shared freely.
 - Immutable objects make great building blocks for other objects



Don't make the client do anything the module can do

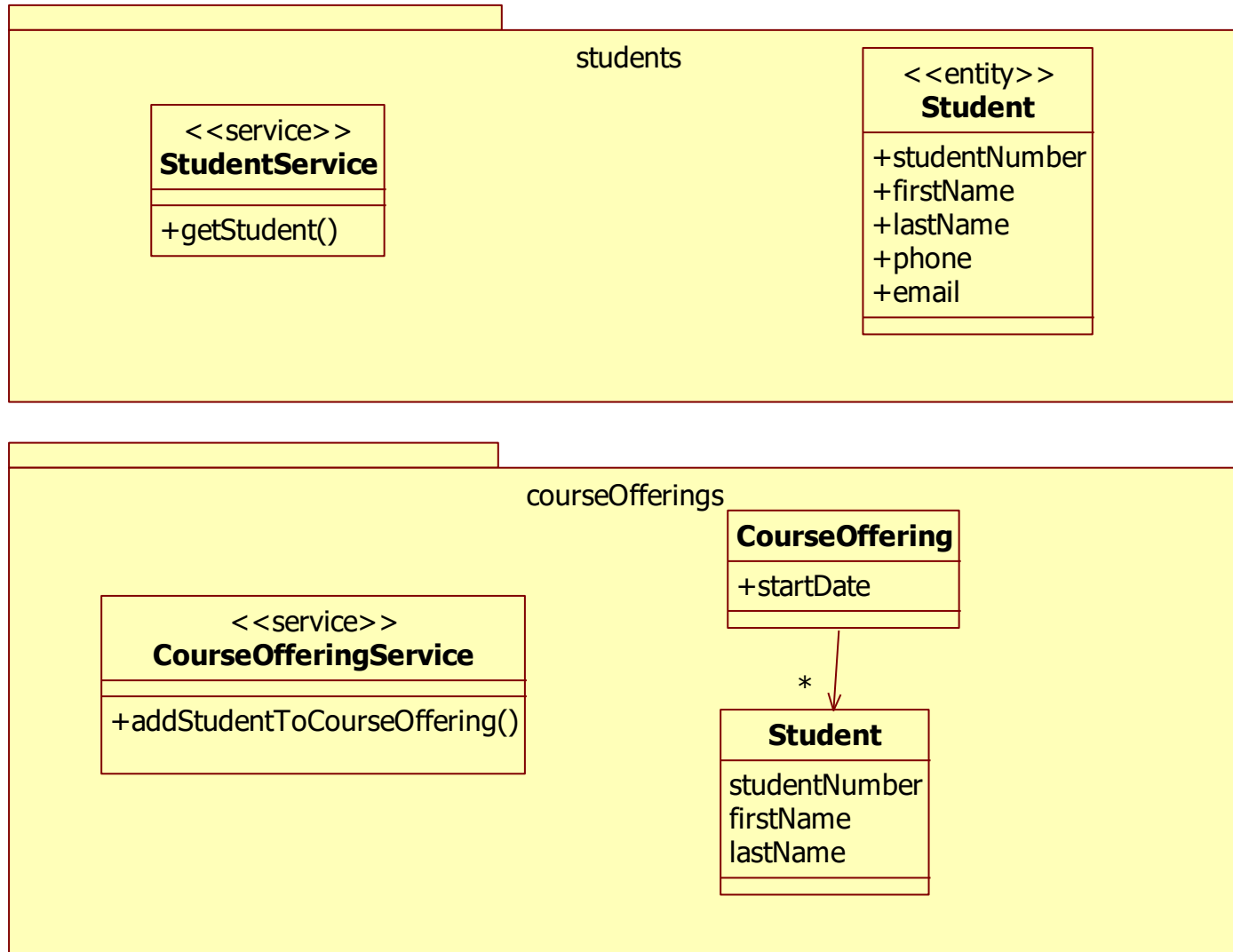
- Reduce the need for boilerplate code
 - Generally done via cut-and-paste
 - Ugly, annoying, and error-prone



DATA TRANSFER OBJECTS (DTO)

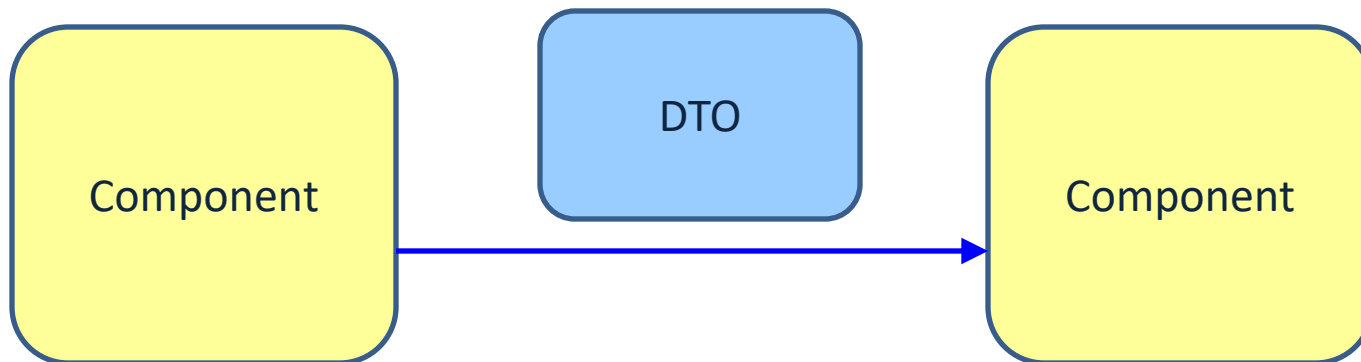


No shared data between components

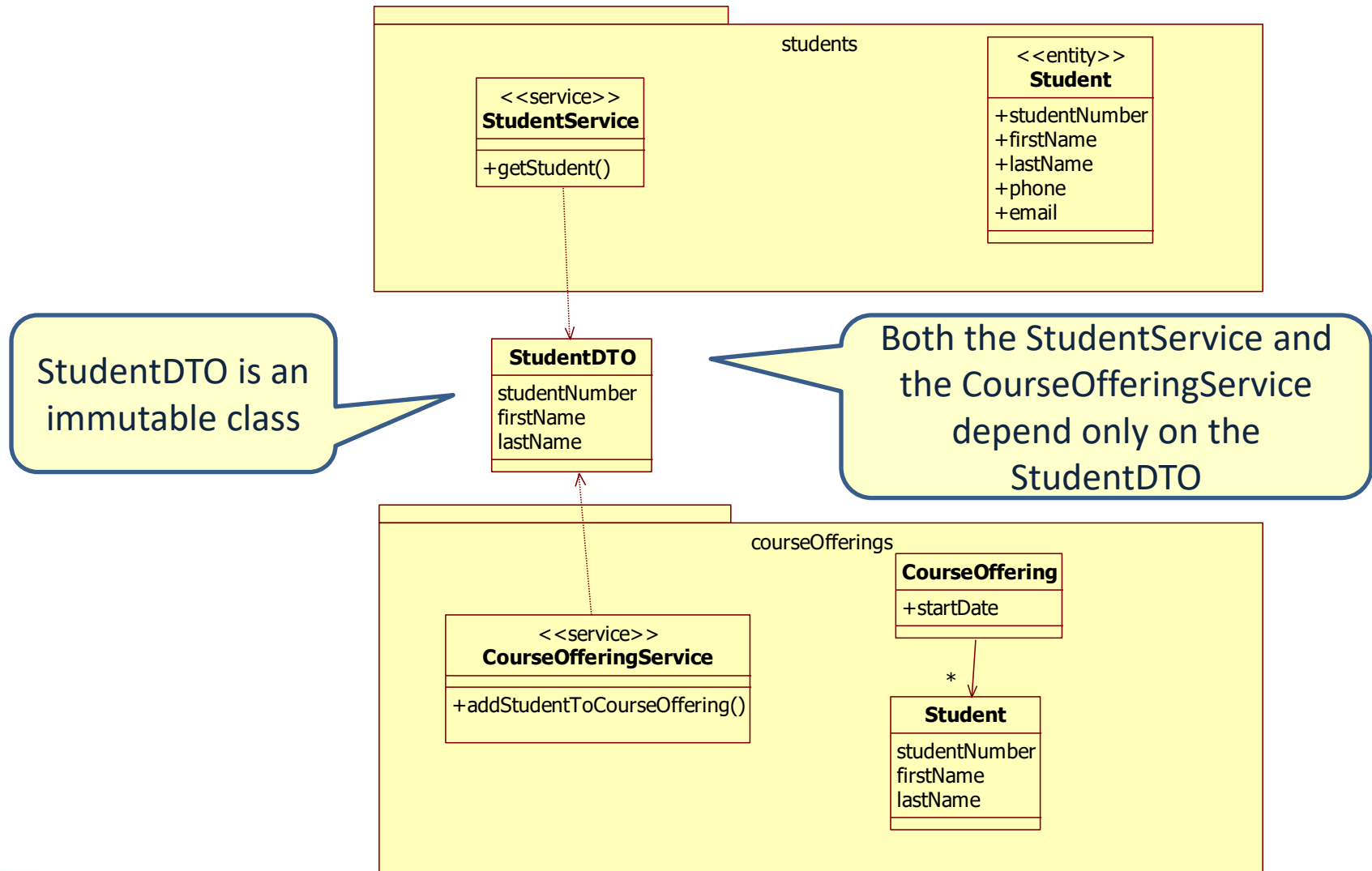


Data Transfer Objects (DTO)

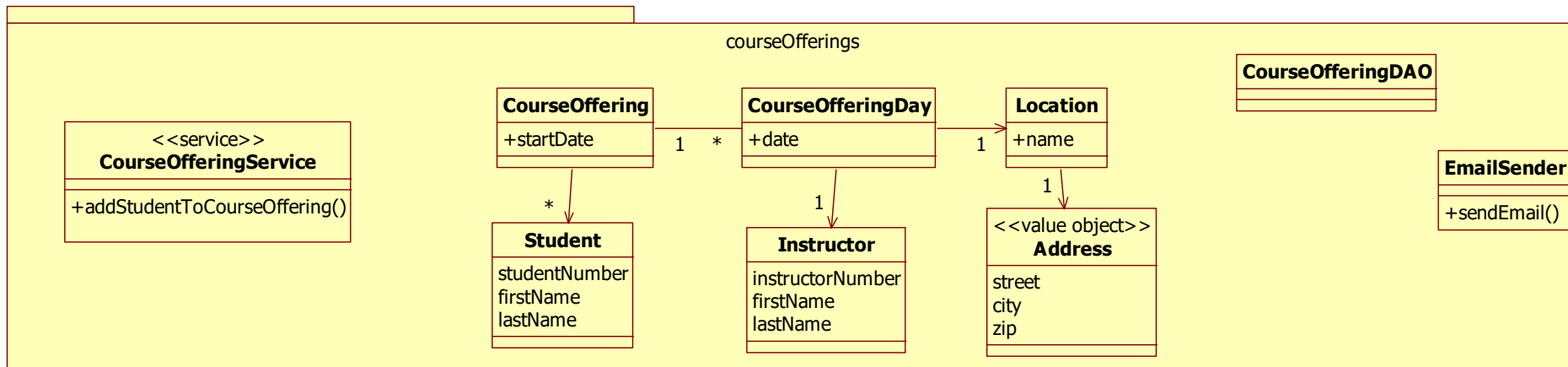
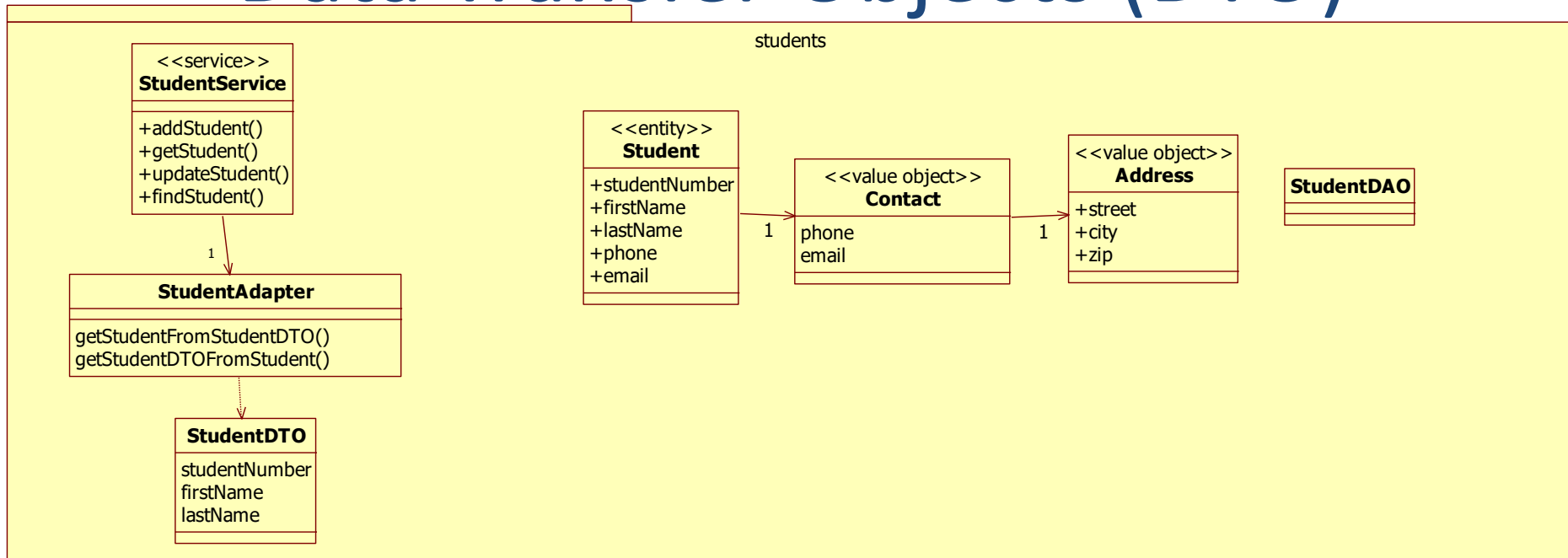
- Object that contains only attributes and getters and setters



Data Transfer Objects (DTO)



Data Transfer Objects (DTO)



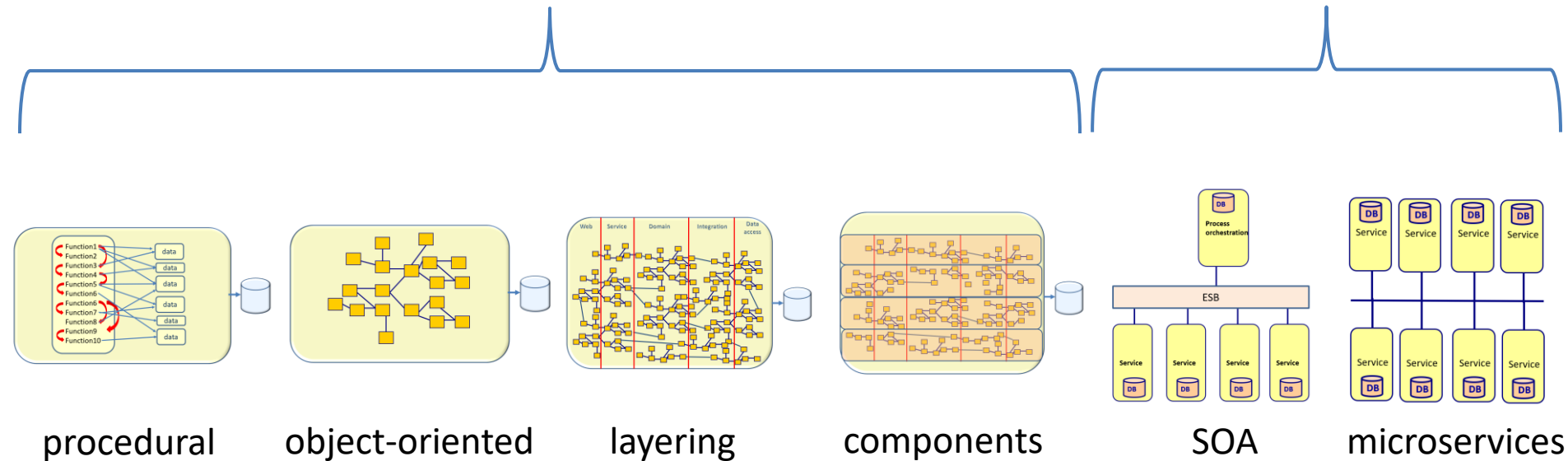
SUMMARY



Architecture evolution

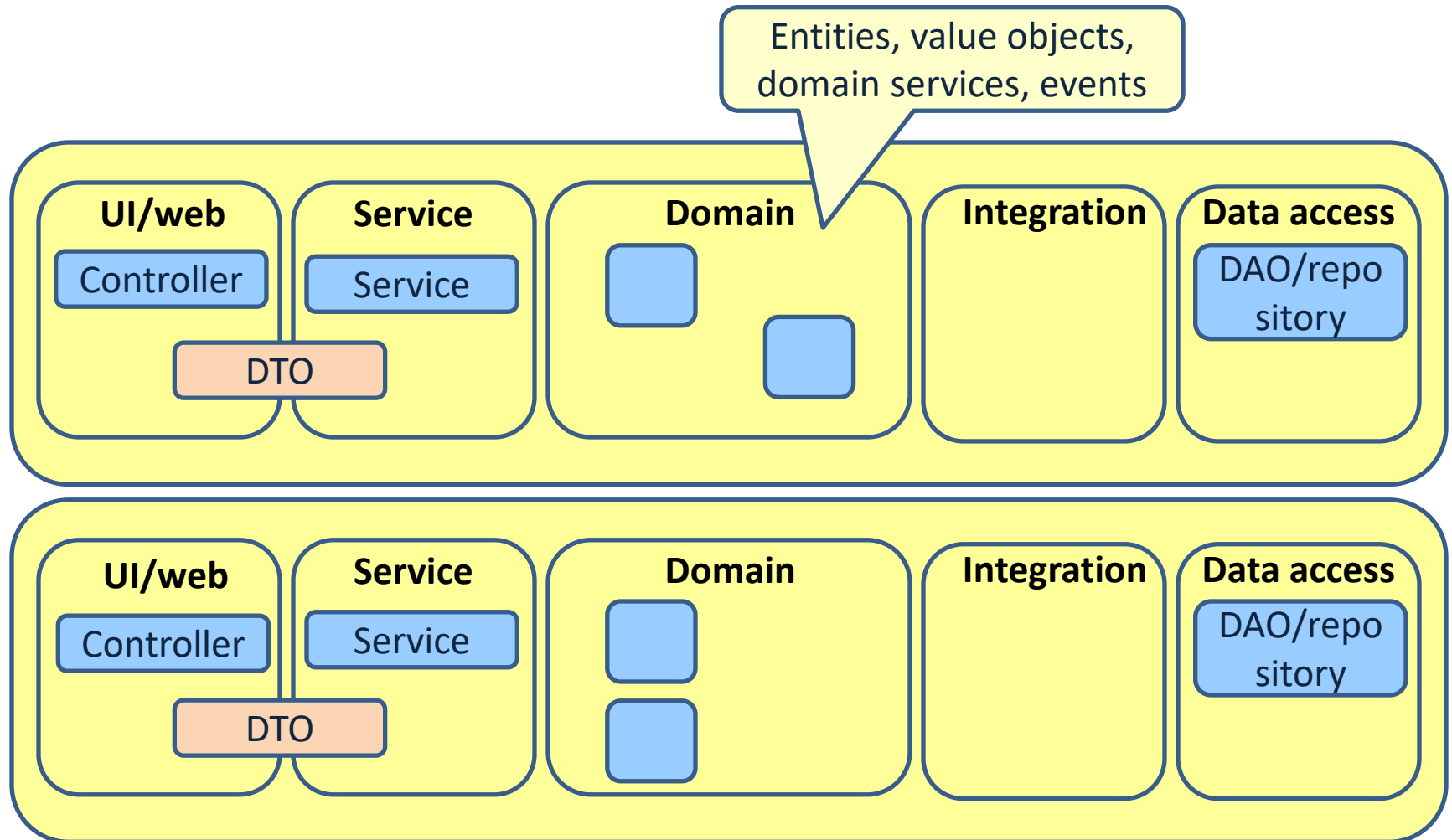
Monolith

Distributed system



- Smaller and simpler parts
- More separation of concern
- More abstraction
- Less dependencies

Component design



Connecting the parts of knowledge with the wholeness of knowledge

1. A component is a loosely coupled module which is encapsulated and has one or more interfaces
 2. The only coupling between components is the interface with Data Transfer Objects
-

3. **Transcendental consciousness** is the source of all intelligence in nature.
4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that all components in creation are just expressions of ones own Self.

