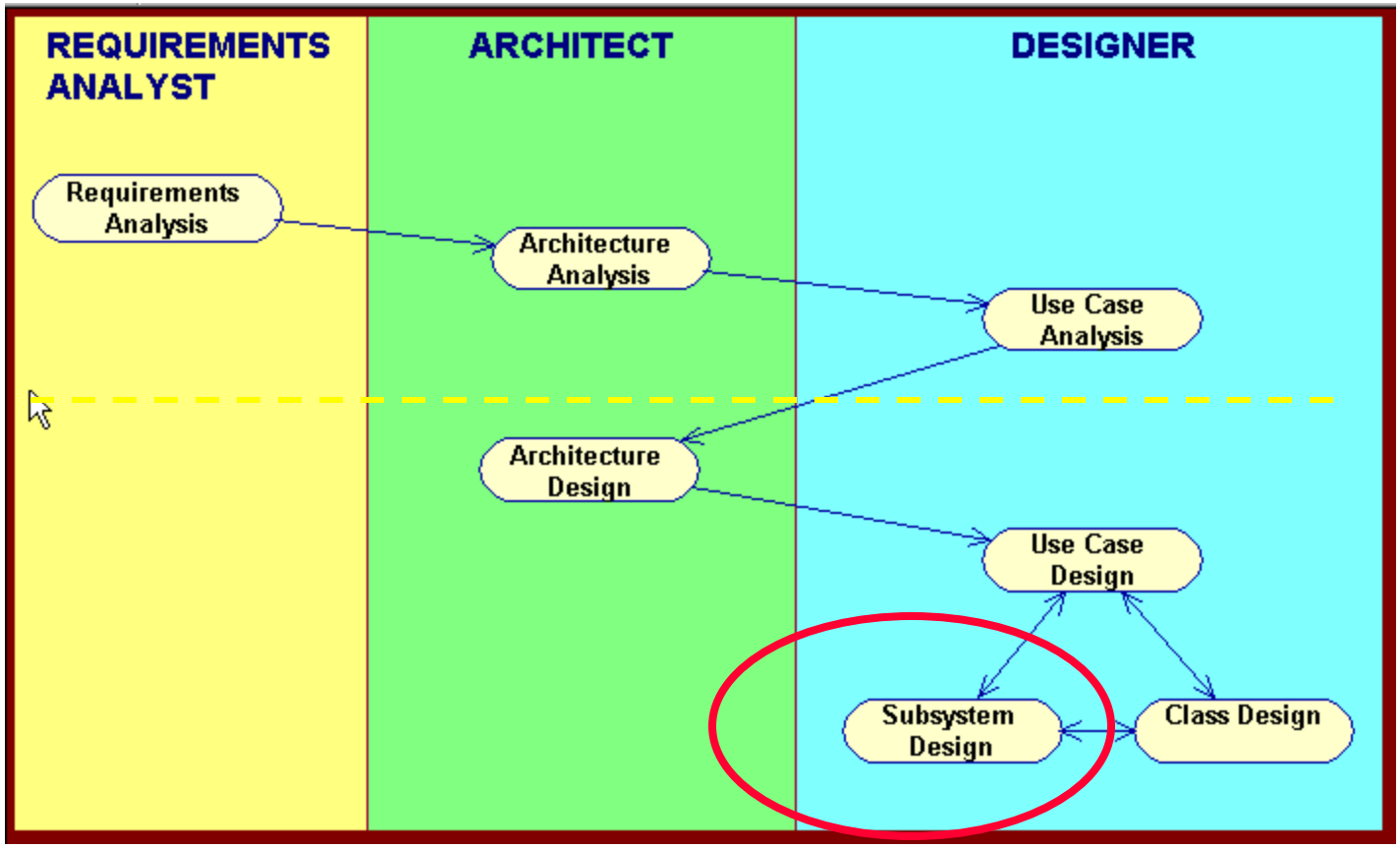
Subsystem Design

Order is present everywhere

Basic RUP OOAD Activities



Subsystem Design vs Analysis

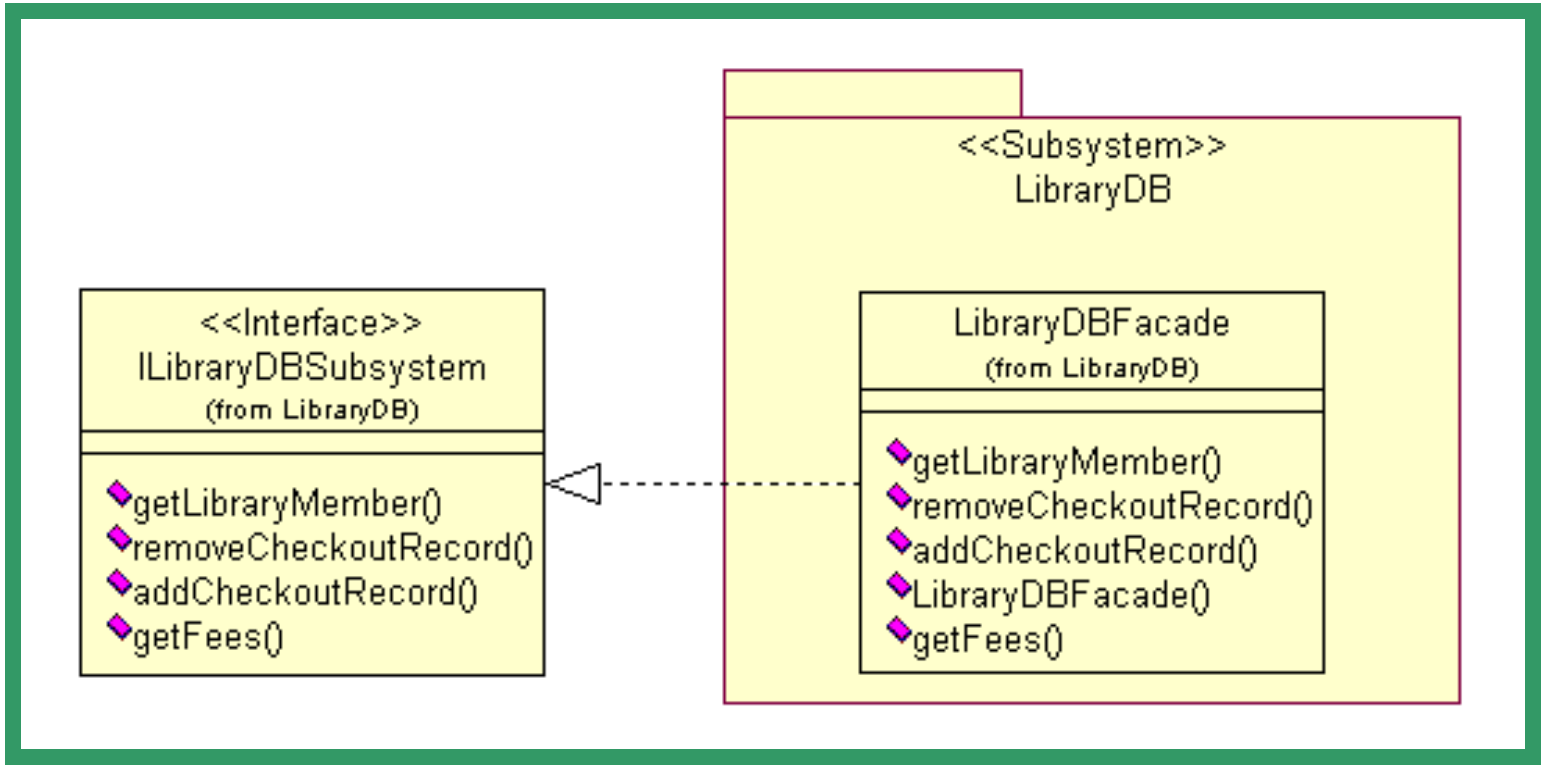
1. In use case analysis we identified classes that could potentially become subsystems from the perspective of the problem domain
2. In architecture design we identified subsystems and designed the subsystem interfaces and dependencies
3. In subsystem design we specify in detail the internal classes and operations of each subsystem

Subsystem Design: Objectives

- Step 1 - Specify detailed operation of subsystems
  - a sequence diagram for each subsystem method defined in our subsystem interface
- Step 2 – Specify the static associations between all the classes in the subsystem
  - Multiple VOPCs for a large subsystem
  - One for our SectionRegistration subsystem.

Subsystem Responsibilities

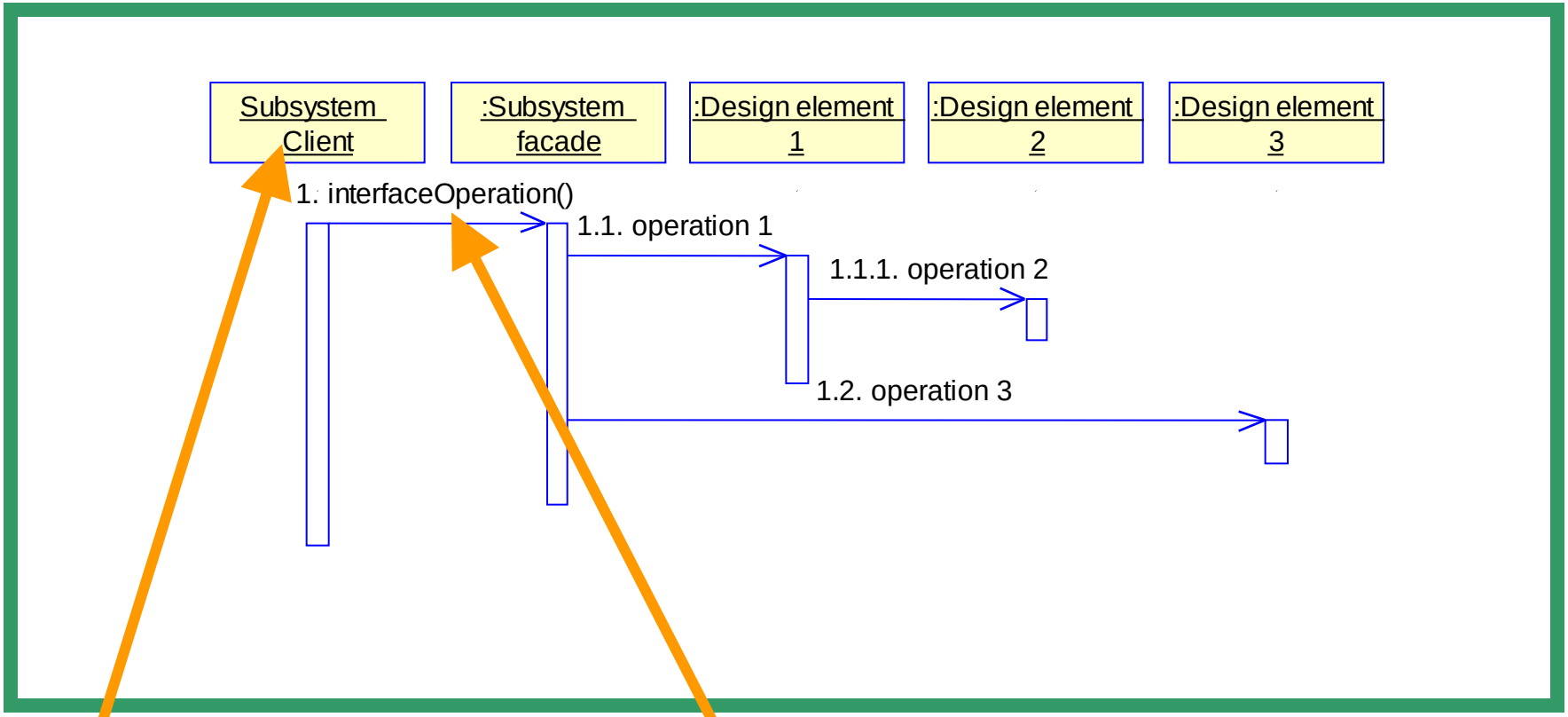
- The subsystem interface defines the responsibilities that are realized by the subsystem.



Distribution of Subsystem Responsibilities

- For each interface operation:
  - Identify existing or new classes (and/or subsystems) that participate in the realization of the interface operation
  - Incorporate applicable mechanisms (persistence, distribution, etc)
  - Create one sequence diagram (optional matching collaboration diagram)
- Revisit Architectural Design to rebalance subsystem responsibilities as necessary.

Subsystem Sequence Diagrams



- One sequence diagram per interface operation
- Interaction diagram starts with a generic subsystem client object
- Subsystem interface does not appear on the internal subsystem interaction diagram
- Show only interfaces of other subsystems

Example – A Data Access Subystem

- Why have a Data Access Subsystem?

- What Services Do We Want From Our Data Access Subsystem?

Example – A Data Access Subystem

- Why have a Data Access Subsystem?
  - Multiple uses – DRY
  - Plug compatible if we switch DBs/DBMS
  - P2I
- What Services Do We Want From Our Data Access Subsystem?
  - CRUD operations
  - Query Interface (more abstract than SQL)
  - ORM
  - Transaction control
  - Security

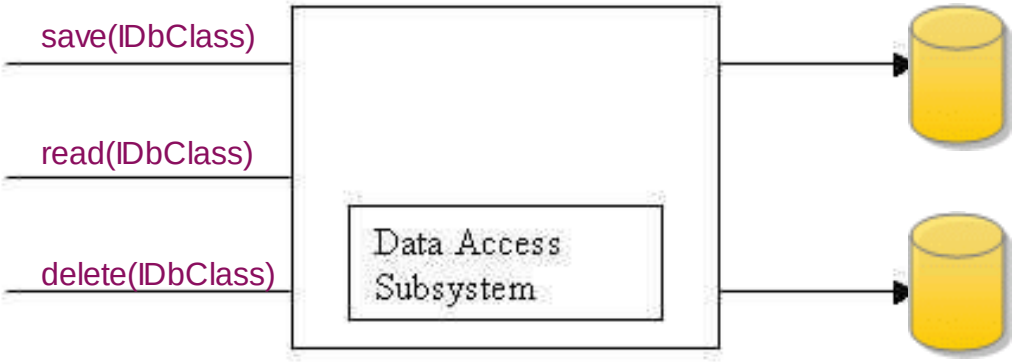
Example: DAO Design Pattern

- ***The Problem:*** Access of a data store involves the same sequence of steps each time. If these repetitive steps are implemented in many places, changes to the way data is retrieved (e.g. new database or new API) will require many changes throughout the code. Need to encapsulate access of the data store in a separate class or subsystem.

- ***The Pattern:*** Create a Data Access Object or Subsystem that receives data read/write requests in a generic way. It encapsulates the routine tasks of database connection and SQL query execution. The special data needs of each particular entity class are encapsulated by a suitable data transfer object (DTO) -- in other words, use the Command Pattern.

- This design pattern can be used as part of the persistency design mechanism

How To Build Your Own Data Access Subsystem



- The Data Access Subsystem must provide a generic way of accessing the basic operations needed in communicating with the database -- a typical subsystem interface would include read, save, and delete operations
- The subsystem should not be required to know about the classes and systems that will use it. Therefore, to get the information it needs, the Data Access Subsystem requires the client to pass in an appropriate command object
- The questions the Data Access Subsystem requires the client to answer are:
  - which database should I use?
  - what query do you want me to run?
  - what do you want me to do with the results?

The Command Object For Data Access

- A command object for data can answer the questions required by the Data Access Subsystem. It is the object that mediates between data and database access
- In this example, the command object is called a "DbClass". Every command object implements the IDbClass interface (thereby answering the necessary questions)
- An entity class that needs to be read, saved or deleted, like ***Address***, will have its own command object, called in this case ***DbClassAddress***

```
public interface IDbClass {
    public void buildQuery();
    public String getDbUrl();
    public String getQuery();
    public void populateEntity(ResultSet resultSet)
}
```

- To save an address, the Address object is inserted into DbClassAddress, and the DbClassAddress is then told to send itself to the Data Access Subsystem.

A Hand Built Data Access Subsystem

- For each entity class, provide a corresponding concrete DbClass that implements IDbClass.

  (Example: DbClassAddress provides database access for the Address persistency class)

- The DbClass is a *data transfer object.* It is a kind of container class that knows:
  - details about reading and persisting data.
  - encapsulates SQL queries
  - knows how to build instance of entity from raw data

Hand Built Data Access Subsystem (continued)

- Implement the IDbClass methods *buildQuery*, *getDbUrl*, *getQuery*, and *populateEntity* (the latter is used only for reads)
  - getDbUrl returns a pointer to the appropriate database
    - Example of dburl:
      - *jdbc:mysql:///accountsdb* (for a MySql db)
  - getQuery returns the value in the *query* variable, which is the SQL statement, created in *buildQuery*

Hand Built Data Access Subsystem (continued)

- buildQuery is the code that stores the needed SQL statement in the *query* variable

- populateEntity -- after a read is performed, a ResultSet object is returned, and this is passed to populateEntity; the code in populateEntity extracts the data from the result set and inserts it into the appropriate entity class, which will later be retrieved by the calling client

Hand Built Data Access Subsystem (continued)

Overview of Usage

1. Controller - new() concrete DbClassX
2. insert the X Entity class
3. call DbClass method (for save, read, etc.)
4. DbClass method calls one of the Data Access Subsystem operations (save, read, delete, etc.)
   - DbClassX passes itself in as an argument

Sample Code

- FROM CustomerController

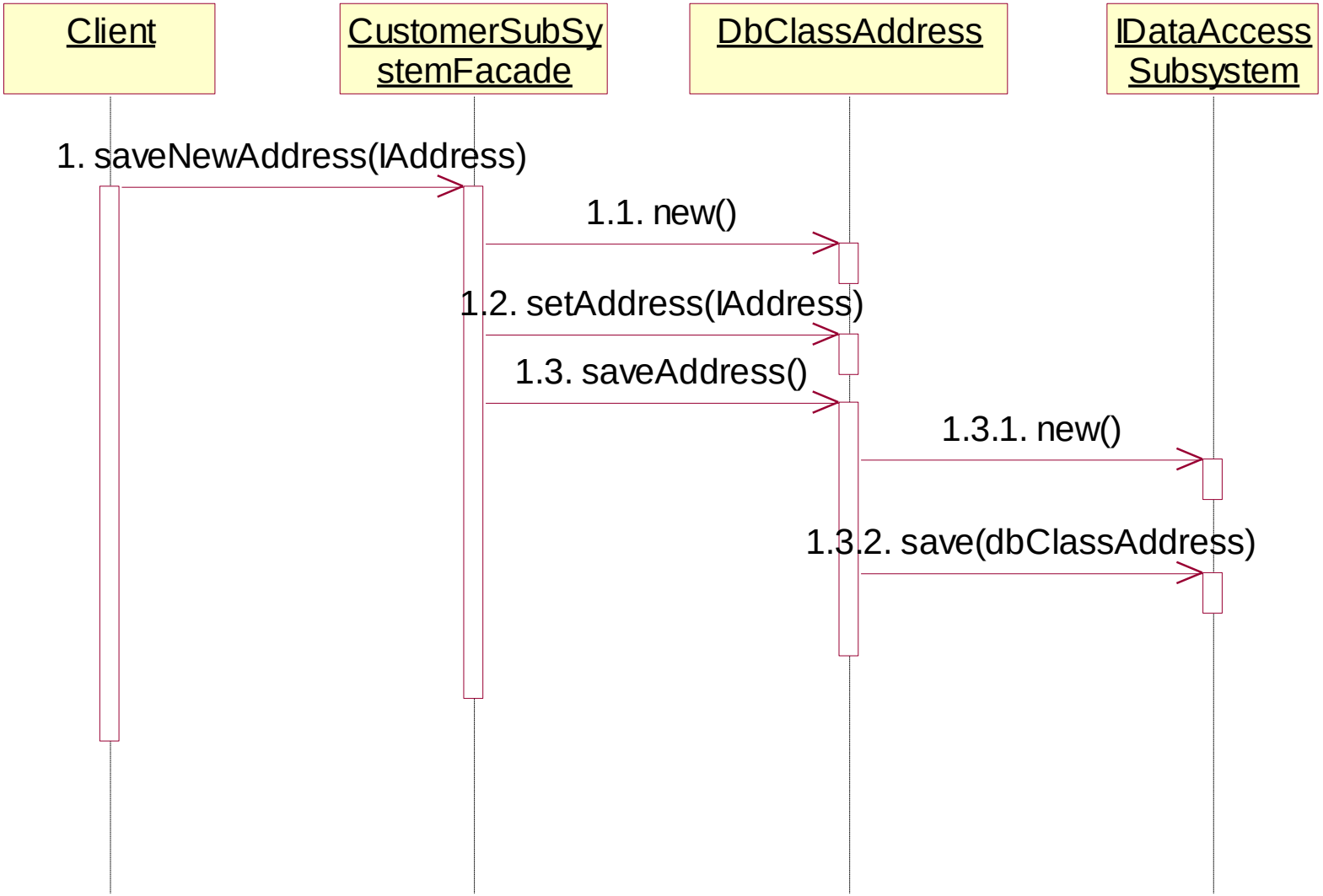
```
public void saveNewAddress(IAddress addr) throws DatabaseException {
    DbClassAddress dbClass = new DbClassAddress();
    dbClass.setAddress(addr);
    dbClass.saveAddress(custId);

}
```

- FROM DbClassAddress

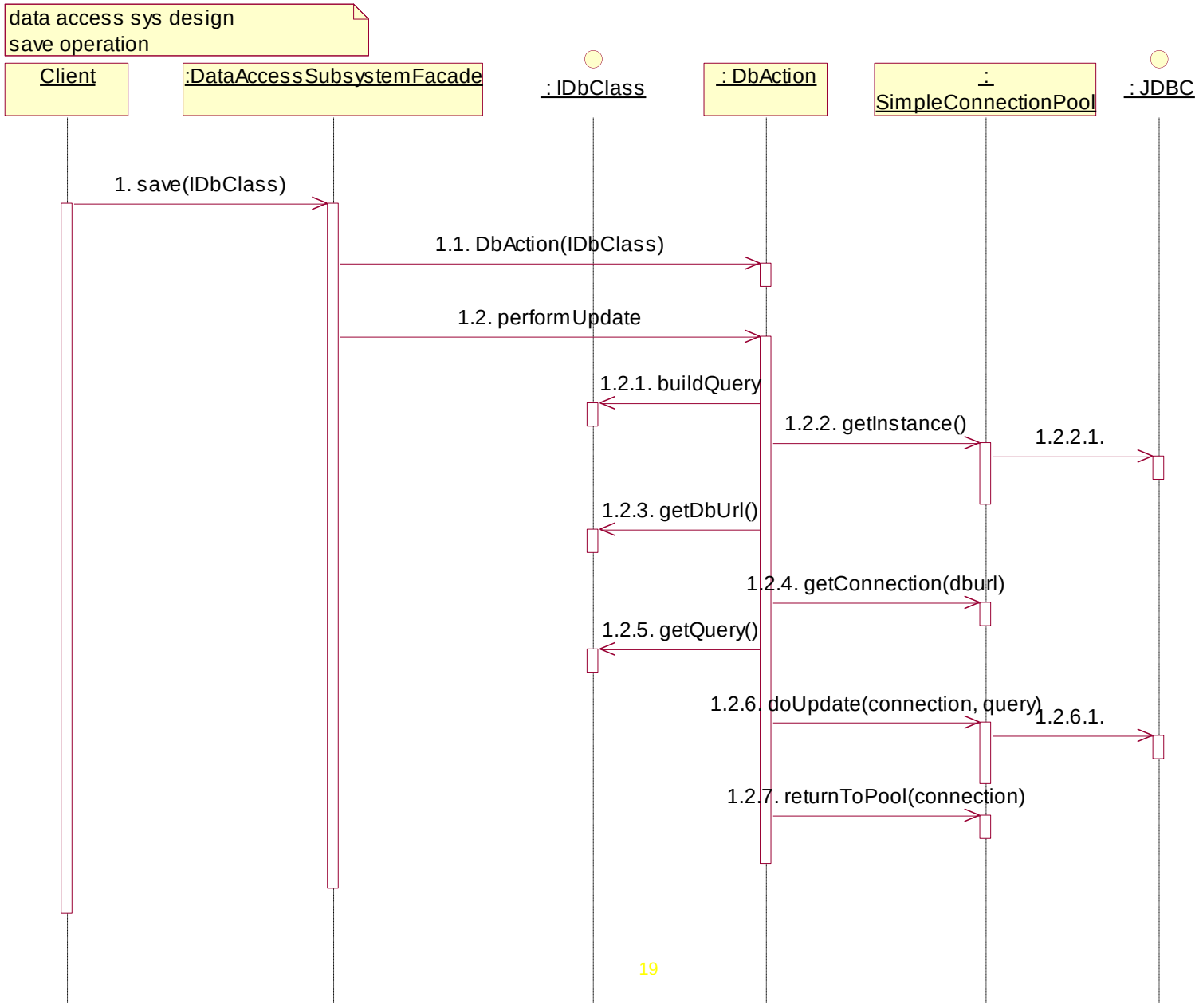
```
public void saveAddress(String custId) throws DatabaseException {
    this.custId = custId;
    queryType = SAVE;
    dataAccess = new DataAccessSubsystemFacade();
    dataAccess.save(this);
}
```

What is another option to the ***new*** of DataAccessSubsystemFacade ?

Example: Customer Subsystem Interaction



Data Access Subsystem Sequence Diagram



Data Access Object and JPA

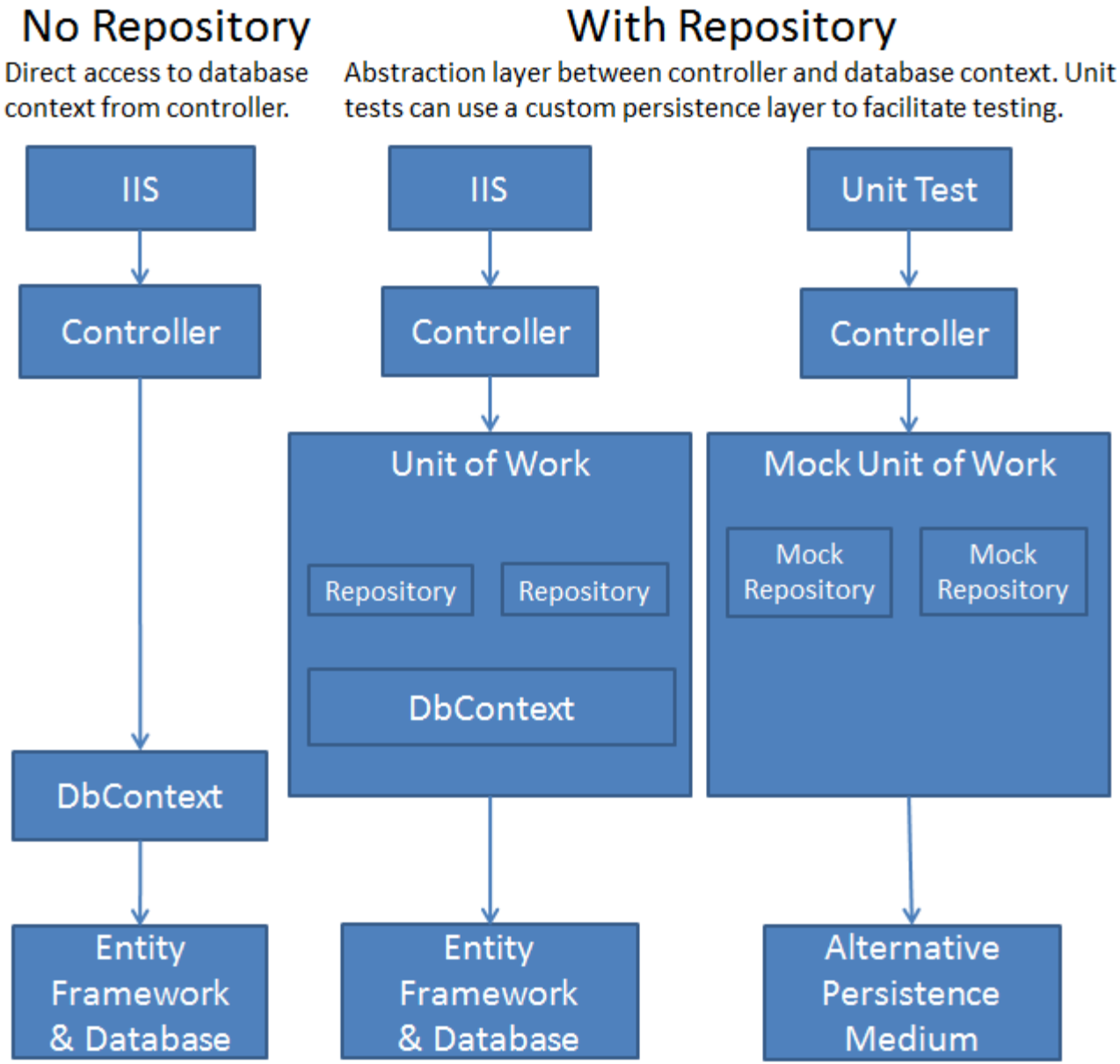
1. In NetBeans we define (when starting the project and when creating the first Entity class):
   1. Persistence Unit
   2. Persistence Provider
   3. Data Source – JNDI name
   4. Connection Pool
   5. DB Driver
   6. RDBMS
   7. DB
   8. JDBC URL
   9. DB schema
2. The result can be seen in the configuration file persistence.xml

Spring Hibernate Data Access Object

In our Spring Boot Demo project StudentSpringBoot Application.properties has the following:

```
spring.datasource.url = jdbc:mysql://localhost:3306/msched?createDatabaseIfNotExist=true

spring.datasource.driverClassName = com.mysql.jdbc.Driver

spring.datasource.password=root
spring.datasource.username=root

spring.jpa.properties.hibernate.dialect: org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.ddl-auto = create
spring.jpa.show-sql= true
 spring.jpa.properties.hibernate.hbm2ddl.import_files: import_data.sql
spring.messages.basename=messages
```

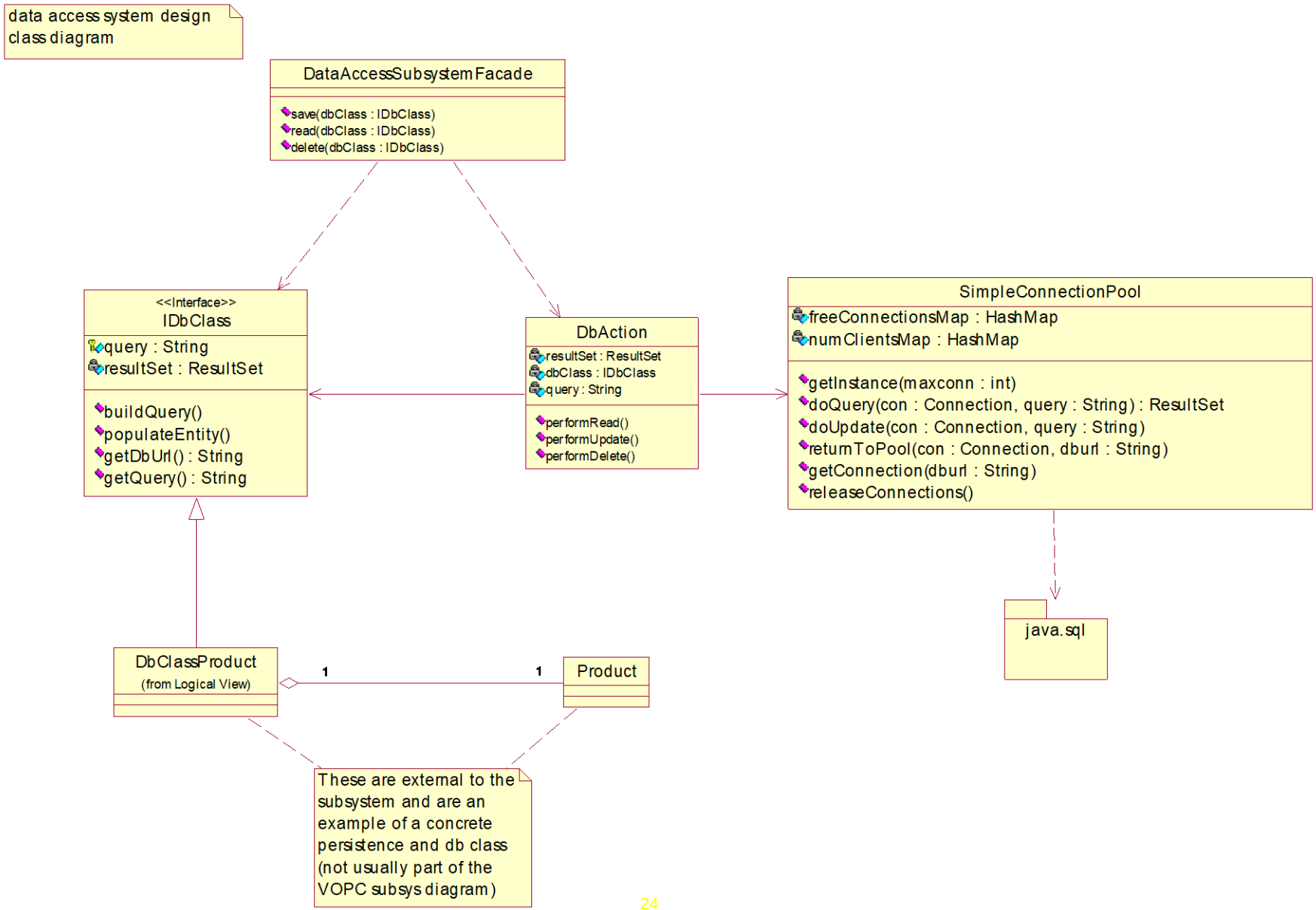
.NET Repository Pattern



Subsystem Design Topics

- Step 2 – Build one VOPC for the subsystem.
- For example - the VOPC for the hand-built data access subsystem follows:

Data Access Subsystem VOPC



A Pattern for Subsystem Design and Implementation

- Each operation on the subsystem interface is a subsystem-specific scenario, analogous to use case scenarios

- Interface implemented by a Façade - A singleton may be a good choice.
- A Façade is a design pattern we can use to encapsulate a subsystem
  - Hides internal classes
  - Forwards requests for services to internal classes

- Frequently, the subsystem will contain a Mediator Pattern or Controller
  - Similar to Use Case controllers
  - Mediator responsible for internal “workflow” of subsystem

Section Registration Subsystem Design

As a group review your student section registration subsystem:

- How many methods does your subsystem interface have?
- Do you have a façade class that realizes all the interface methods?
- Try (at least) one sequence diagram for one of those methods.
- Build a VOPC for your subsystem

Microservices

- Microservices - like the subsystem concept with greatly increased independence between the parts of the system.
- Like subsystems - break up the development into manageable chunks
- Microservice frameworks also can improve:
  - Reliability -- redundant microservices
  - Performance – add additional microservices as the load increases
  - Load balancing - across multiple microservices
  - Deployment – microservices rolled out independently

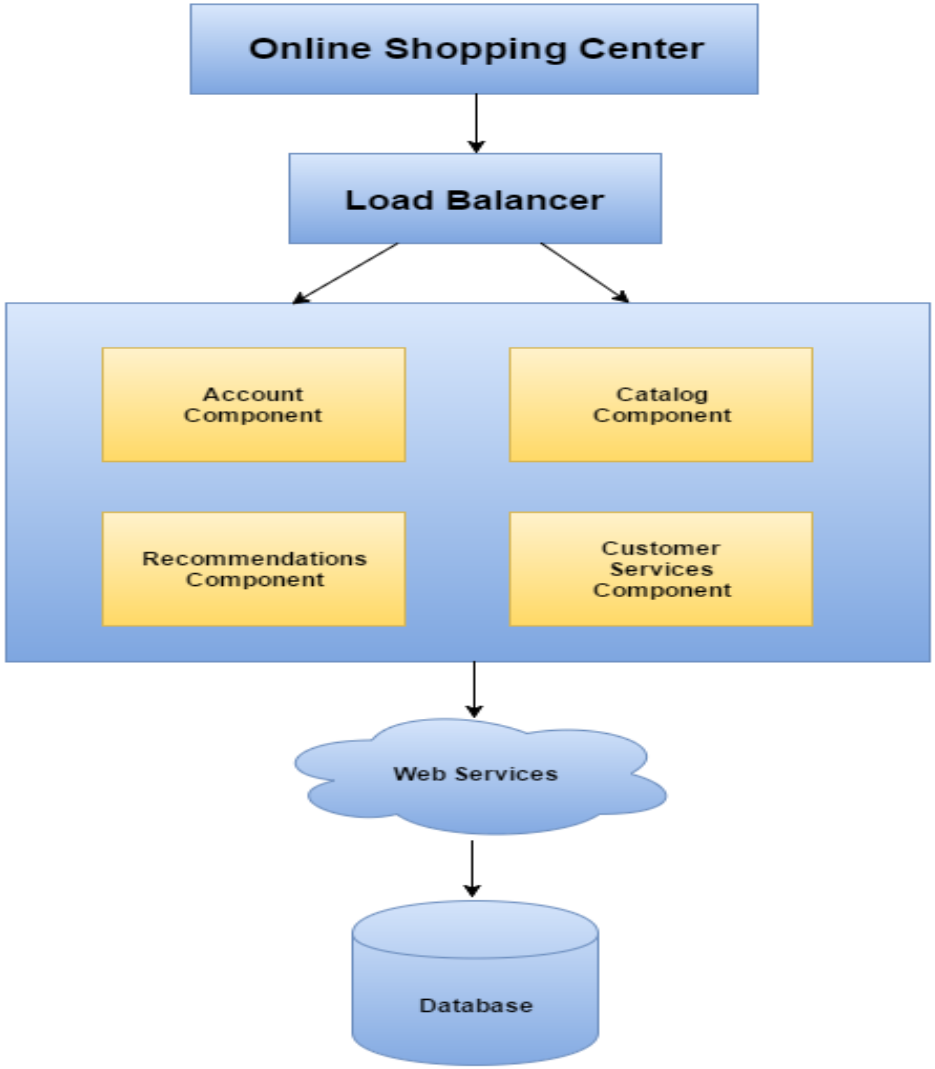


Figure 1: Conventional Approach

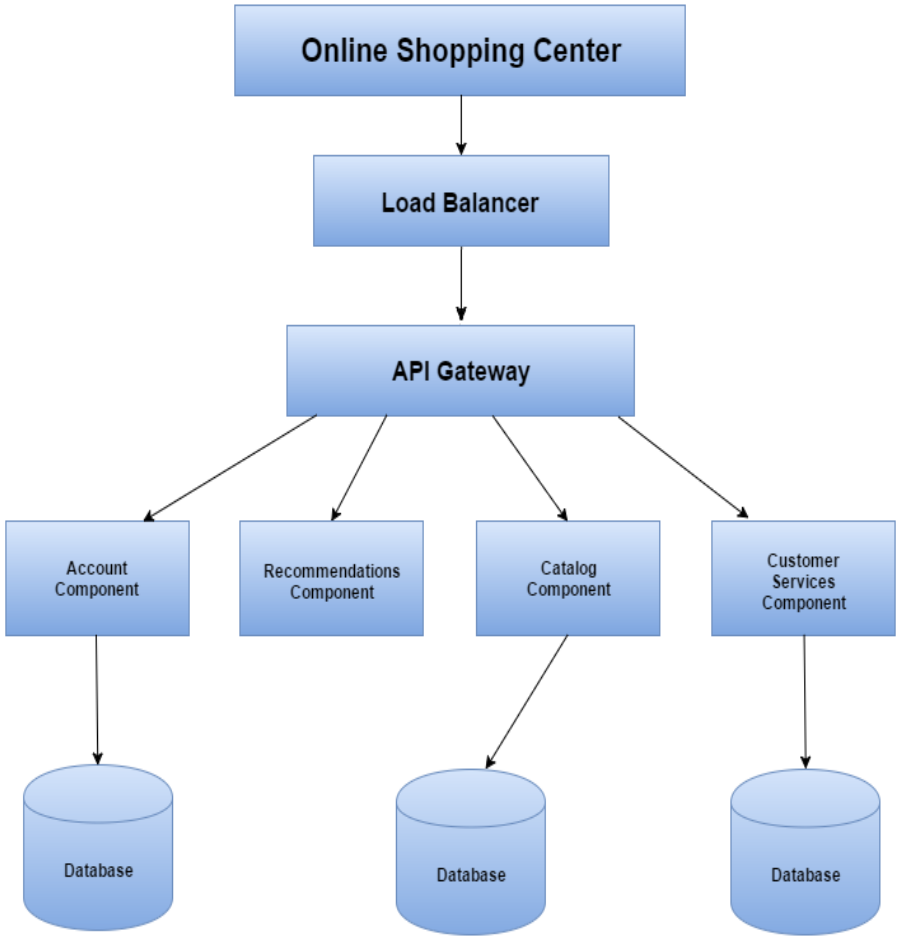
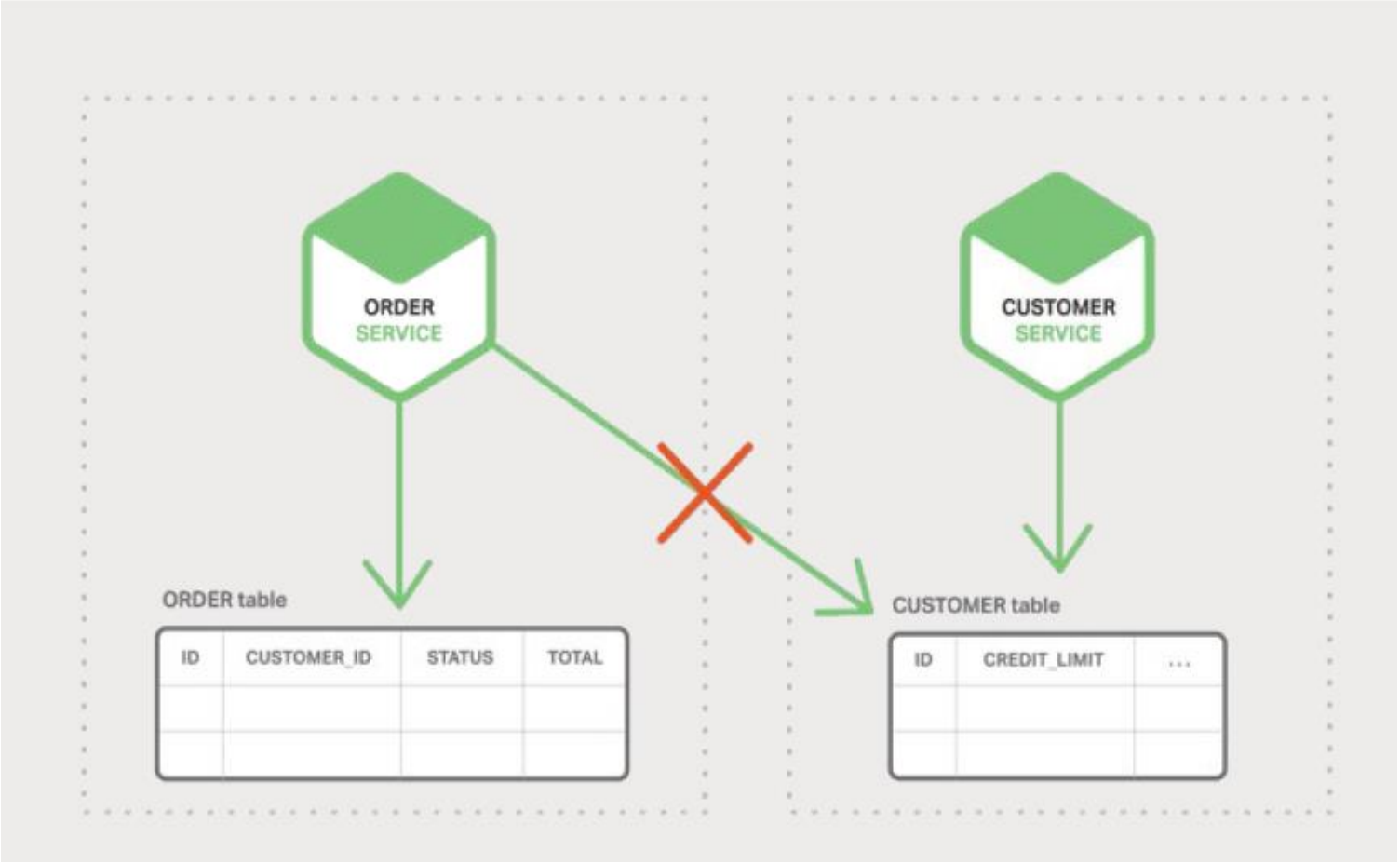
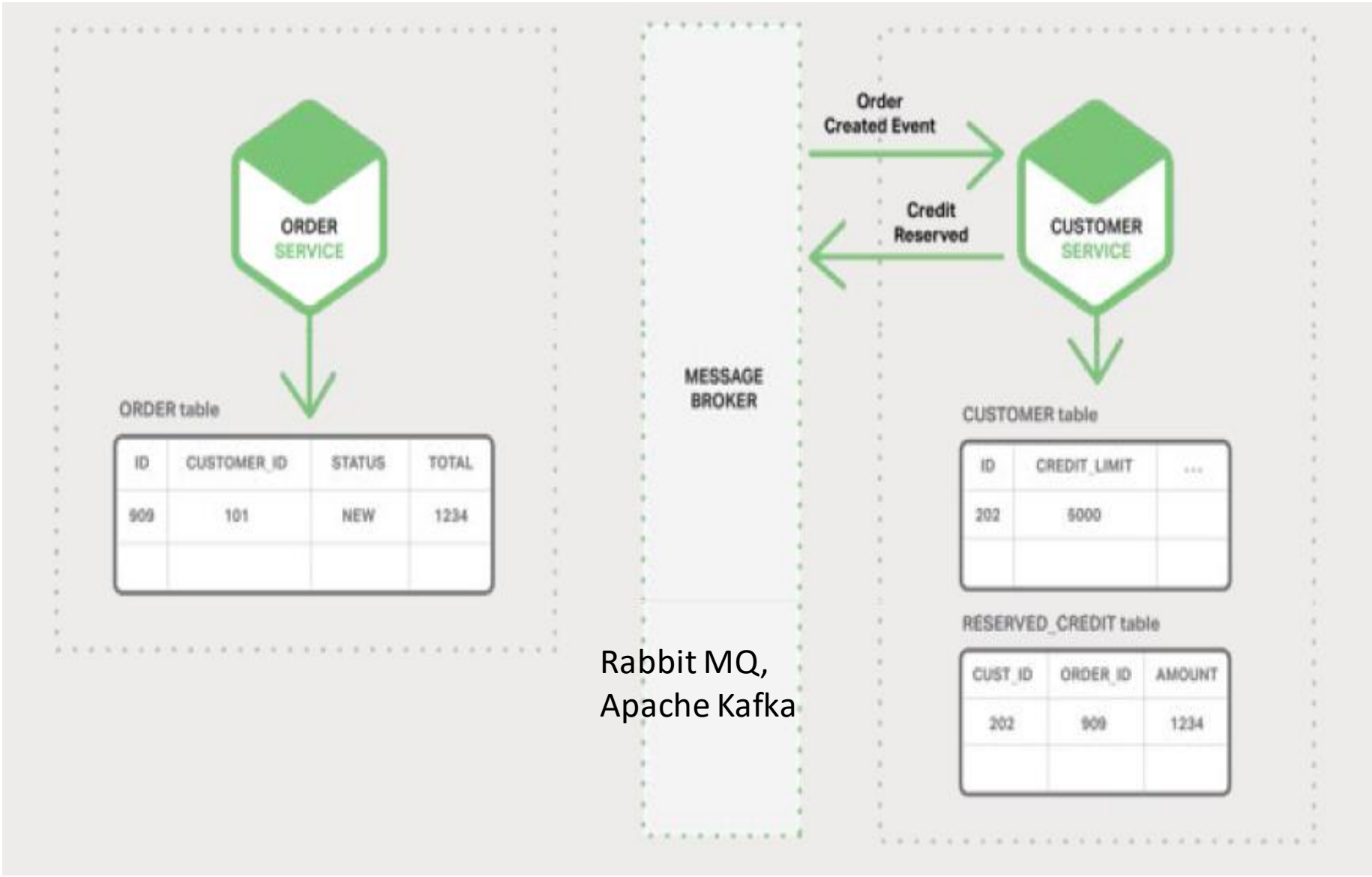
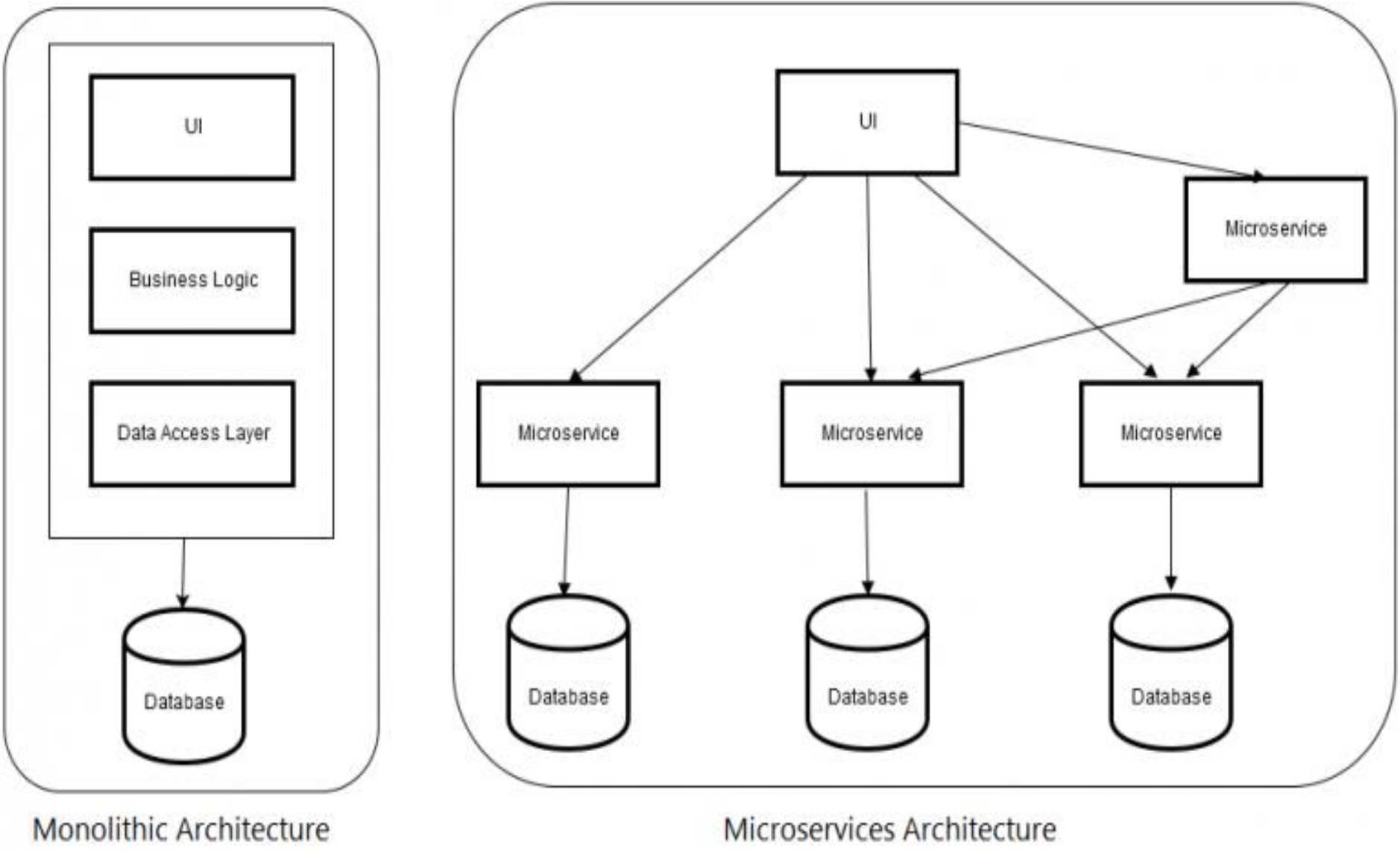


Figure 2: Micro Services Approach


ORDER
SERVICE
CUSTOMER
SERVICE
ORDER table
ID
CUSTOMER_ID
STATUS
TOTAL
CUSTOMER table
ID
CREDIT_LIMIT
...

Event Driven Architecture

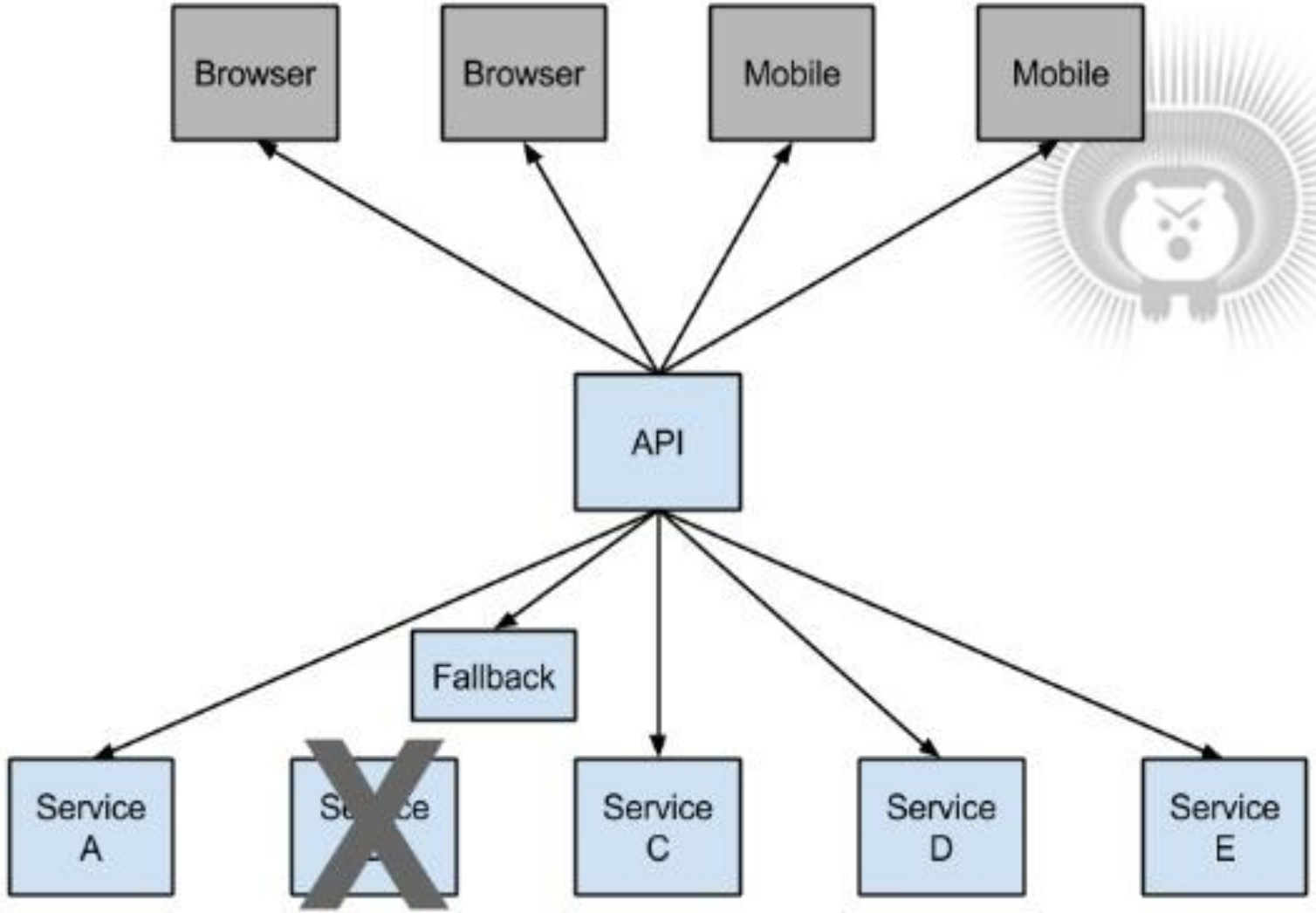




Monolithic Architecture

Microservices Architecture

Circuit Breaker



http://cloud.spring.io/spring-cloud-netflix/images/HystrixFallback.png

Review Questions

1. List some advantages of using subsystems.
2. What are the main inputs for subsystem design?
3. What are the main outputs for subsystem design?
4. What are some differences between microservices and subsystems.

Subsystem Design Summary

1. Purpose of subsystem design is to model the internal behavior of subsystems.
2. Subsystems are a major means of decomposing large systems and promoting parallel development and reuse.
3. The steps for designing the internal behavior of a subsystem are very similar to the steps of use case analysis and use case design.

   a) A sequence diagram for each of the subsystem interface methods.

   b) One VOPC for the subsystem

Subsystem Design Checklist

- Are all interface operations realized by the subsystem façade class, which distributes the realization to other classes or subsystems?
- Ensure that the subsystem only exposes contained behavior through interfaces, and that all the elements within the subsystem are encapsulated.

Subsystem Design – Main Points

Order is Present Everywhere

During subsystem design we work out the details of subsystem operations. The steps are basically the same as those of use case analysis and use case design, but now our actors are any clients of the subsystem.

MAIN POINTS

As in use case design, the main artifacts are interaction (dynamic) and class (static) diagrams of the subsystem elements, both of which emerge on the basis of the architecture design model.

The designer pays close attention to subsystem dependencies upon the rest of the system and attempts to minimize dependencies of subsystems with external elements. Good encapsulation means minimal dependencies, ease of maintenance, and maximum reuse possibilities. Microservices design takes good encapsulation and independence of subsystems to the fullest expression.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Transcendental consciousness is a state of in which the macro level of life is directly experienced at the individual level.

Impulses within the transcendental field: The steps of expression of pure consciousness in individual life exactly parallel the steps of expression in the unfoldment of creation itself. We see order is present at the macro and micro levels of life.

Wholeness moving within itself: In unity consciousness, the ultimate inner value, experienced as Transcendental Consciousness, is recognized as the true nature of all other particulars of experience. Inner and outer values of life are completely unified.