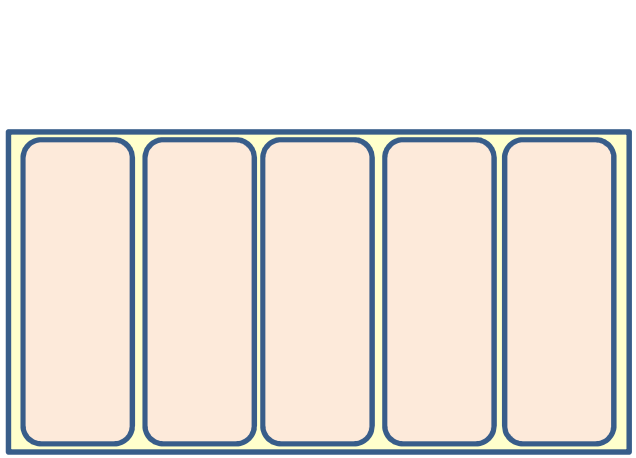
MIDTERM REVIEW

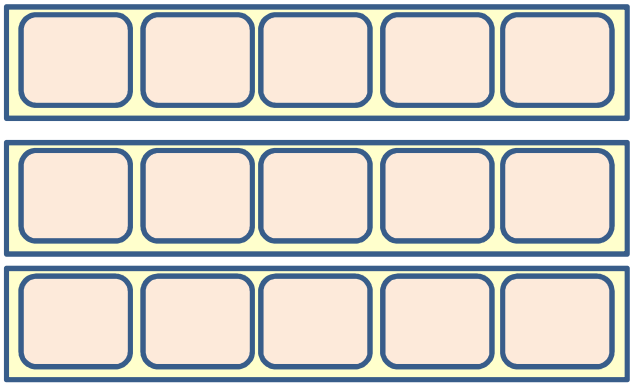
Midterm

- **Dalby Hall**
- **9:30 – 12:00**
- Closed book/notes.
- No personal items including electronic devices (cell phones, computers, calculators, PDAs).
- No additional papers are allowed. Sufficient blank paper is included in the exam packet.
- Restroom and other personal breaks are not permitted.
- Bring only pen, pencil (eraser)

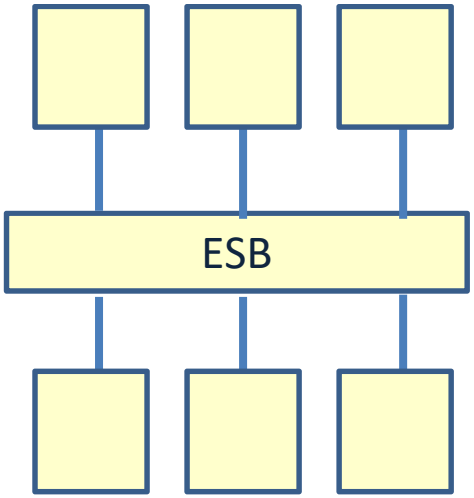
Architecture styles



Layering



Components



Service Oriented Architecture

LESSON 1

Lesson 1 Software Architecture topics

- Software qualities
- Architecture (design) principles
- Communicating architecture
  - Context diagram
  - Container diagram
  - Component diagram , sequence diagram of components
  - Class diagram , sequence diagram of classes
- Identify risk
- Clustering
- Failover

Agile architecture

Functional
- Use cases
- User stories
- Scenarios
- Processes
- Business rules
- Acceptance criteria
- …

Qualities
- Performance
- Scalability
- Reusability
- Security
- Availability
- Modifiability
- …

Constraints
- Budget
- Deadlines
- Enterprise arch.
- Experience
- …

Principles
- Keep it Simple
- Separation of concern
- Loose coupling
- Information hiding
- Modularity
- Flexibility
- …

Architectural requirements

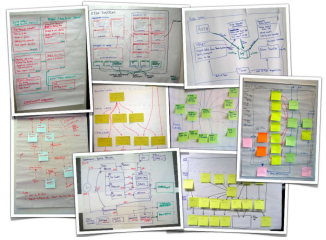
Context diagram: Shows the big picture.

Container diagram: Shows the high-level software architecture using containers and technology choices.

Component diagram: Decompose each container further into a number of distinct components.

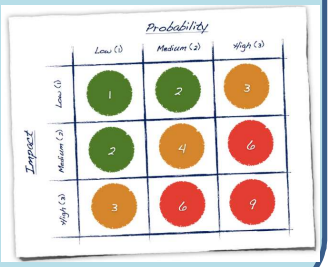
Detailed diagrams: Class diagram, sequence diagram, deployment diagram.

Design and visualize the architecture

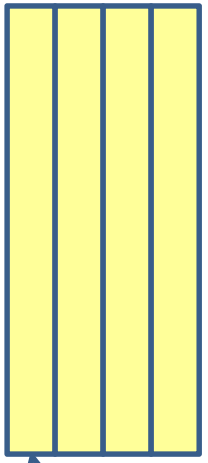


Identify risks

Risk storming



Refine the architecture in the Sprints



Proof the architecture.
Do the most risky things first.

Architecture (design) principles

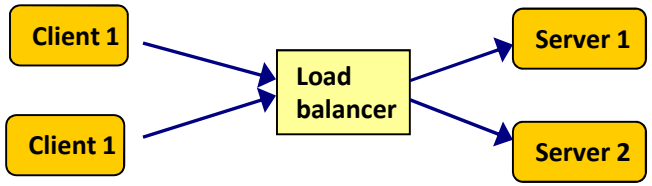
- Keep it simple
- Keep it flexible
- Loose coupling
- Separation of concern
- Information hiding
- Principle of modularity
- High cohesion, low coupling
- Open-closed principle

Most important qualities

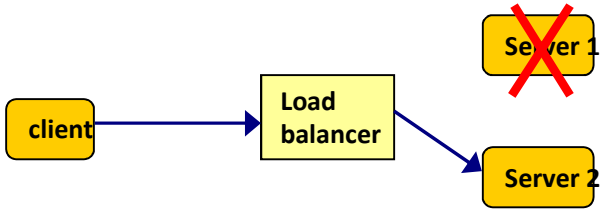
- Performance
- Scalability
- Availability
- Reliability
- Maintainability
- Security
- Interoperability
- Usability

Clustering

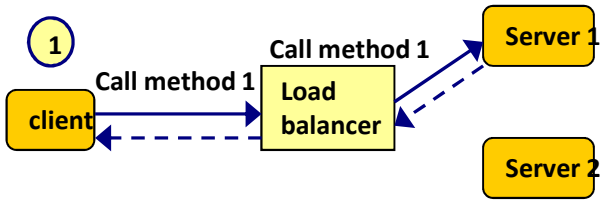
- Load balancing

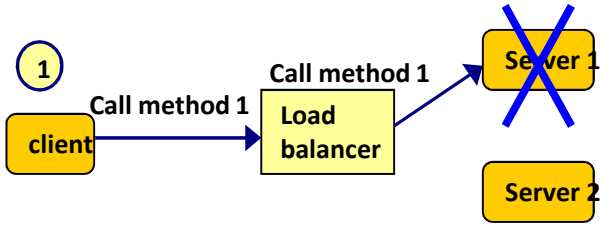


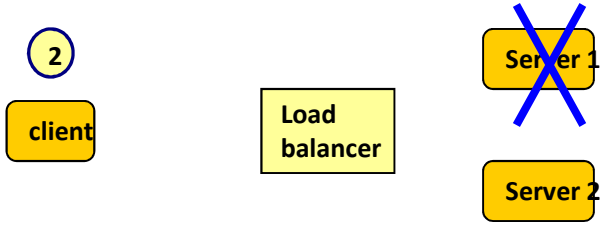
- Failover

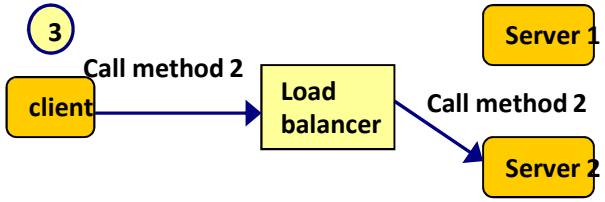


What does failover solve?









- Failover works for crashes between distributed calls.
- But what if the crash happens during the call?

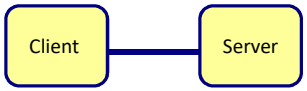
Failover reduces the change on errors, but cannot prevent all errors

LESSON 2

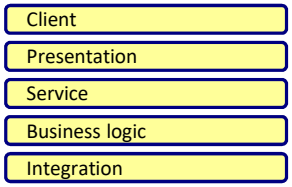
Lesson 2 Application Architecture topics

- Architecture styles
  - Layering
  - Client-server
  - Pipe and filter
  - Master-slave
  - Microkernel
- Service class
- DOA/repository class
- Proxy/Gateway class
- Relational database versus nosql database
- Scaling databases
- Brewers cap theorem
- Strict consistency – eventual consistency

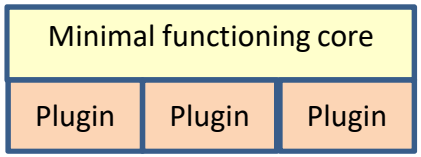
Architecture styles



Client-server

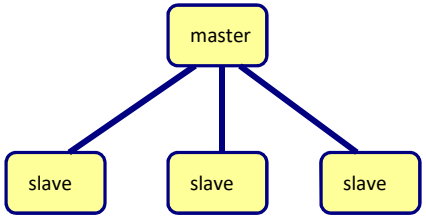


Layering



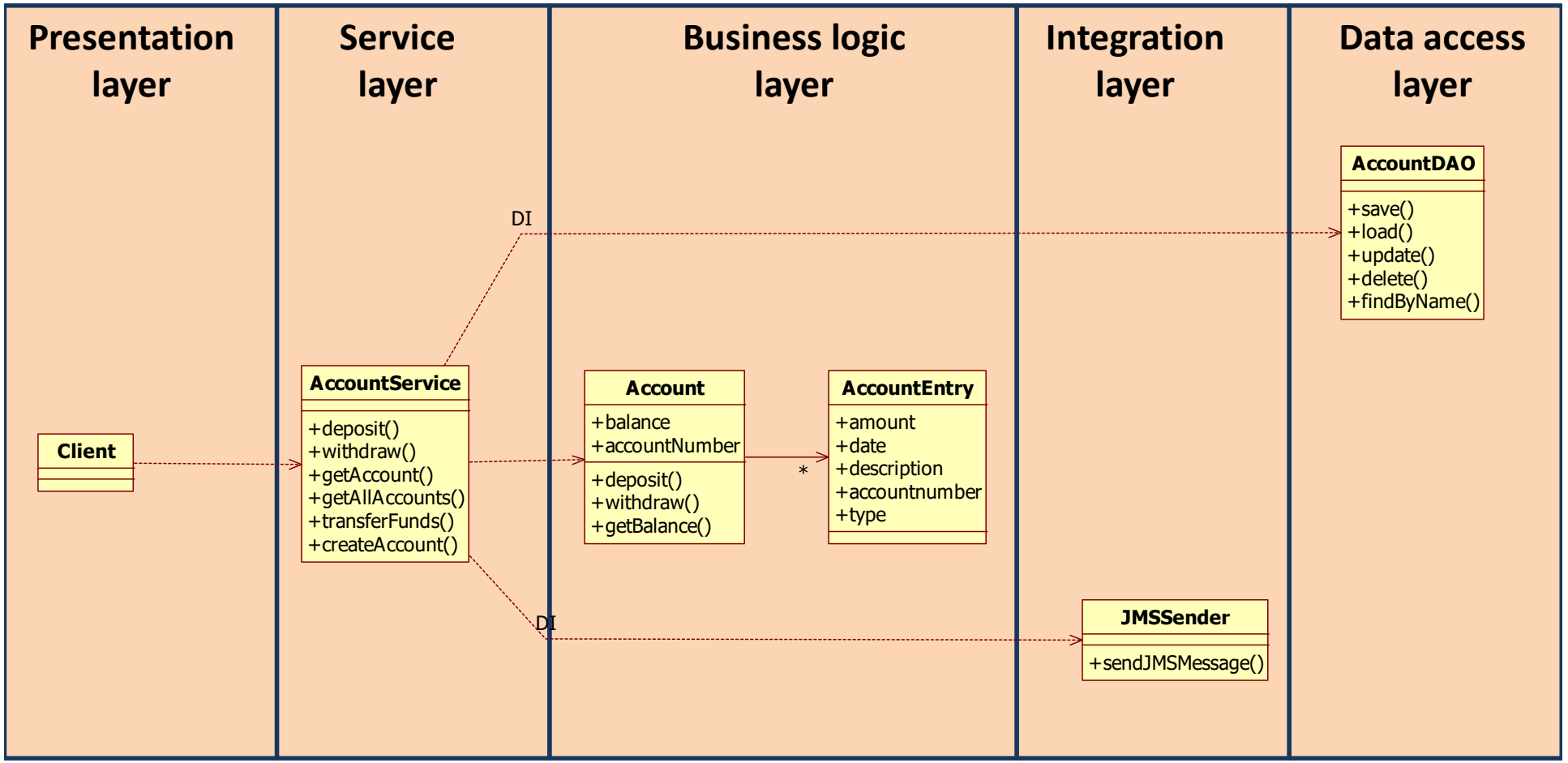
Microkernel

Pipe-and-Filter



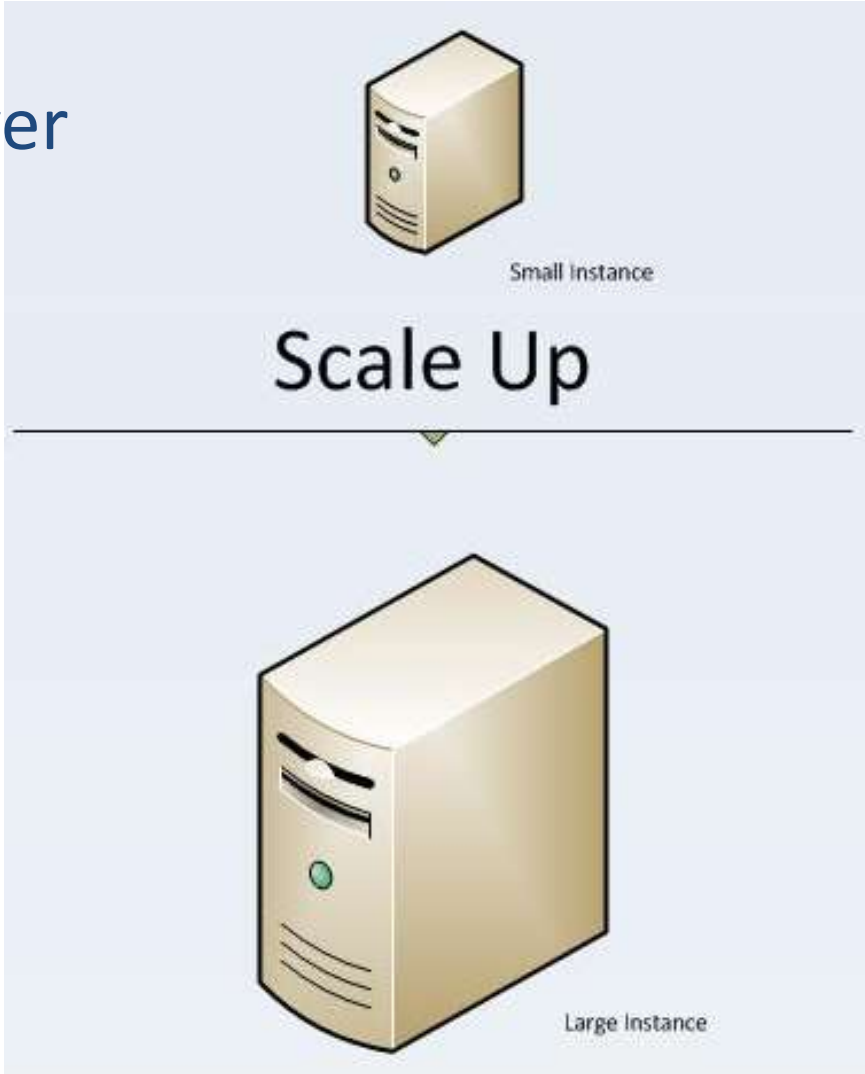
Master-Slave

Application layers



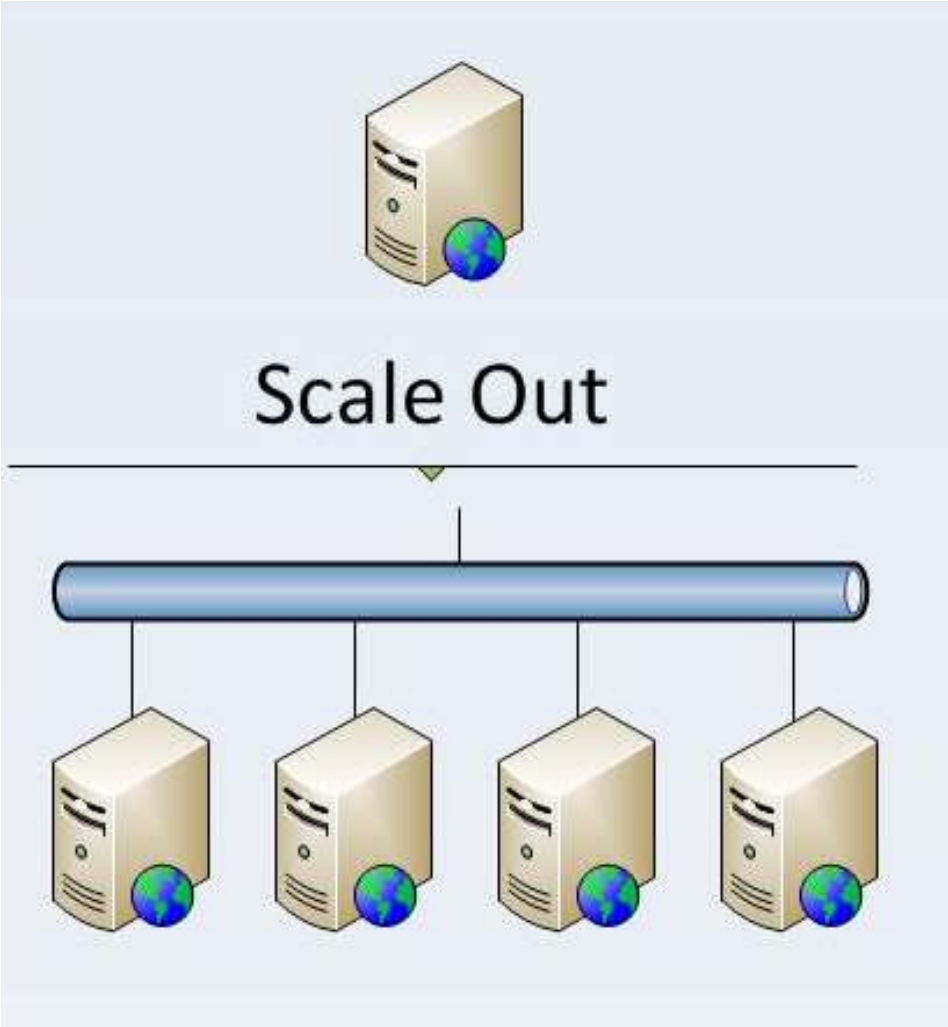
Vertical Scaling

- Scale up
- Use a more powerful server
- Single point of failure
- Upgrading results in downtime
- Limitations
  - Cost
  - Software does not use all resources
  - Hardware
- Vendor lock-in



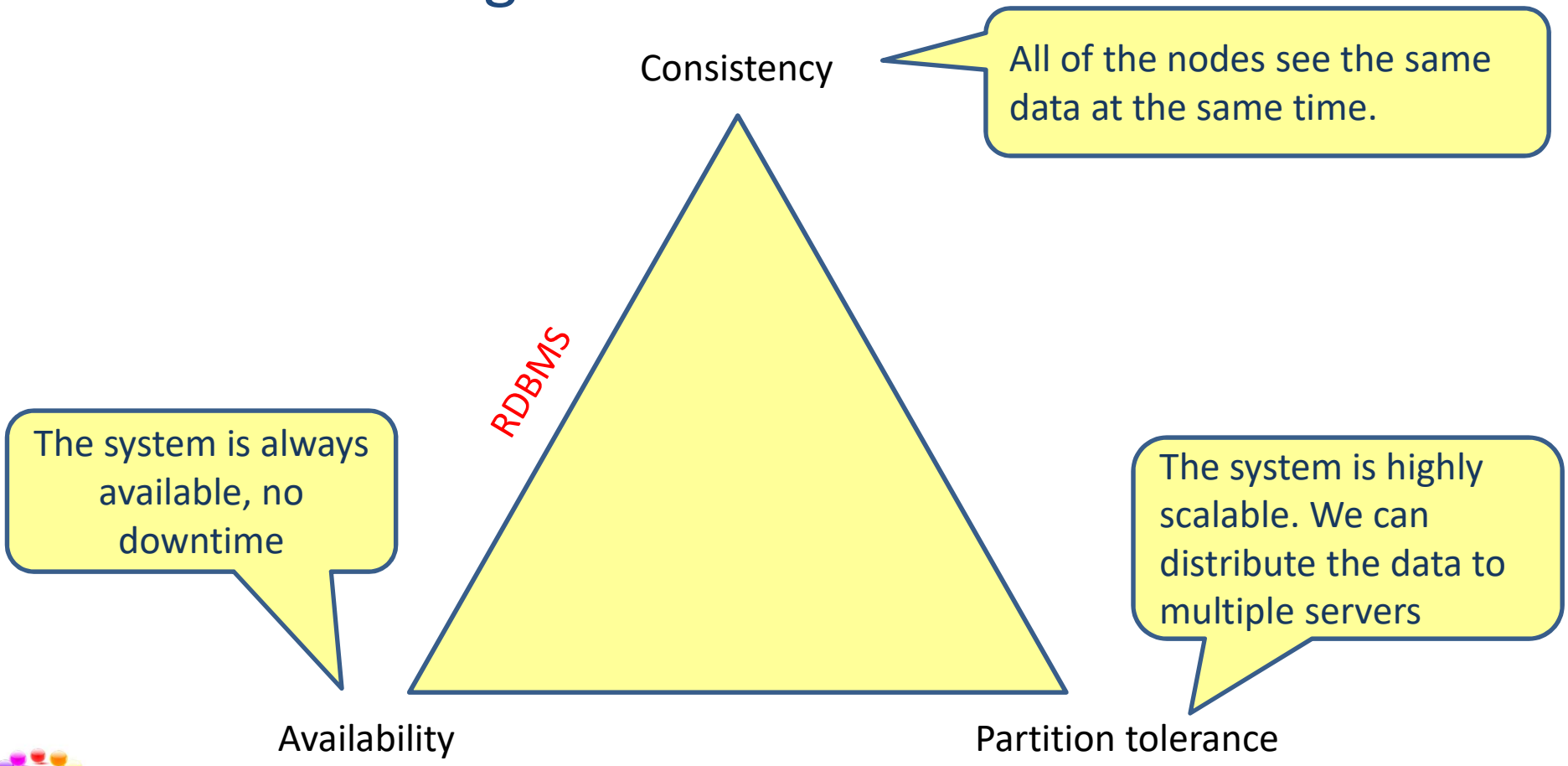
Horizontal scaling

- Scale out
- Divide the data over multiple servers
- Easy to add more servers
  - Without downtime

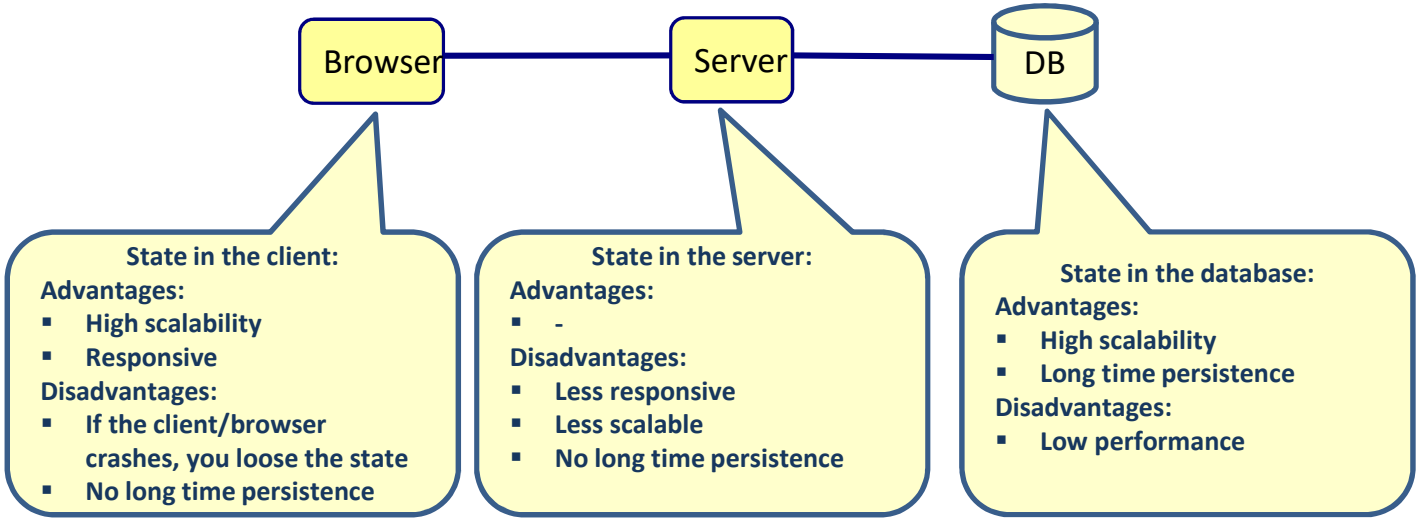


Brewer‘s CAP Theorem

- A distributed system can support only two of the following characteristics



Where to store state?



Integration techniques

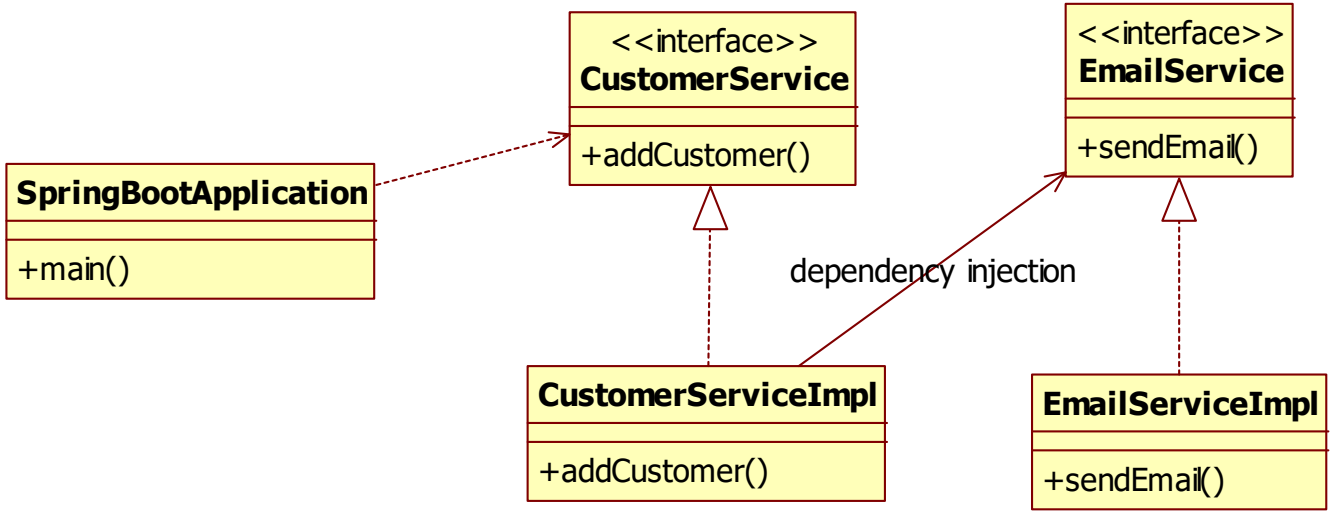
technique	advantage	disadvantage	When to use?
RMI		Only Java to Java High coupling	Never
Messaging	Buffer Low coupling Asynchronous	You need messaging middleware	Between applications within the same organization When you need asynchronous requests
SOAP	Standards for security, transactions, etc. Standard for interface description	Complex	When you need standards
REST	Simple	No standards for security, transactions, etc No standard for interface description	Everywhere you need synchronous requests
Serialized objects over HTTP	Simpler as RMI Webcontainer functionality	Only Java to Java High coupling	For fast Java to Java integration between applications managed by the same project
Database integration	Simple	High coupling	Never
File based integration		High coupling	If you have to communicate large files

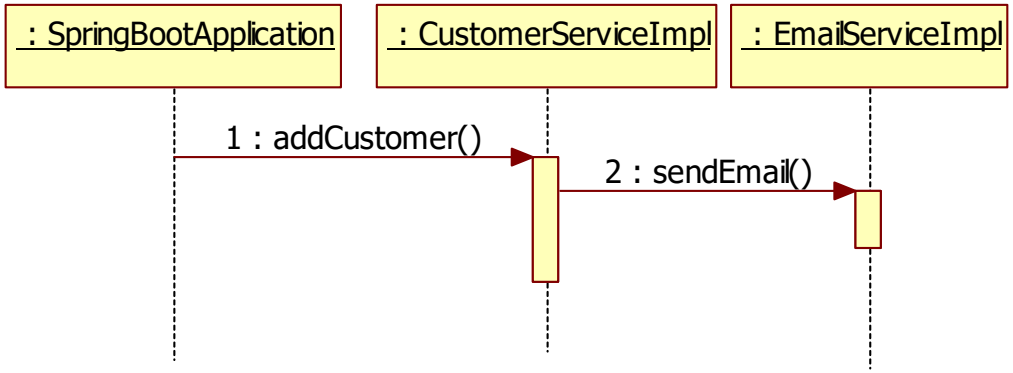
LESSON 3

Lesson 3 Spring Boot topics

- Dependency injection
- REST
- Mongo

Dependency injection





Dependency Injection

```
public interface EmailService {
  void sendEmail();
}
```

```
@Service
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

```
public interface CustomerService {
  void addCustomer();
}
```

```
@Service
public class CustomerServiceImpl implements CustomerService{
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Inject the EmailService in the CustomerService

REST

```
@RestController
public class ContactController {

}
```

```
@RequestMapping("/contact/{firstName}")
public ResponseEntity<?> getContact(@PathVariable String firstName) {
  Contact contact = contacts.get(firstName);
  if (contact == null) {
    return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with firstname= "
      + firstName + " is not available"),HttpStatus.NOT_FOUND);
  }
  return new ResponseEntity<Contact>(contact, HttpStatus.OK);
}
```

```
@RequestMapping(value="/contact", method=RequestMethod.POST)
public ResponseEntity<?> addContact(@RequestBody Contact contact) {
  contacts.put(contact.getFirstName(), contact);
  return new ResponseEntity<Contact>(contact, HttpStatus.OK);
}
```

Mapping annotations

```
@RequestMapping(value = "/add", method = RequestMethod.GET)
```

```
@GetMapping("/add")
```

Same

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
```

```
@PostMapping("/add")
```

Same

```
@RequestMapping(value = "/del", method = RequestMethod.DELETE)
```

```
@DeleteMapping("/del")
```

Same

```
@RequestMapping(value = "/mod", method = RequestMethod.PUT)
```

```
@PutMapping("/mod")
```

Same

Mongo repository

```
public interface StudentRepository extends MongoRepository<Student, String> {

}
```

```
public interface StudentRepository extends MongoRepository<Student, String> {
  Student findByName(String name);
}
```

Define your own method based on a standard convention

application.properties

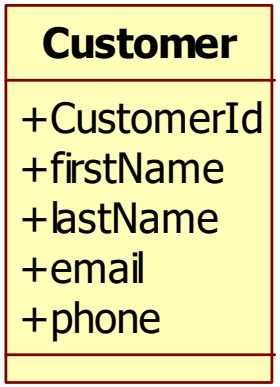
```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=testdb
```

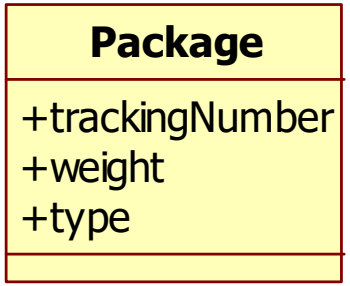
LESSON 4

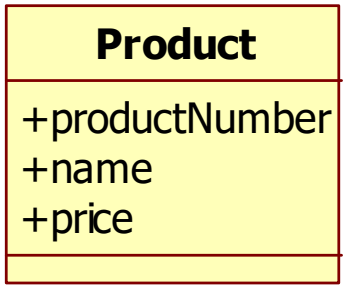
Lesson 4 DDD topics

- Entity
- Value object
- Domain service
- Events
- Aggregate root
- RestTemplate

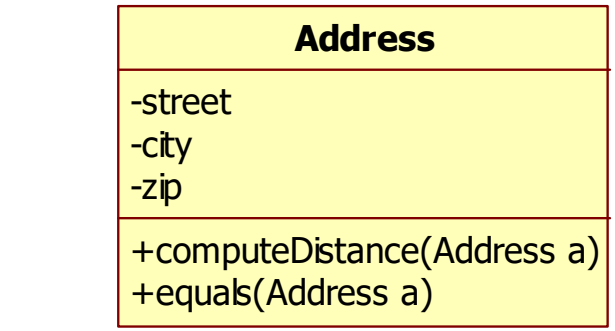
Example entity classes

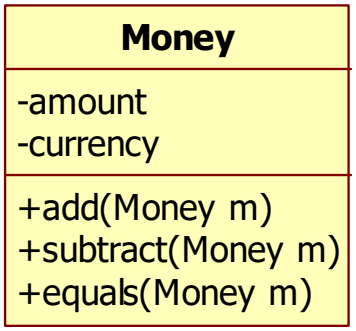


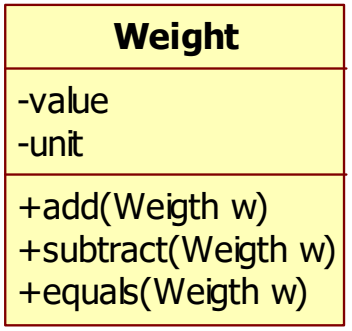


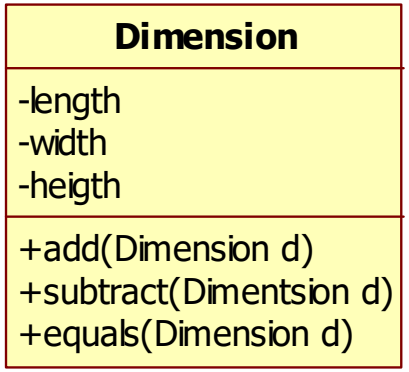


Example value object classes



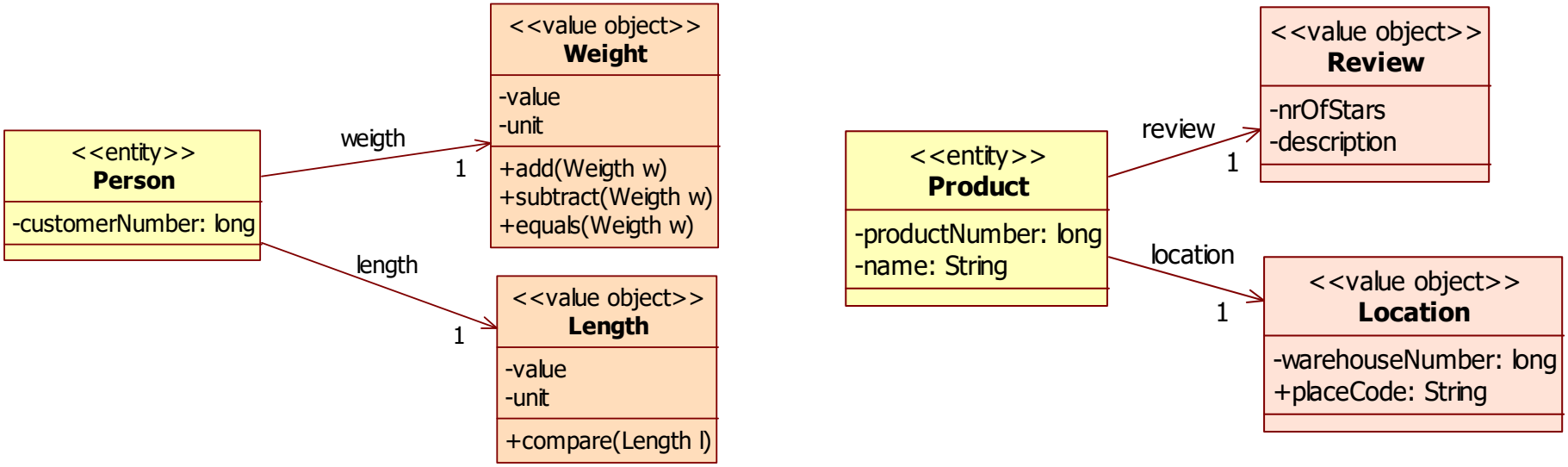






No identity

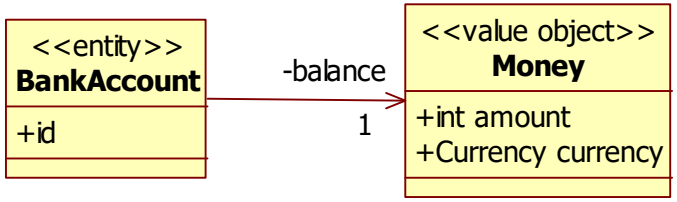
- Value objects tell something about another object



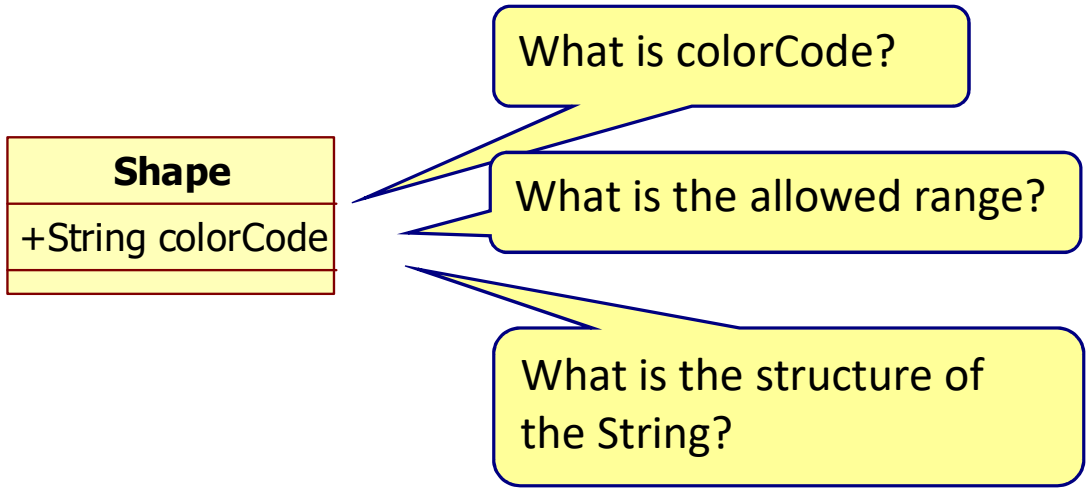
- Technically, value objects may have IDs using some database persistence strategies.
  - But they have no identity in the domain.

When to use value objects?

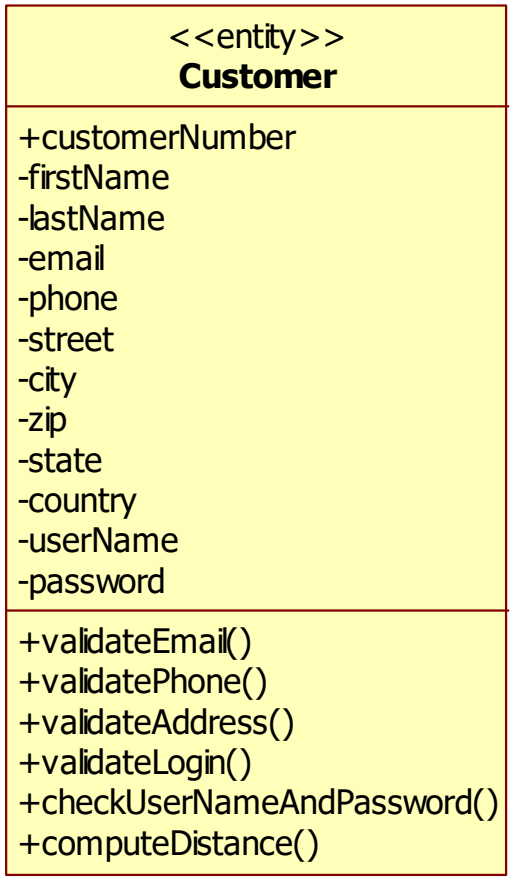
1. Representing a descriptive identity-less concept

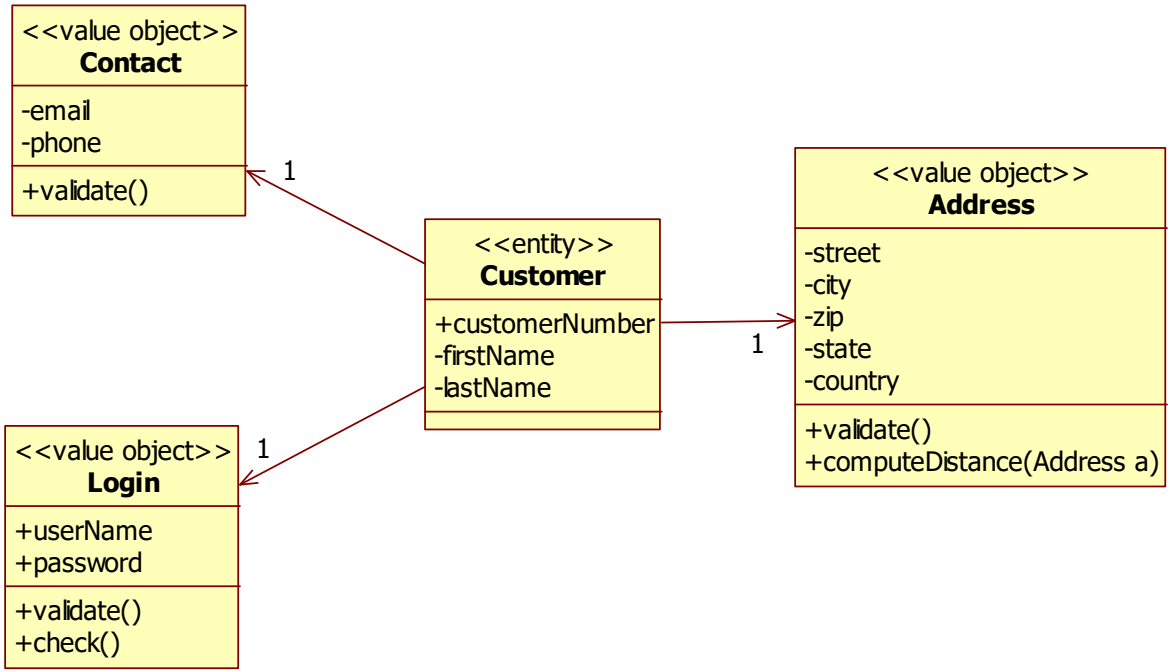


2. Enhancing explicitness



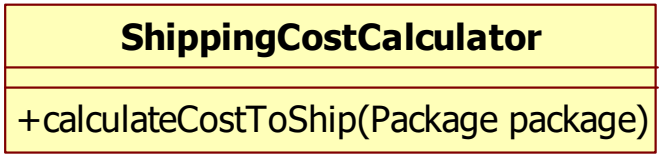
Pushing behavior into value objects





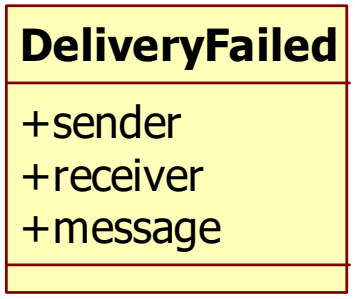
Domain service

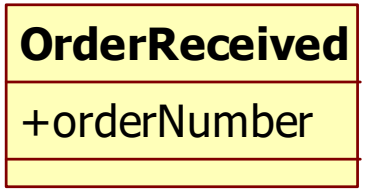
- Sometimes behavior does not belong to an entity or value object
  - But it is still an important domain concept
- Use a domain service.



Domain event

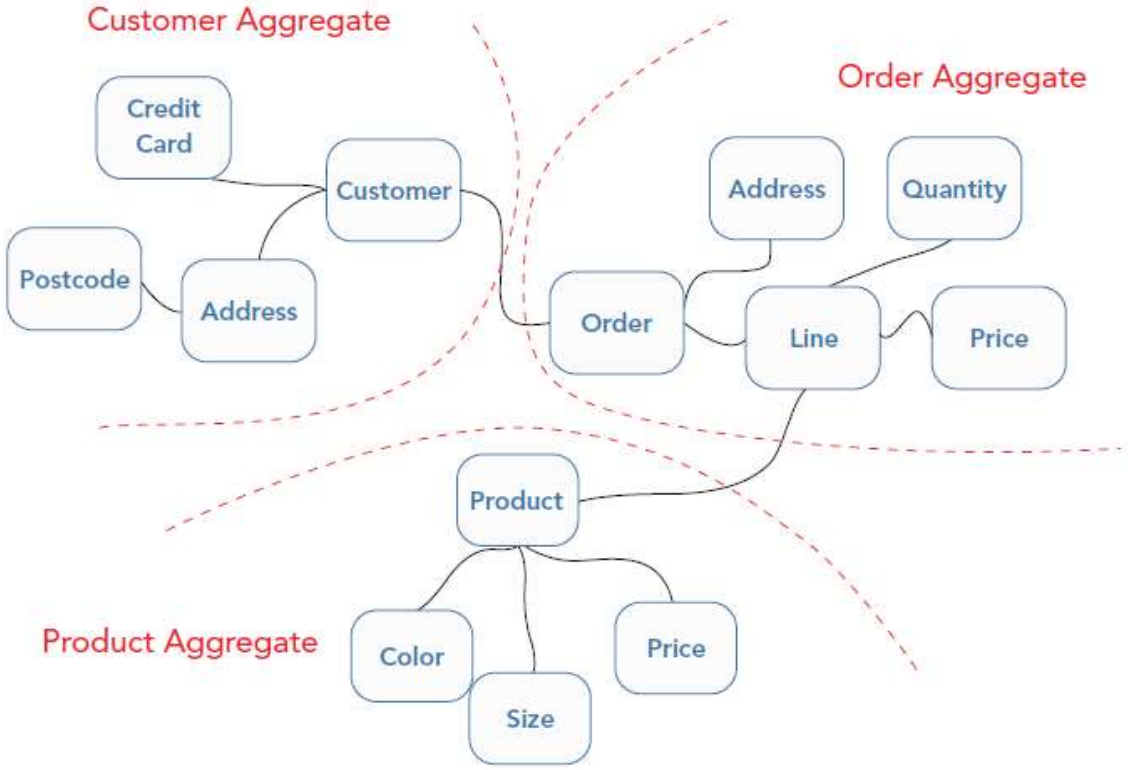
- Classes that represent important events in the problem domain that have already happened
  - Immutable





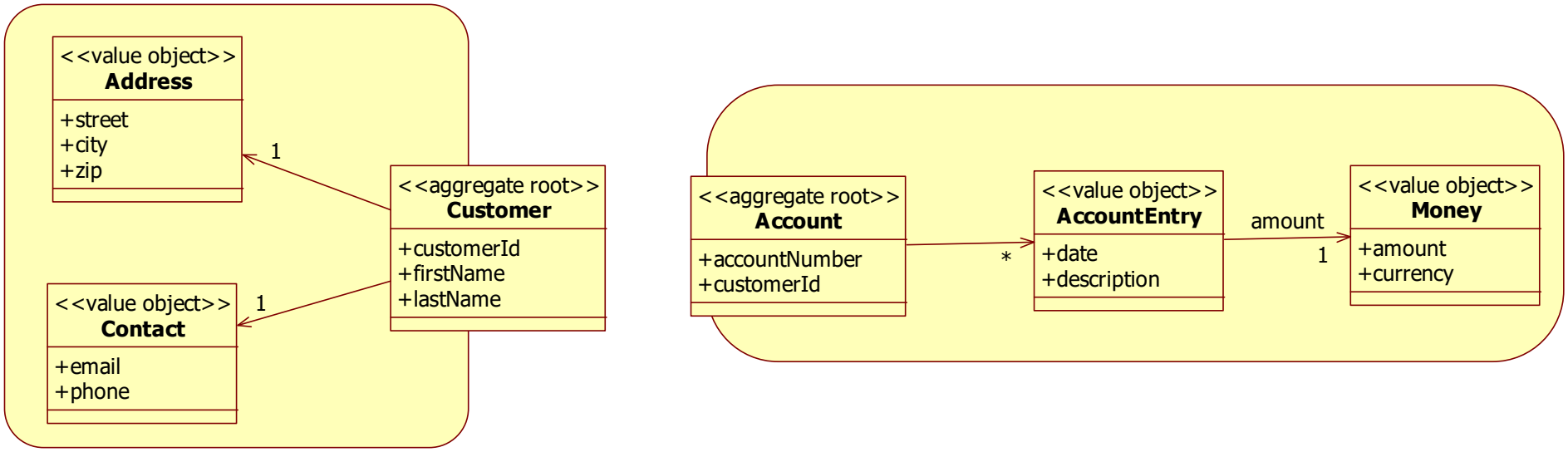
Aggregates

- Large models are split and grouped into aggregates of entities and value objects that are treated as a conceptual whole.



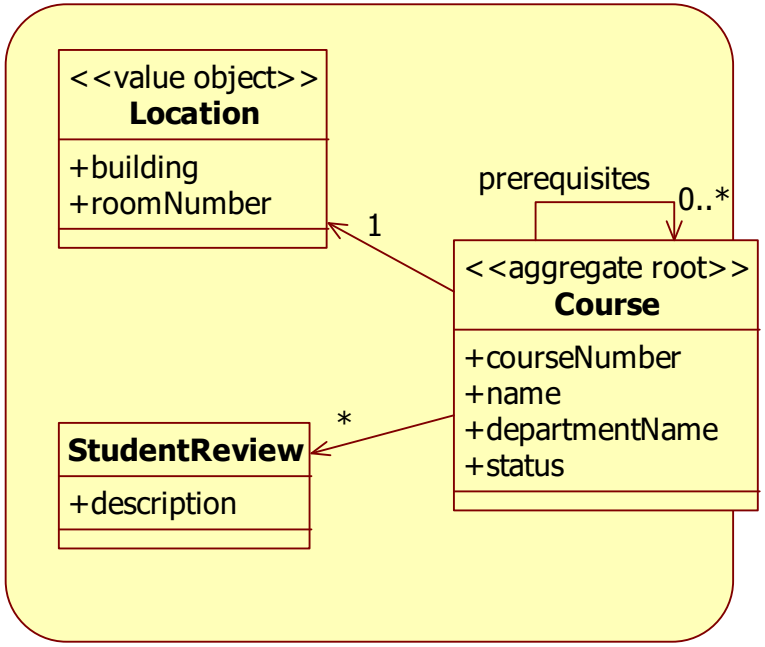
Referencing other aggregates

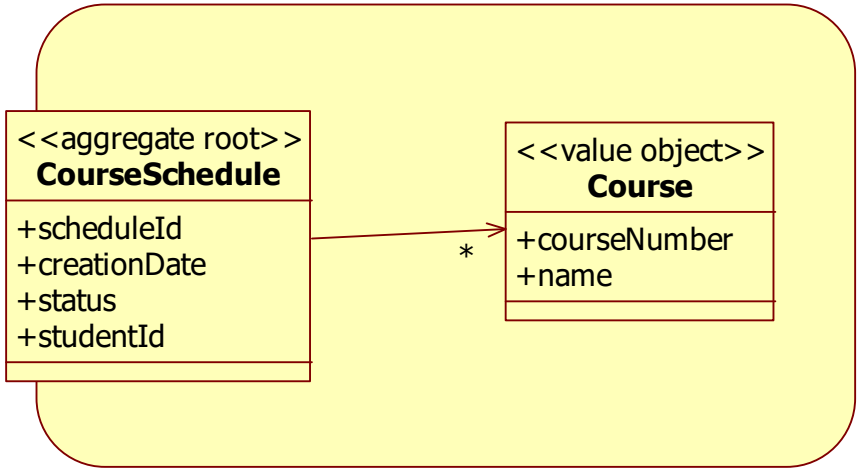
- Using an Id



Referencing other aggregates

- Add a new class





RestTemplate

```
@Autowired
private RestOperations  restTemplate;


@Bean
RestTemplate restTemplate() {
  RestTemplate restTemplate = new RestTemplate();
  restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());
  restTemplate.getMessageConverters().add(new StringHttpMessageConverter());
  return restTemplate;
}
```

LESSON 5

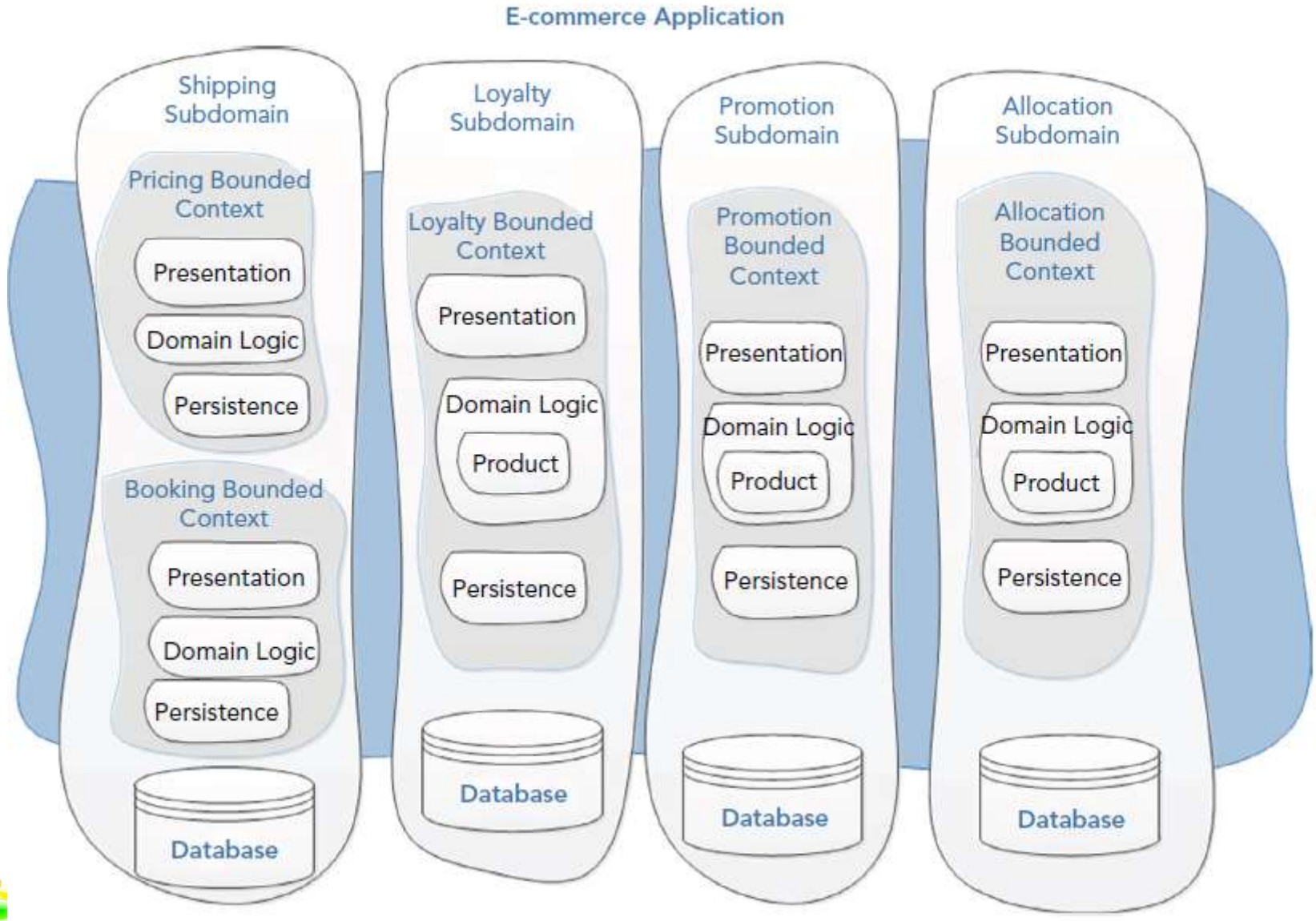
Lesson 5 DDD/components topics

- Ubiquitous language
- Rich domain model
- Bounded context
- Components
- Interface design
- DTO’s
- Adapter
- Proxy
- Spring Events

Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
  - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
  - Rich domain model
- Let the software be a reflection of the real world domain
- Create small contexts in which a domain model is valid
  - Bounded context

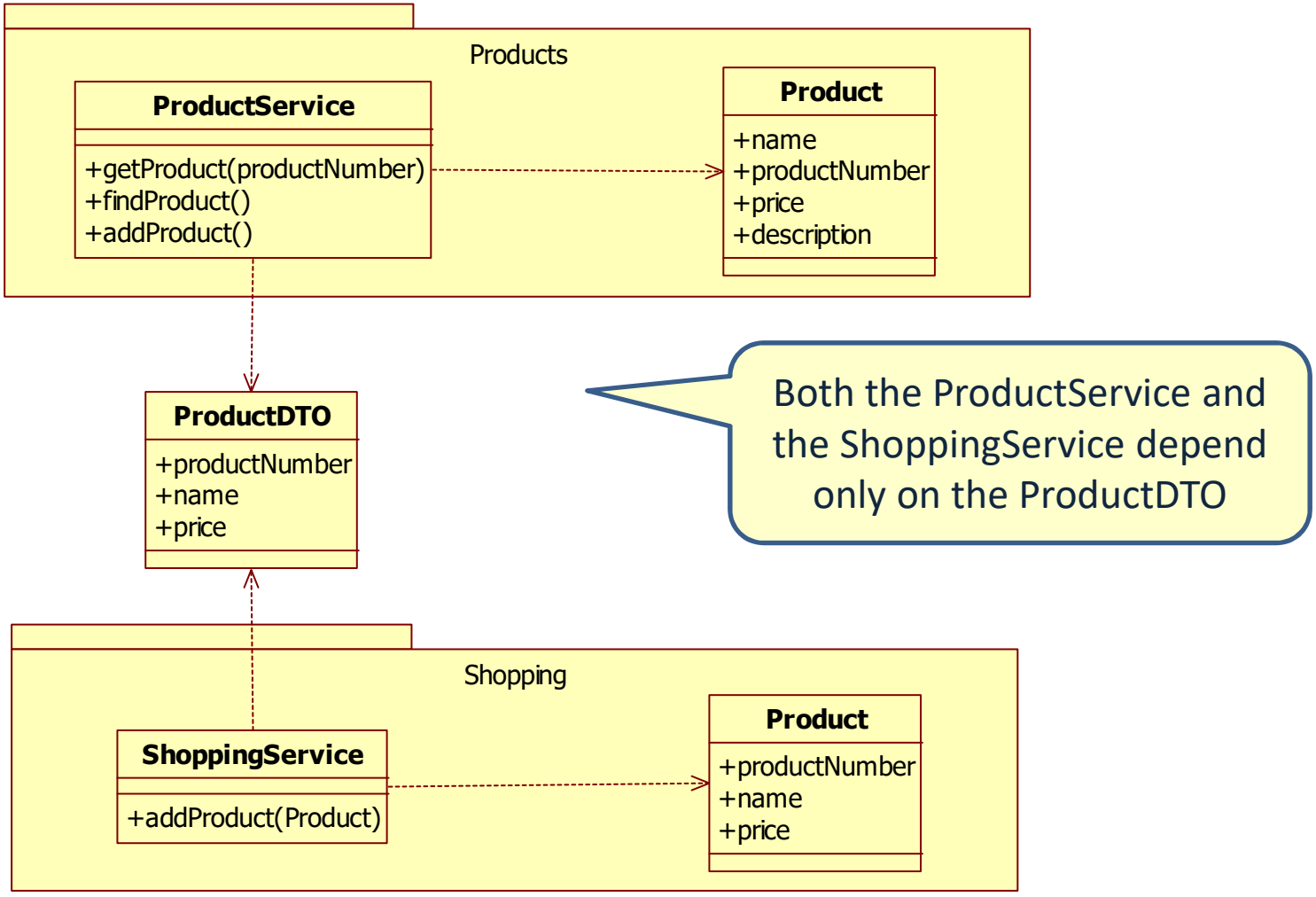
Bounded context example



Interface design

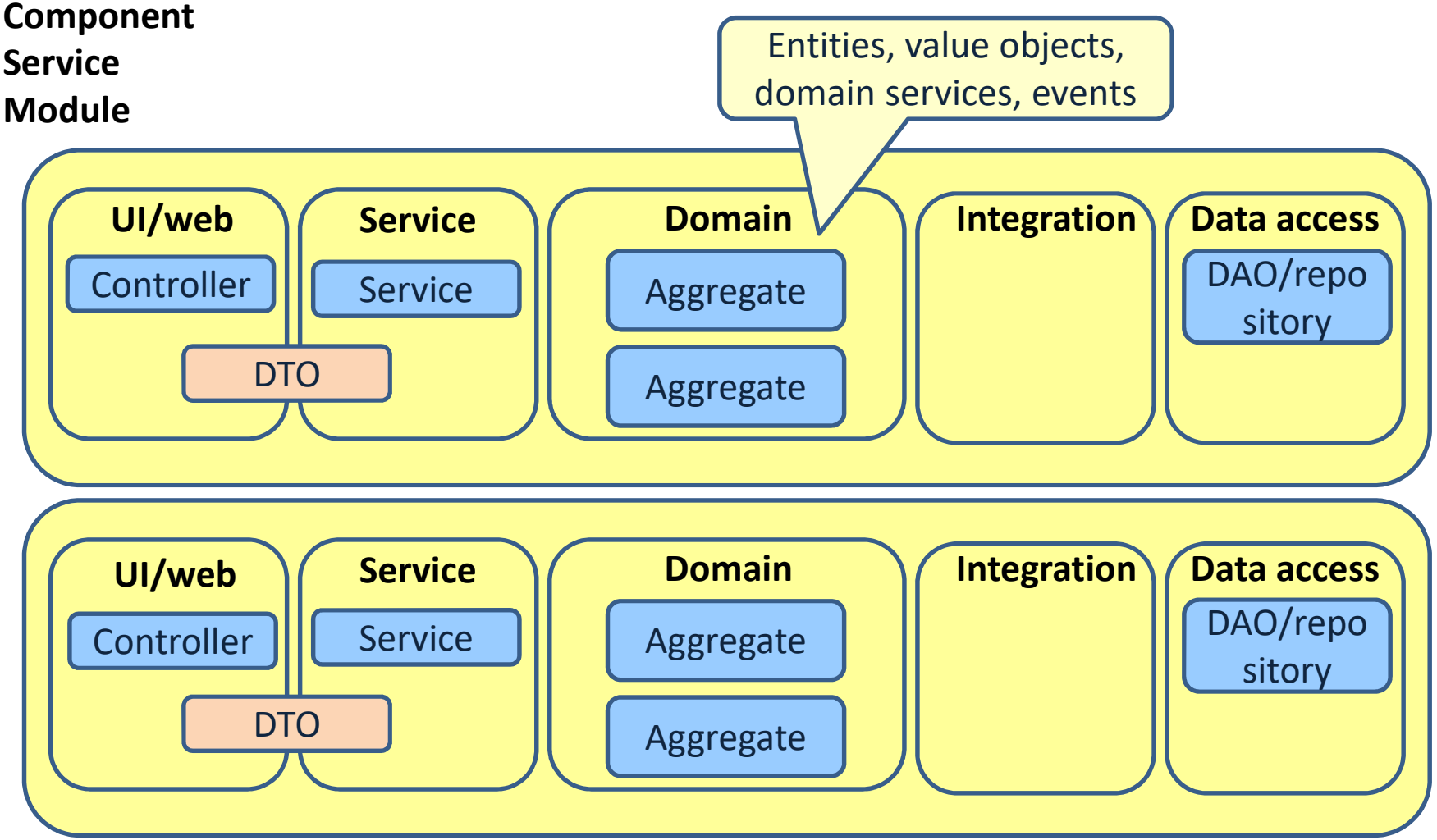
- Start with the client first
- Single responsibility principle
- Interface segregation principle
- Easy to use
- Easy to learn

Data Transfer Objects (DTO)



How does it all fits together?

- **Bounded context**
- **Component**
- **Service**
- **Module**



Spring Events

```
public class AddCustomerEvent {
  private String message;

  public AddCustomerEvent(String message) {
    this.message = message;
  }

  public String getMessage() {
    return message;
  }
}
```

A simple event class

Immutable

Event publisher and listener

```
@Service
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private ApplicationEventPublisher publisher;

  public void addCustomer() {
    publisher.publishEvent(new AddCustomerEvent("New customer is added"));
  }
}
```

Inject a publisher

```
@Service
public class Listener {

  @EventListener
  public void onEvent(AddCustomerEvent event) {
    System.out.println("received event :" + event.getMessage());;
  }
}
```

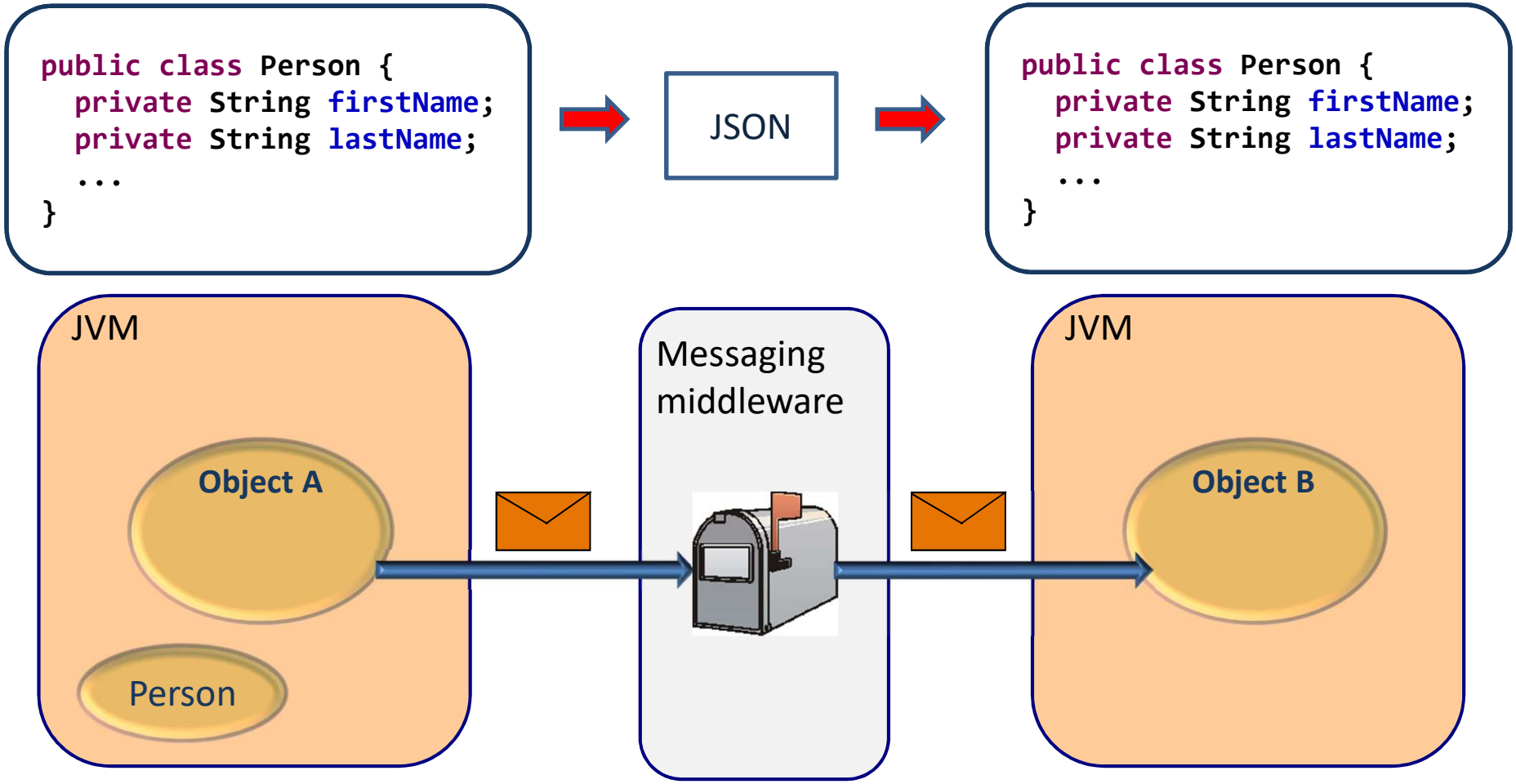
Listen to AddCustomer events

LESSON 6

Lesson 6 Integration patterns topics

- JMS
- Hub and spoke
- ESB
- Integration patterns

Sending an object



Spring JMS sender

```
@Component
public class JmsSender {
  @Autowired
  JmsTemplate jmsTemplate;

  public void sendJMSMessage(Person person) {
    System.out.println("Sending a JMS message.");
    jmsTemplate.convertAndSend("testQueue",person);
  }
}
```

Name of the queue

application.properties

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

Spring JMS receiver

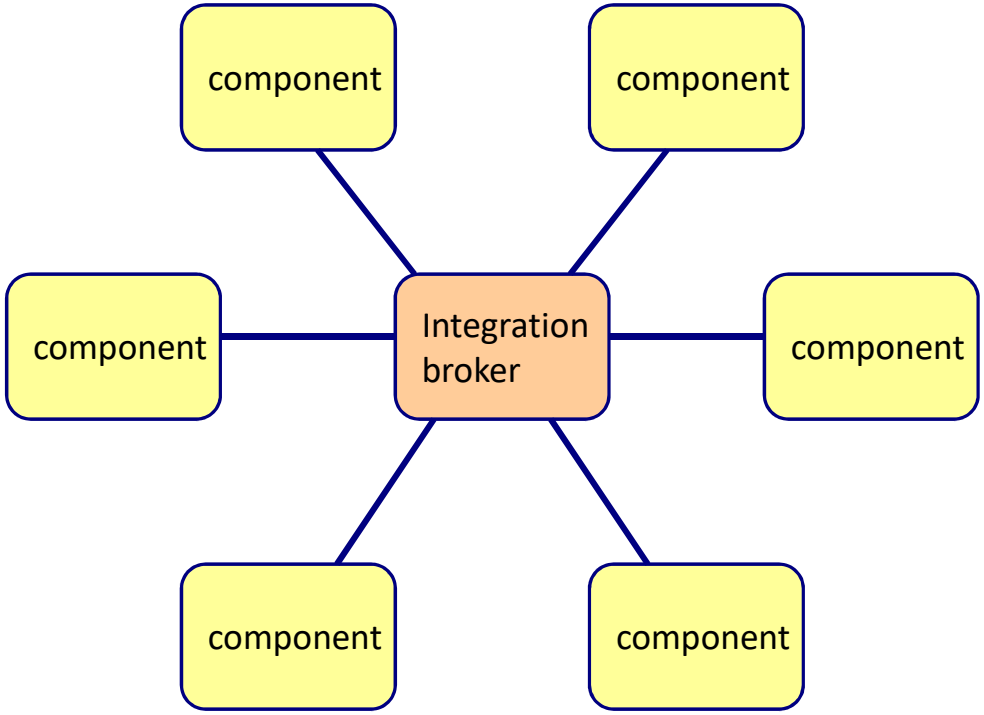
```
@Component
public class PersonMessageListener {

    @JmsListener(destination = "testQueue")
    public void receiveMessage(final Person person) {
      System.out.println("JMS receiver received message:" + person.getFirstName()+"
       "+person.getLastName());
    }
}
```

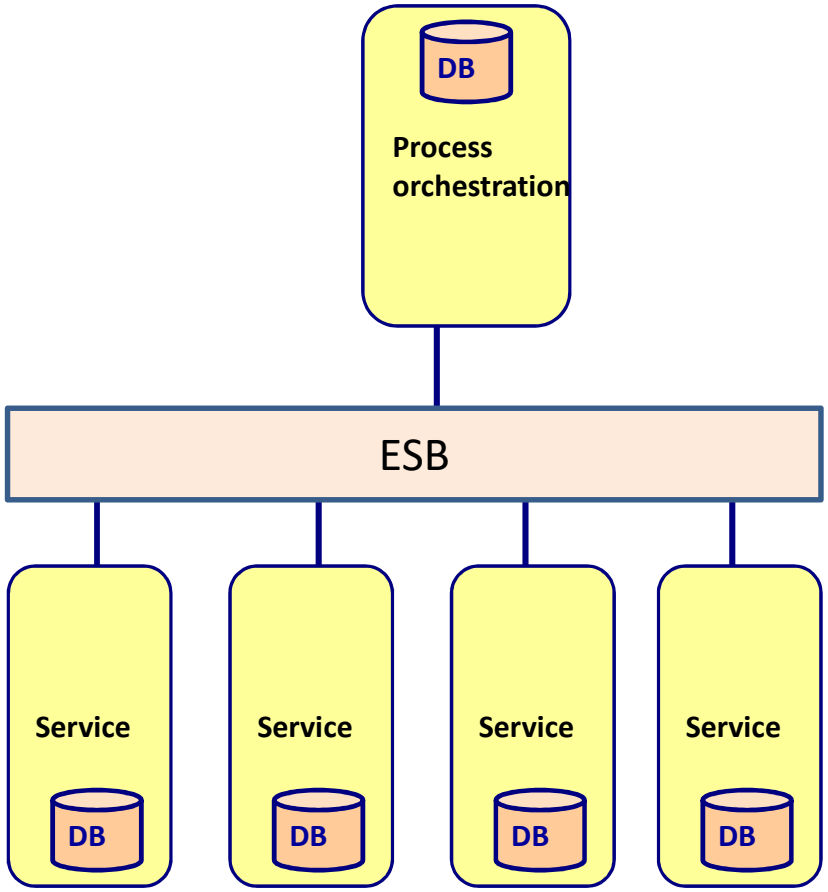
Name of the queue

Hub and Spoke

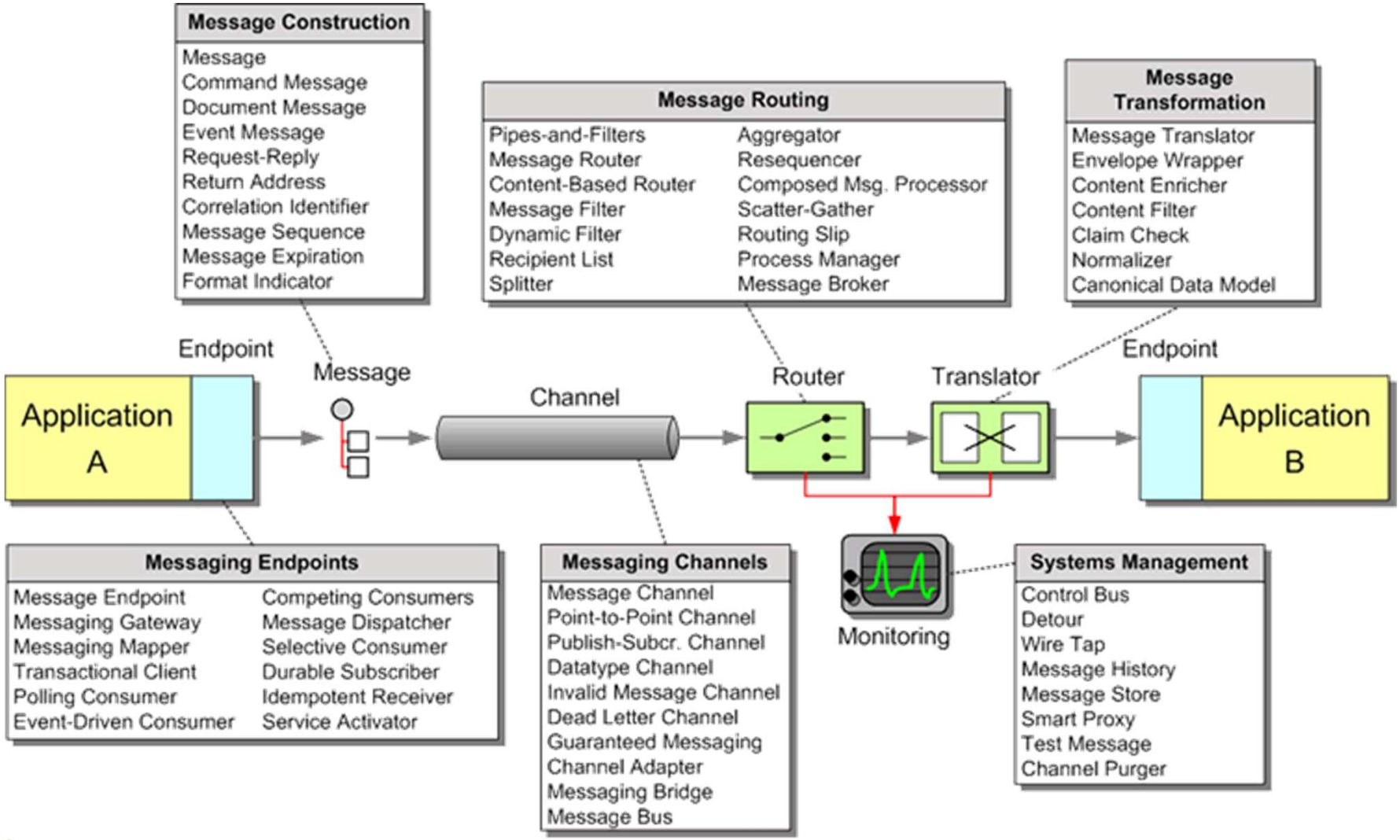
- Integration broker



Service Oriented Architecture



Enterprise Integration Patterns



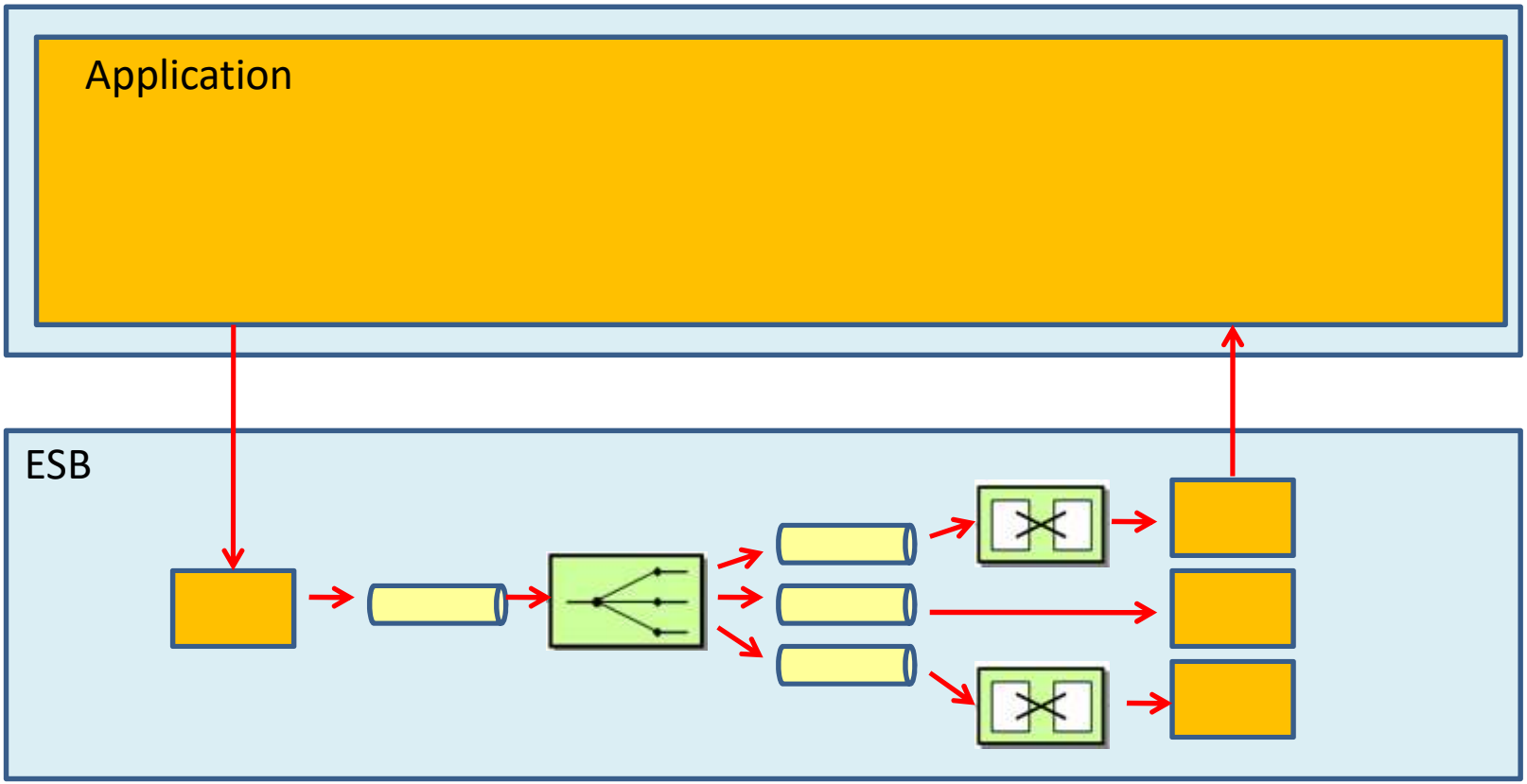
LESSON 7

Lesson 7 Spring Integration topics

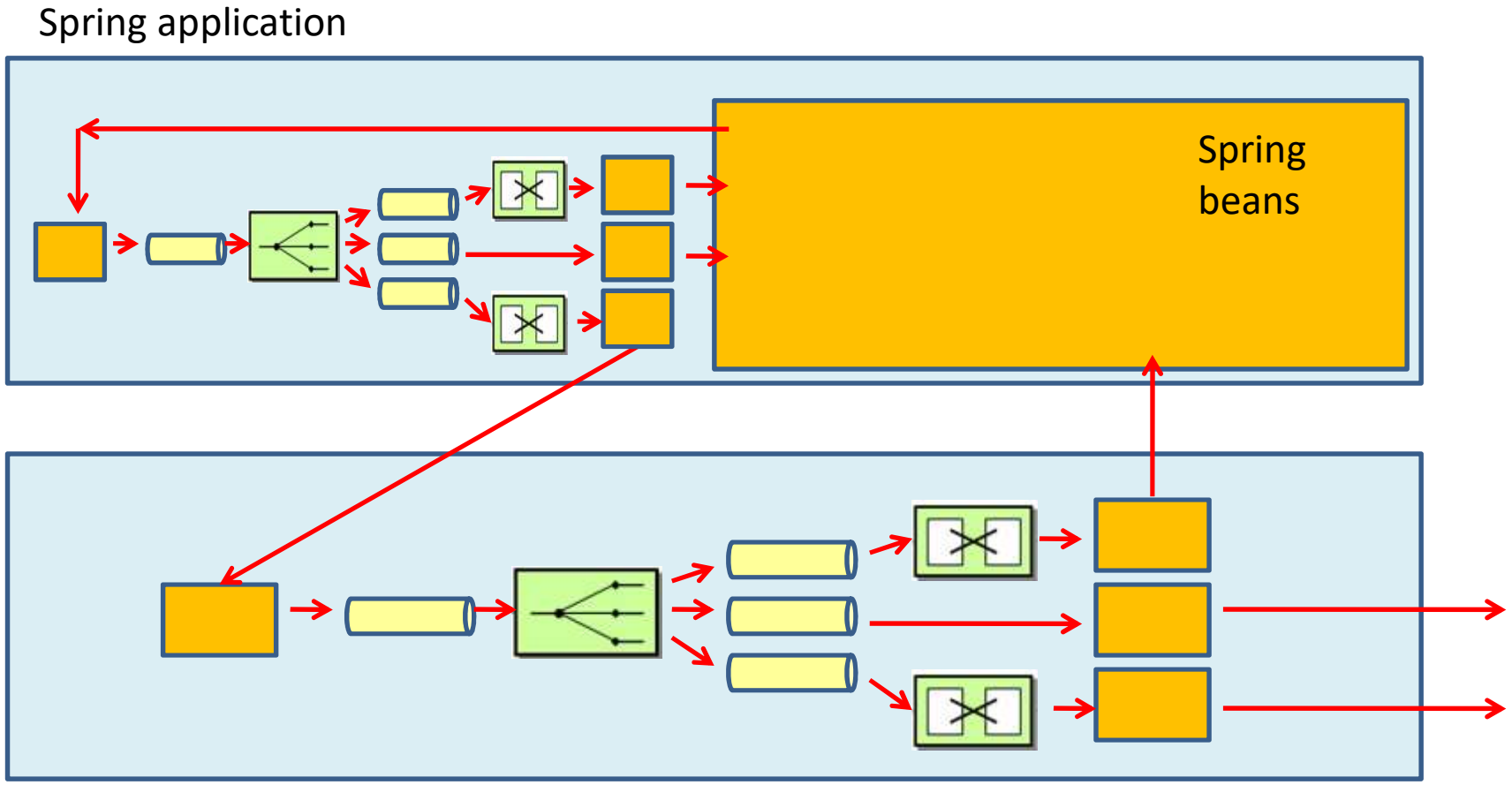
- ESB versus Integration Framework
- Integration patterns in spring integration
  - Service activator
  - Gateway
  - Channels
    - Point-to-point vs. Publish-subscribe
    - Synchronous vs asynchronous
  - Custom router
  - Filter
  - Transformer

ESB

- Runs outside the application
  - Needs to be installed, started, stopped, monitored.



Using Spring Integration



- Use SI inside and outside your application

What you should know

- Draw
  - All integration patterns
- Implement
  - POJO classes for
    - Controller
    - Gateway
    - Custom router
    - Splitter
    - Aggregator
    - Filter
    - Transformer