

CS544

LESSON 2

SPRING BOOT AND AOP

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
April 3 Lesson 1 Introduction Spring framework Dependency injection	April 4 Lesson 2 Spring Boot AOP	April 5 Lesson 3 JDBC JPA	April 6 Lesson 4 JPA mapping 1	April 7 Lesson 5 JPA mapping 2	April 8 Lesson 6 JPA queries	April 9
April 10 Lesson 7 Transactions	April 11 Lesson 8 MongoDB	April 12 Midterm Review	April 13 Midterm exam	April 14 Lesson 9 REST webservices	April 15 Lesson 10 SOAP webservices	April 16
April 17 Lesson 11 Messaging	April 18 Lesson 12 Scheduling Events Configuration	April 19 Lesson 13 Monitoring	April 20 Lesson 14 Testing your application	April 21 Final review/Project	April 22 Project	April 23
April 24 Final exam	April 25 Project	April 26 Project	April 27 Class celebration			

SPRING BOOT

Spring boot

- Framework that makes it easy to configure and run spring applications
- Simple maven configuration
- Default/auto spring configuration
- Containerless deployment

Spring boot POM file

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M6</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Inherit Spring Boot default dependencies and versions

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

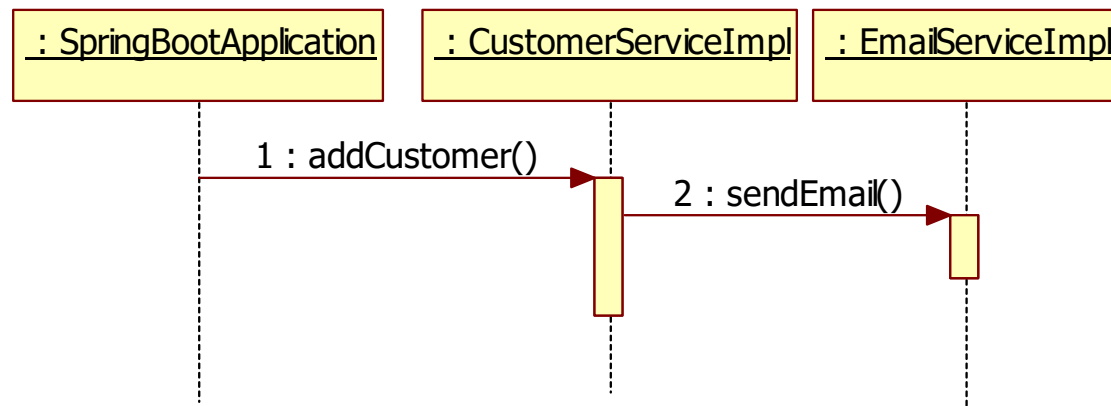
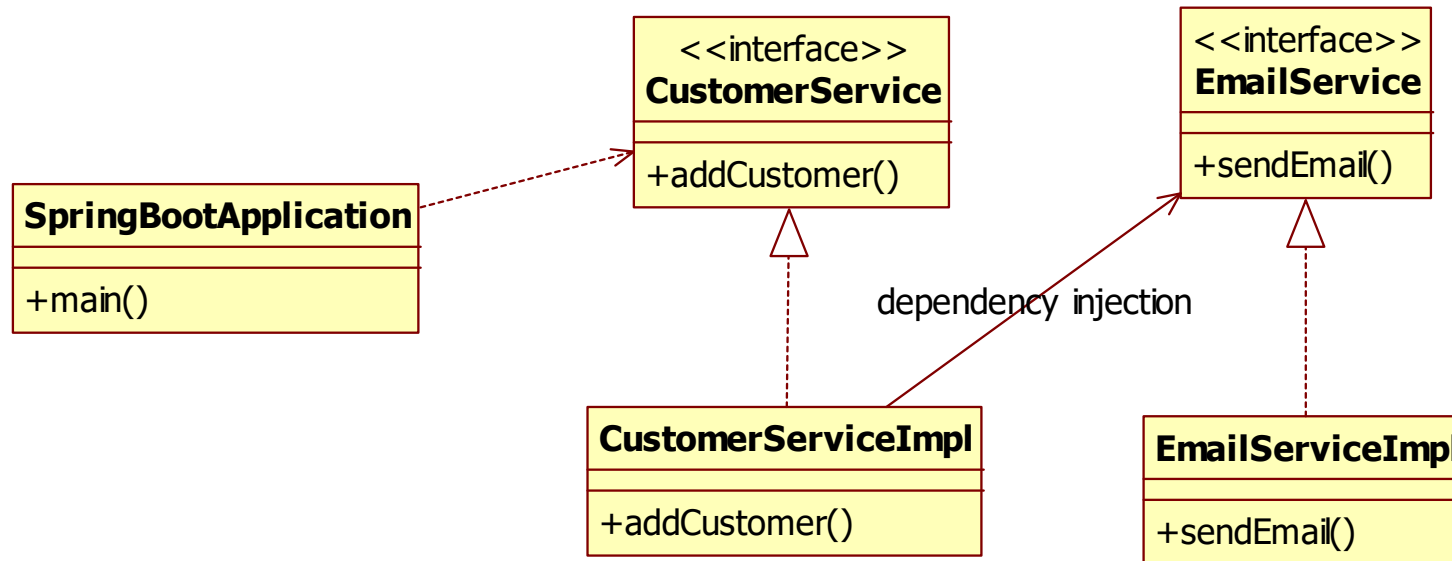
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Starter POM

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Contains goals for packaging the application

Example application



Using annotations

```
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();  
    }  
}
```

Spring Boot option 1

Same as
@Configuration,
@ComponentScan
@EnableAutoConfiguration

```
@SpringBootApplication
public class SpringBootApplication {

    public static void main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(SpringBootApplication.class);
        CustomerService customerService =
            context.getBean("customerServiceImpl", CustomerService.class);
        customerService.addCustomer();
    }
}
```

Create an ApplicationContext
based on the current
configuration class

Spring Boot option 2

```
@SpringBootApplication
public class SpringBootApplication {

    public static void main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(AppConfig.class);
        CustomerService customerService =
            context.getBean("customerServiceImpl", CustomerService.class);
        customerService.addCustomer();
    }
}
```

Create an ApplicationContext
based on an external
configuration class

```
@Configuration
@ComponentScan("customers")
public class AppConfig {
}
```

Spring Boot option 3

```
@SpringBootApplication
public class SpringBootProjectApplication implements CommandLineRunner {
    @Autowired
    private CustomerService customerService;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootProjectApplication.class, args);
    }

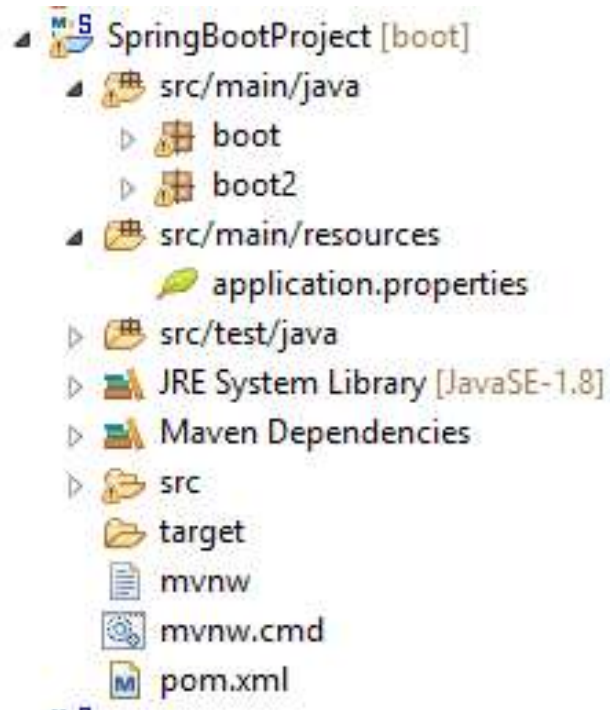
    @Override
    public void run(String... args) throws Exception {
        customerService.addCustomer();
    }
}
```

Implement the CommandLineRunner

Implement the run() method

Spring Boot configuration

- Spring Boot uses **application.properties** as the default configuration file



application.properties

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

```
@Service  
public class EmailServiceImpl implements EmailService{  
    @Value("${smtpserver}")  
    String smtpServer;  
  
    public void sendEmail() {  
        System.out.println("Sending email using smtp server "+smtpServer);  
    }  
}
```

Inject the value from the properties file



application.properties

```
1 smtpserver=smtp.mydomain.com  
2
```

Spring creates the context

```
@SpringBootApplication
public class SpringBootProjectApplication implements CommandLineRunner {
    @Autowired
    private EmailService emailService;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootProjectApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        emailService.sendEmail();
    }
}
```

Spring automatically reads
application.properties

You create the context yourself

```
@SpringBootApplication
@PropertySource("classpath:application.properties")
public class SpringBootProjectApplication {

    public static void main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(SpringBootProjectApplication.class);
        EmailService emailService = context.getBean("emailServiceImpl",
            EmailService.class);
        emailService.sendEmail();
    }
}
```

You need to define the application.properties file yourself

You create the context yourself

Spring component scanning

Spring will automatically scan all classes in the package 'demo' and all sub-packages of 'demo'

```
package demo;
```

```
@SpringBootApplication
```

```
public class SpringBootApplication implements CommandLineRunner {
```

```
    @Autowired
```

```
    private CustomerService customerService;
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(SpringBootApplication.class, args);
```

```
    }
```

```
    @Override
```

```
    public void run(String... args) throws Exception {
```

```
        customerService.addCustomer();
```

```
    }
```

```
}
```

src/main/java

demo

CustomerService.java

SpringBootApplication.java

CustomerService is in the package 'demo'

@ComponentScan

```
package demo;
```

Specify all packages that Spring will scan

```
@SpringBootApplication
@ComponentScan(basePackages = {"service"})
public class SpringBootApplication implements CommandLineRunner {
    @Autowired
    private CustomerService customerService;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        customerService.addCustomer();
    }
}
```

```
src/main/java
├── demo
│   └── SpringBootApplication.java
├── service
│   └── CustomerService.java
```

CustomerService is not in the package 'demo' or subpackage of 'demo'

Set the logging level in application.properties



```
logging.level.root=ERROR  
logging.level.org.springframework=ERROR
```

DI example

```
@Service
public class GreetingService {
    @Autowired
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Spring does not know
which class to inject

```
@Component
public class GreetingOne implements Greeting{

    public String getGreeting() {
        return "Hello World";
    }
}
```

```
public interface Greeting {
    String getGreeting();
}
```

```
@Component
public class GreetingTwo implements Greeting{

    public String getGreeting() {
        return "Hi World";
    }
}
```

DI example

```
@SpringBootApplication
public class DemoProjectApplication implements CommandLineRunner {

    @Autowired
    private GreetingService greetingService;

    public static void main(String[] args) {
        SpringApplication.run(DemoProjectApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(greetingService.getTheGreeting());
    }
}
```

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Field greeting in demo.GreetingService required a single bean, but 2 were found:

- greetingOne: defined in file [C:\springtraining\workspace\DemoProject\target\classes\demo\GreetingOne.class]
- greetingTwo: defined in file [C:\springtraining\workspace\DemoProject\target\classes\demo\GreetingTwo.class]

Solution 1: use qualifier

```
@Service
public class GreetingService {
    @Autowired
    @Qualifier(value="greetingOne")
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Solution 2: use profiles

```
@Service
public class GreetingService {
    @Autowired
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Set the active profile in
application.properties

```
1 spring.profiles.active=One
2 |
```

```
@Component
@Profile("One")
public class GreetingOne implements Greeting{
    public String getGreeting() {
        return "Hello World";
    }
}
```

Define a profile

```
@Component
@Profile("Two")
public class GreetingTwo implements Greeting{
    public String getGreeting() {
        return "Hi World";
    }
}
```

Define a profile

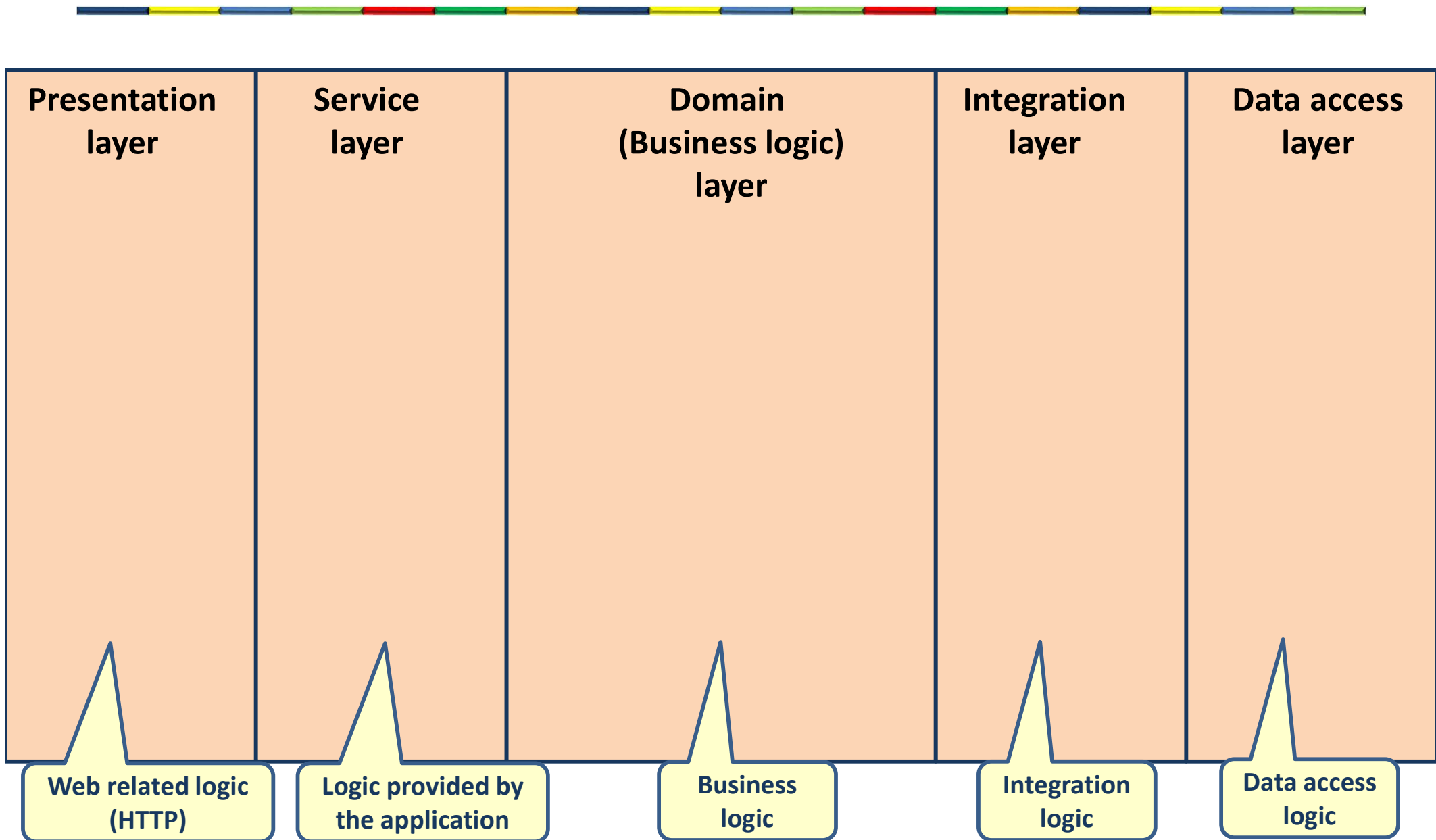
Main point

- Spring boot makes writing enterprise applications simpler by using convention over configuration.

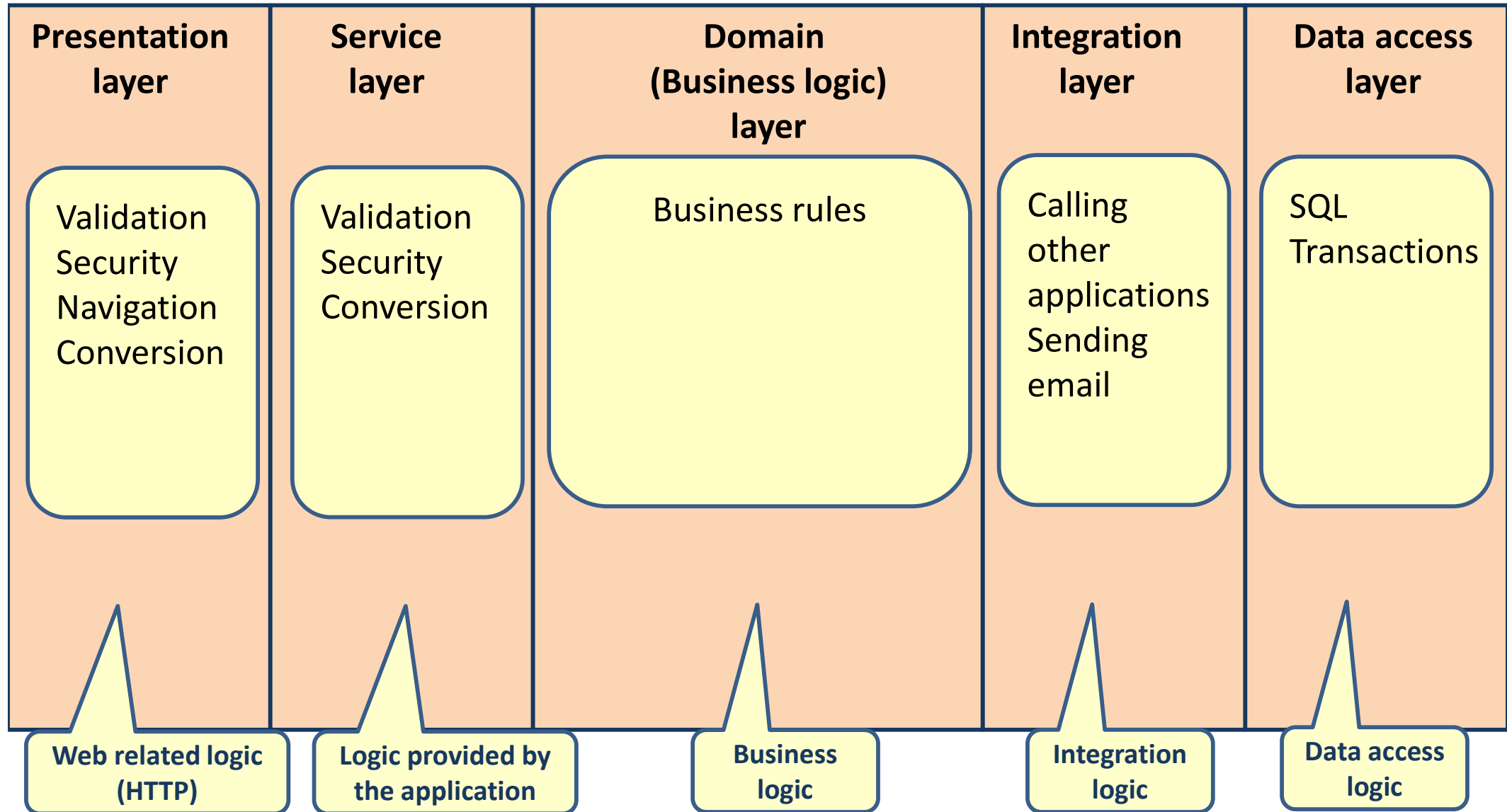
Science of Consciousness: One gains full support of Nature when one operates from the level of the Unified Field, the source of all creation.

LAYERS OF AN ENTERPRISE APPLICATION

Application layers

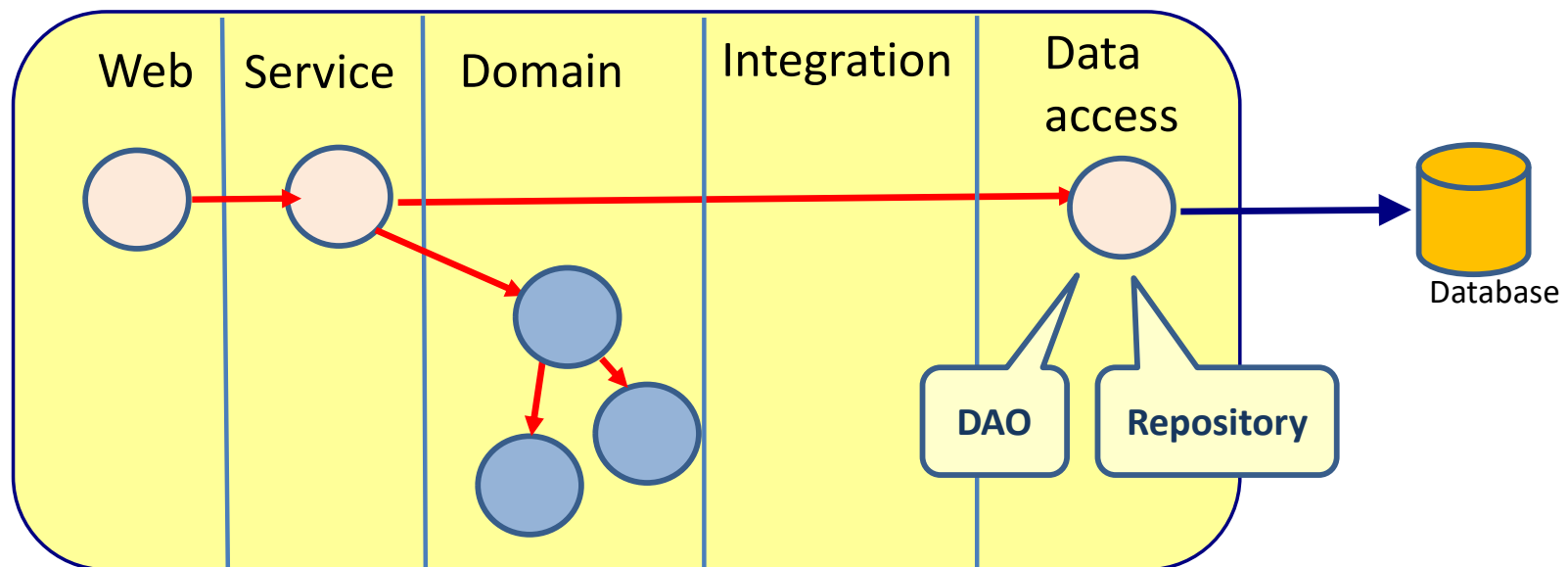


Application layers



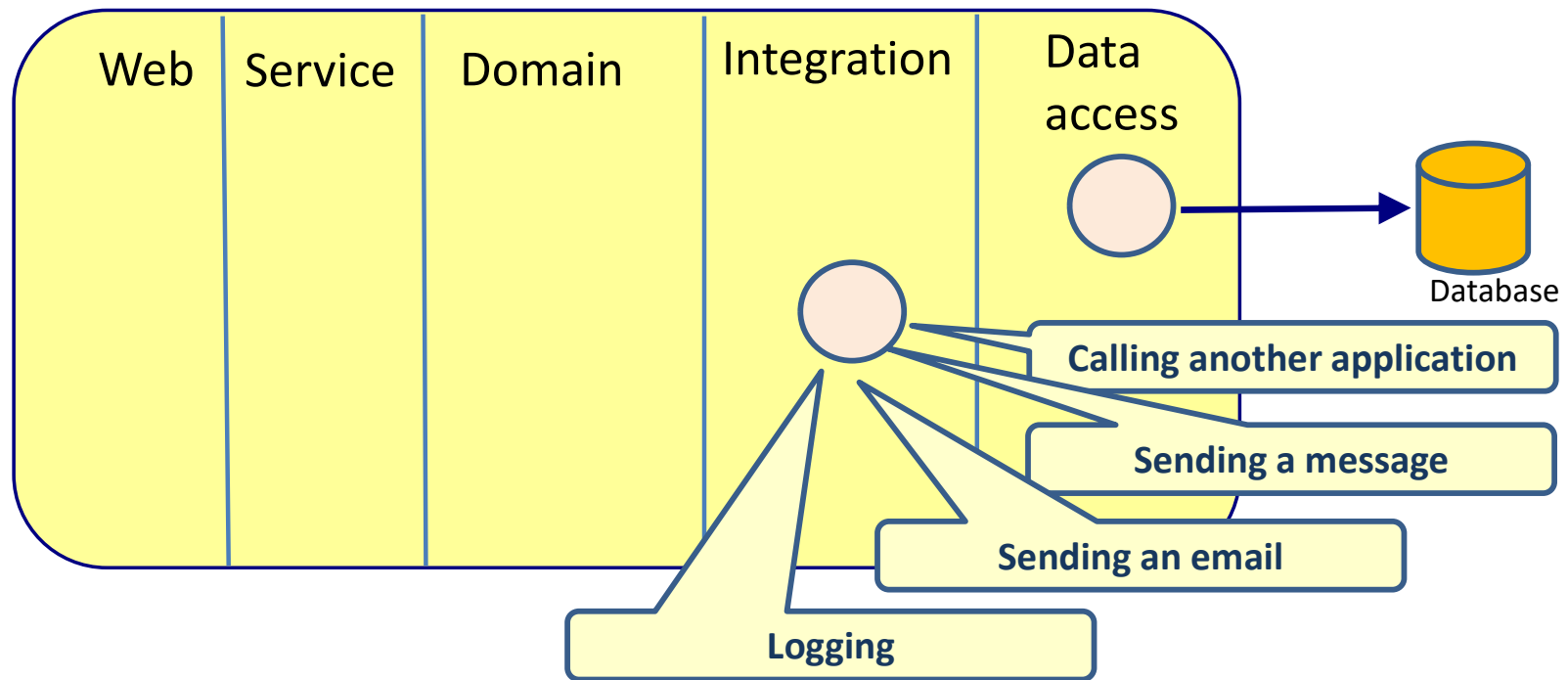
Data Access Object (DAO)/Repository

- Object that knows how to access the database
- Contains all database related logic
- Also called repository



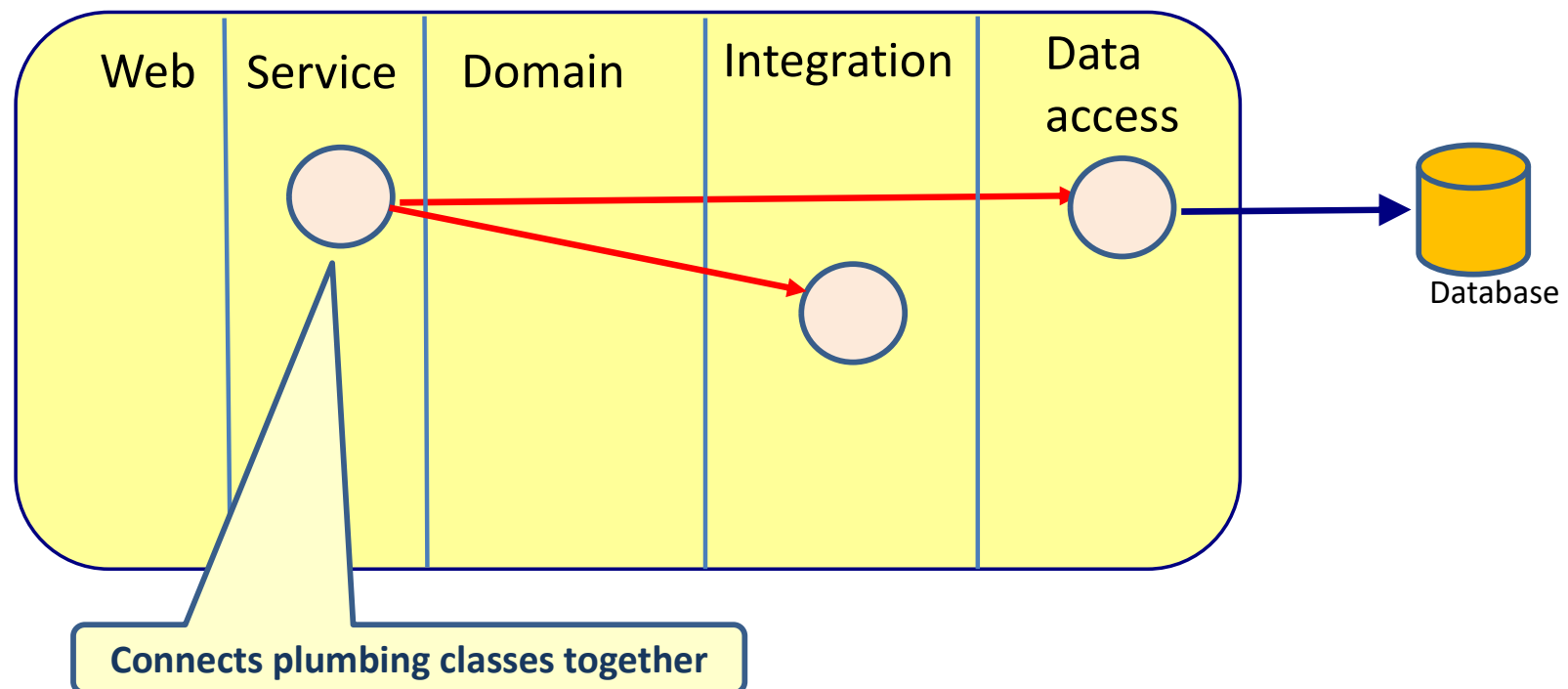
Technical plumbing classes

- Single responsibility



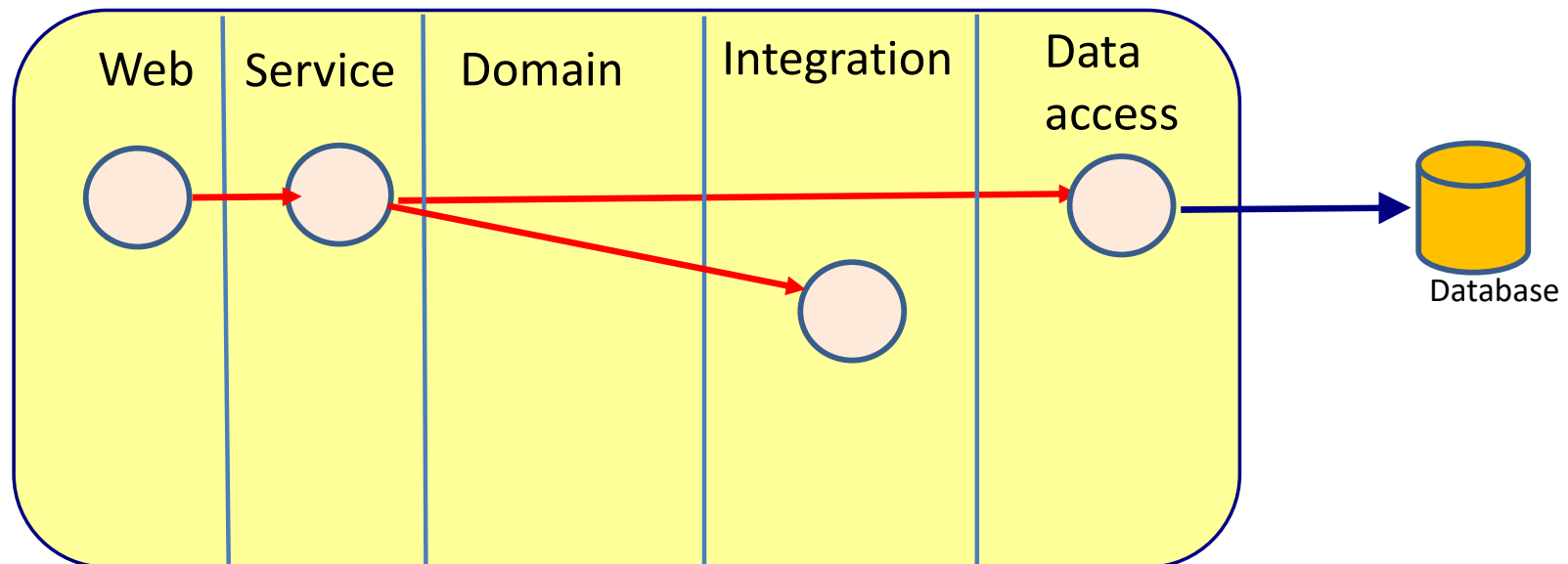
Service classes

- Reception
- Façade



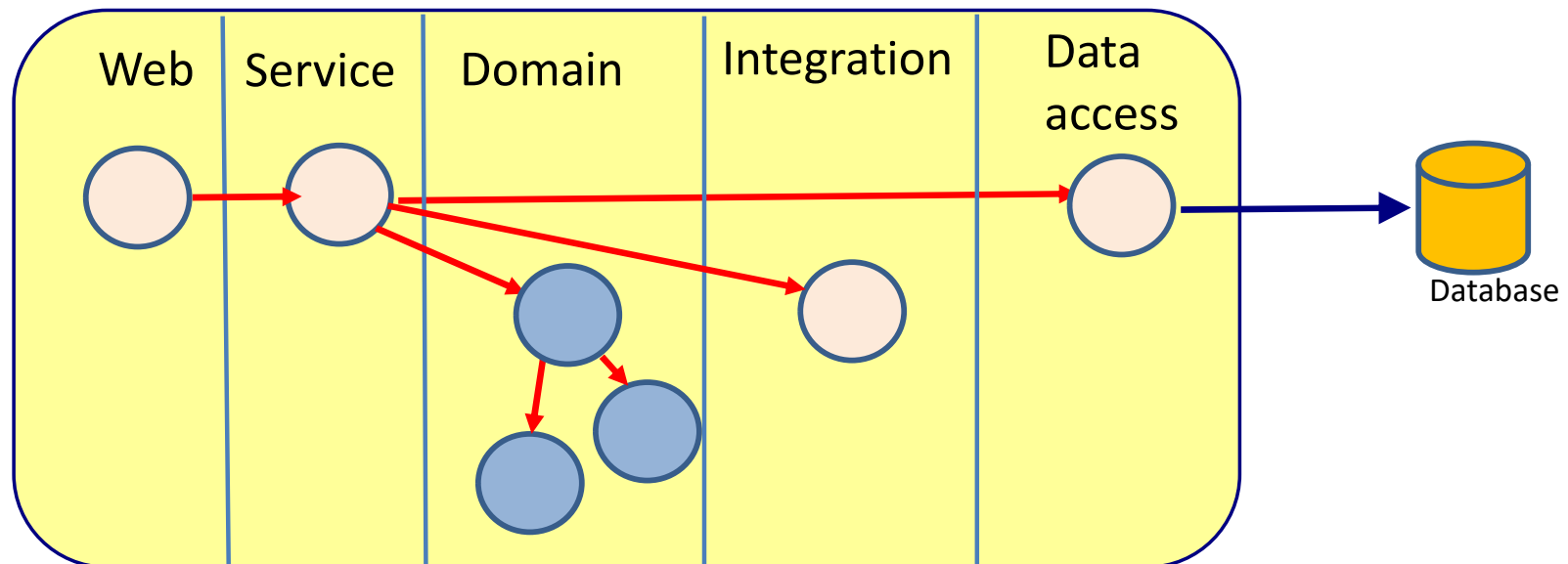
Controller classes

- HTTP controller
 - Receives the HTTP request

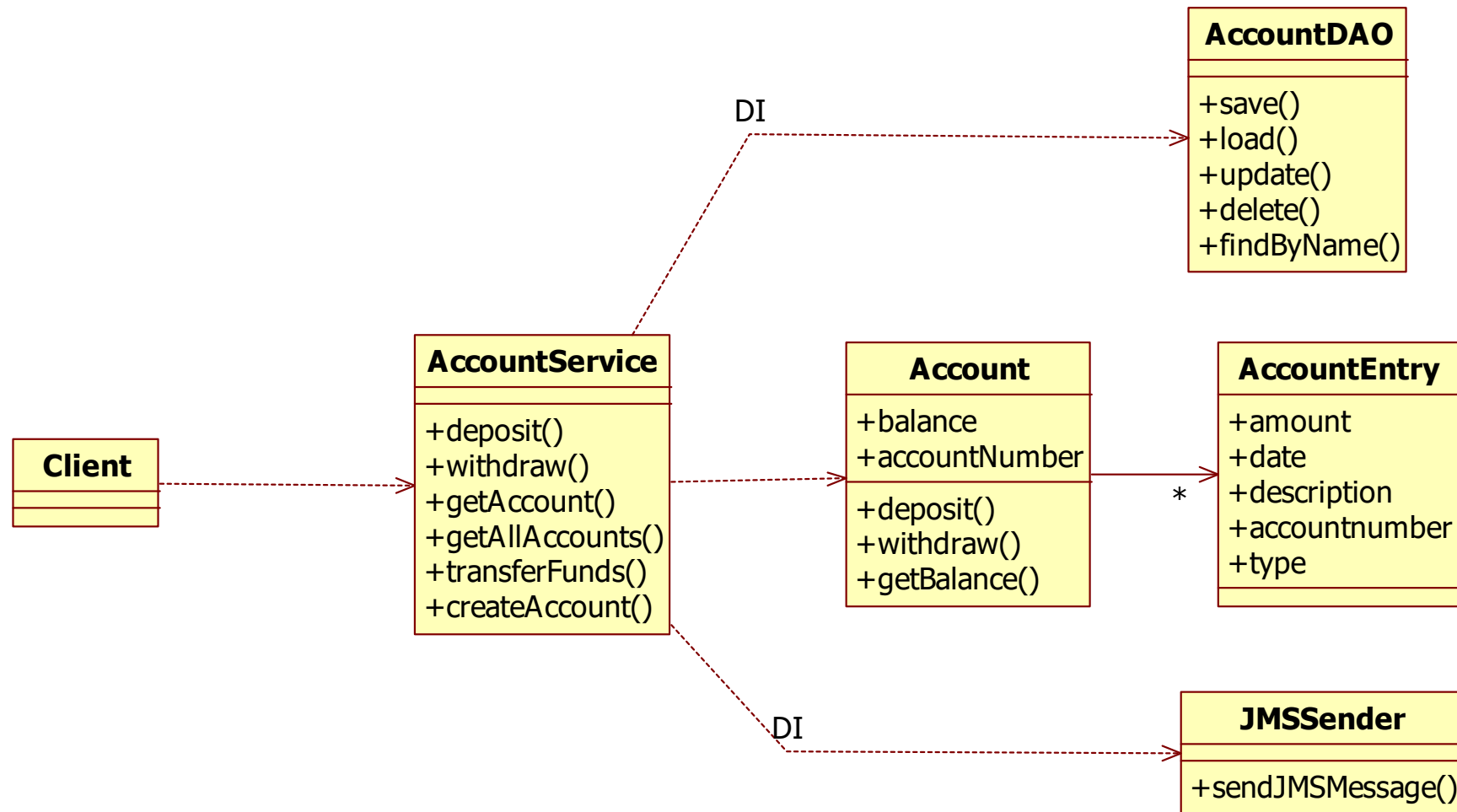


Domain classes

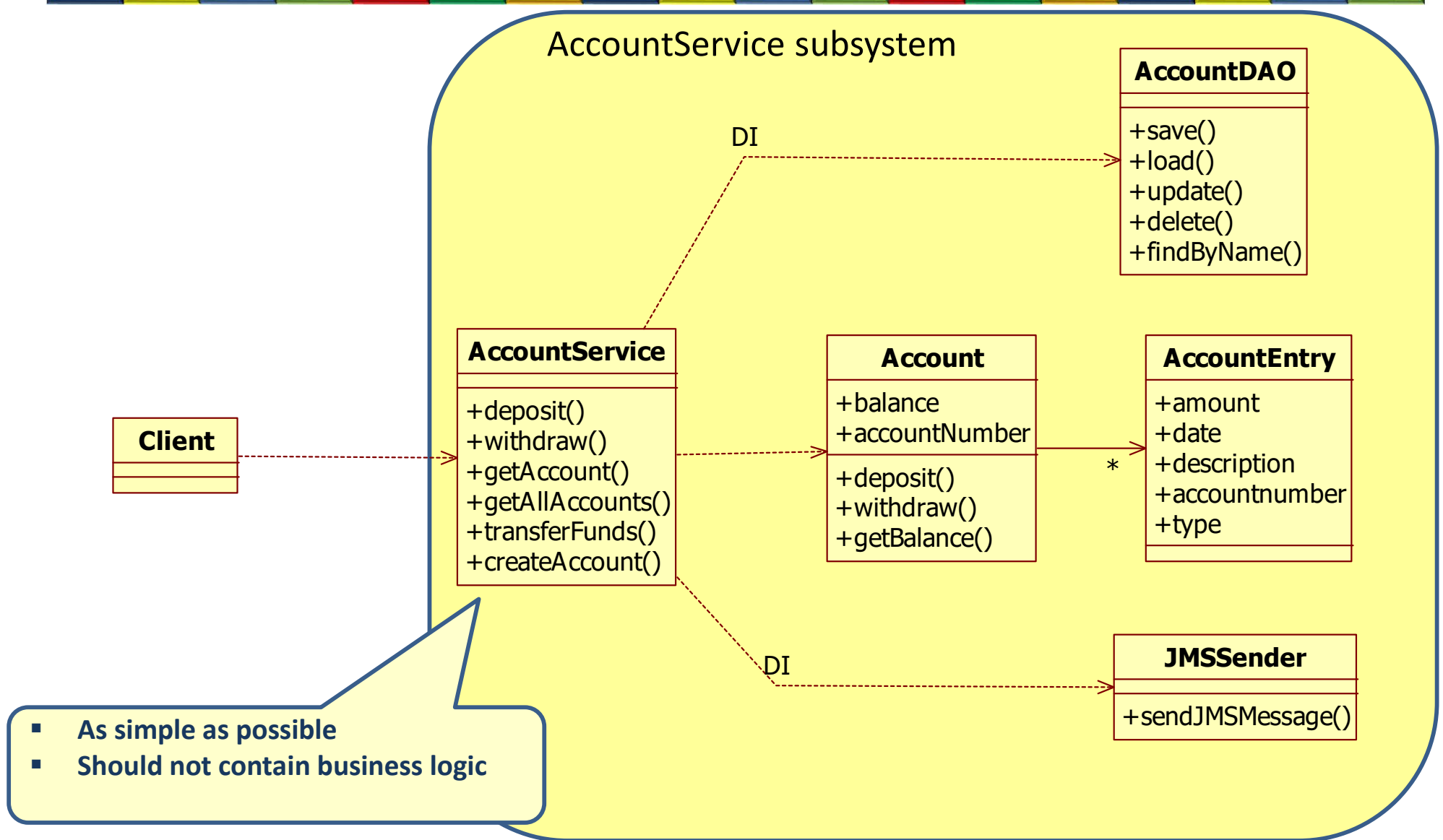
- Implement the business logic
 - Contains no technical code



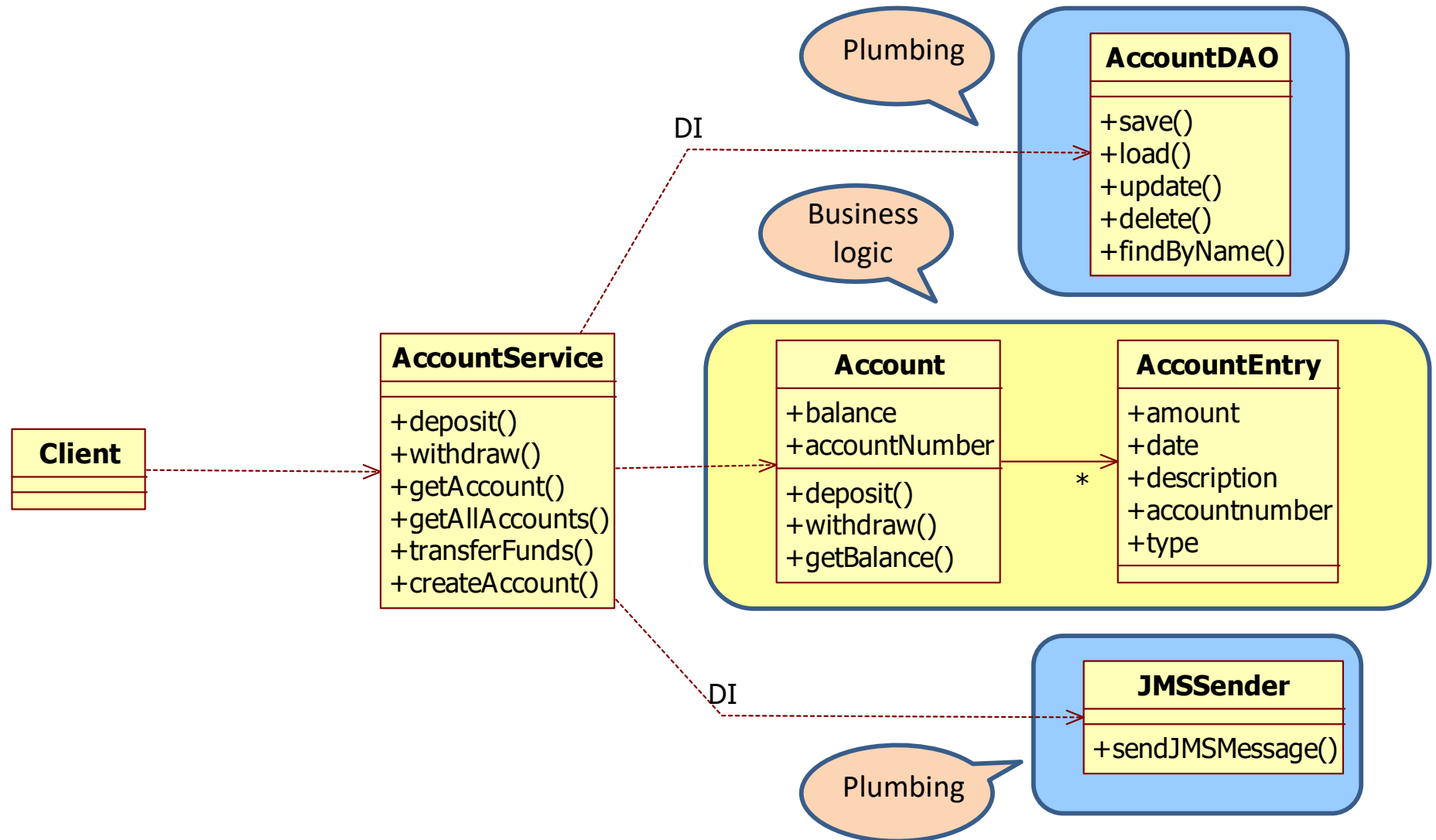
Service Object



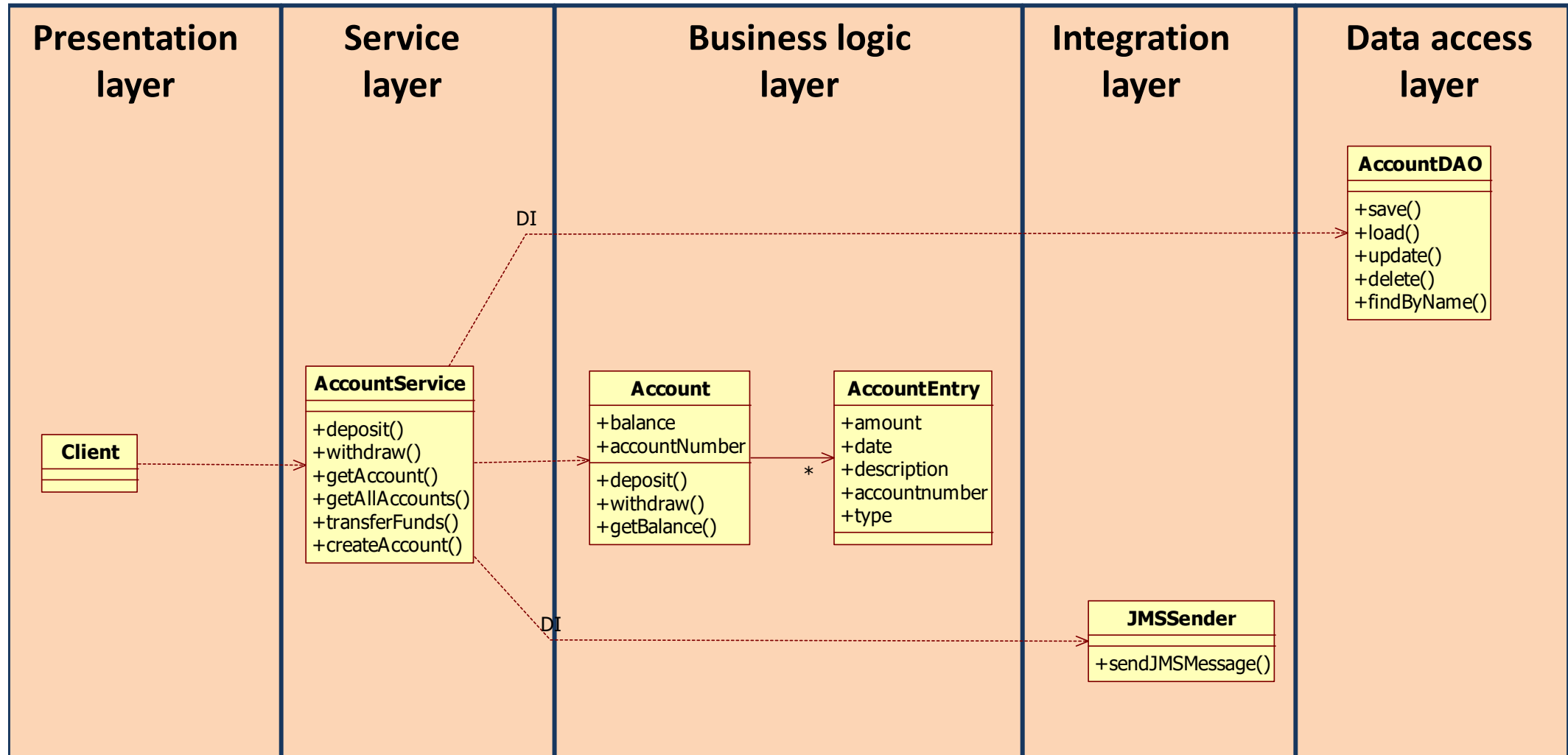
Entry of a complex subsystem



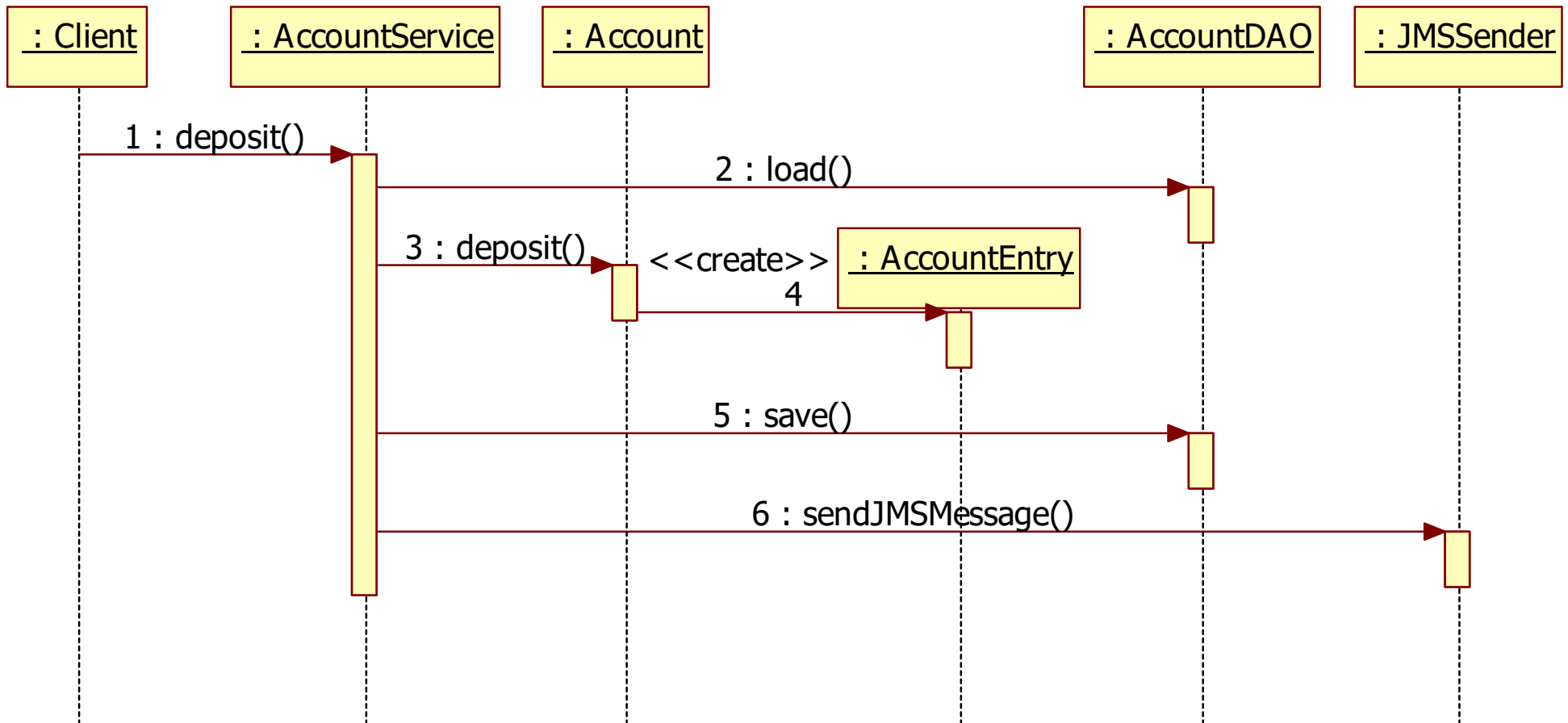
Separation of concern



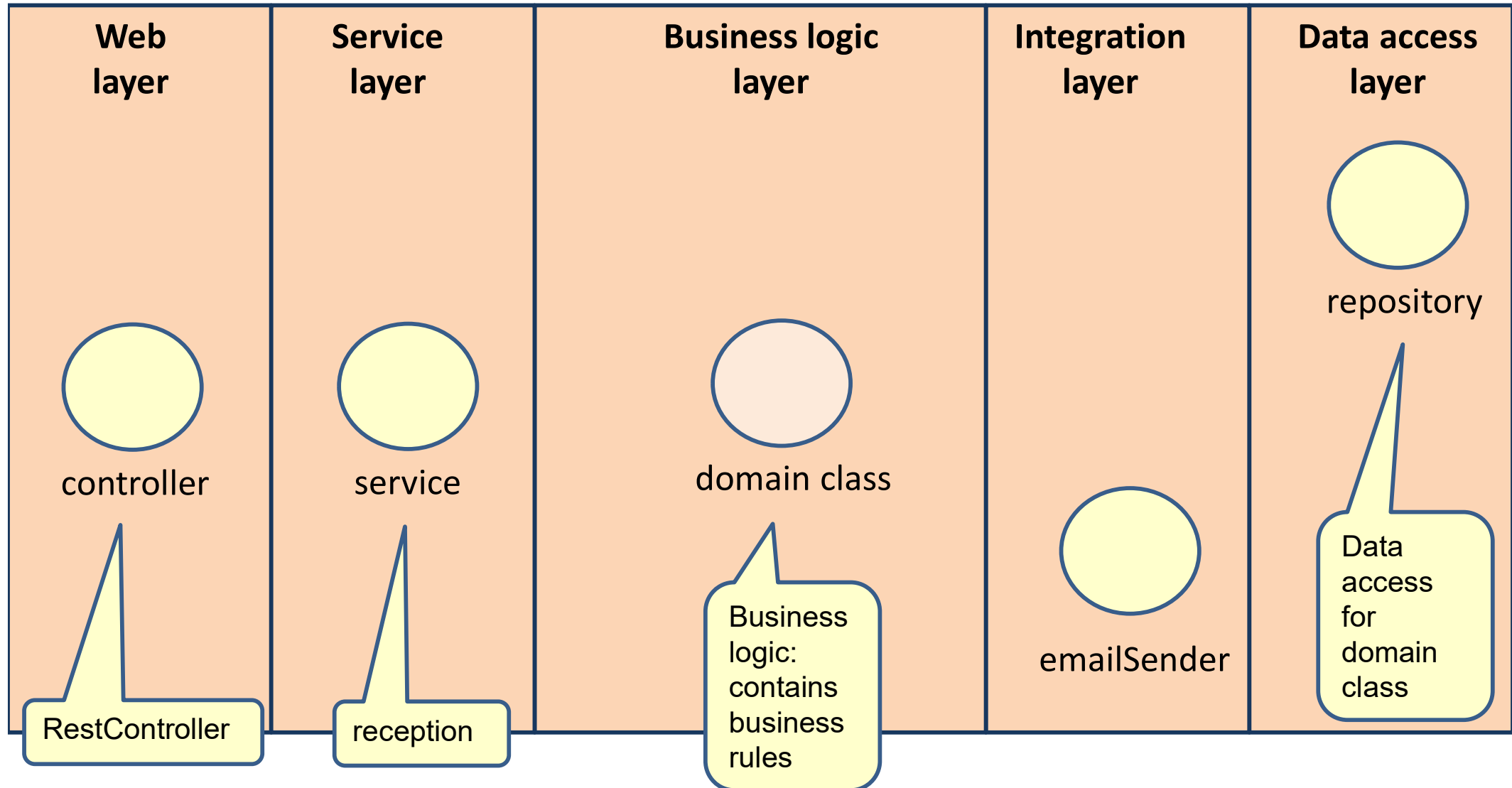
Application layers



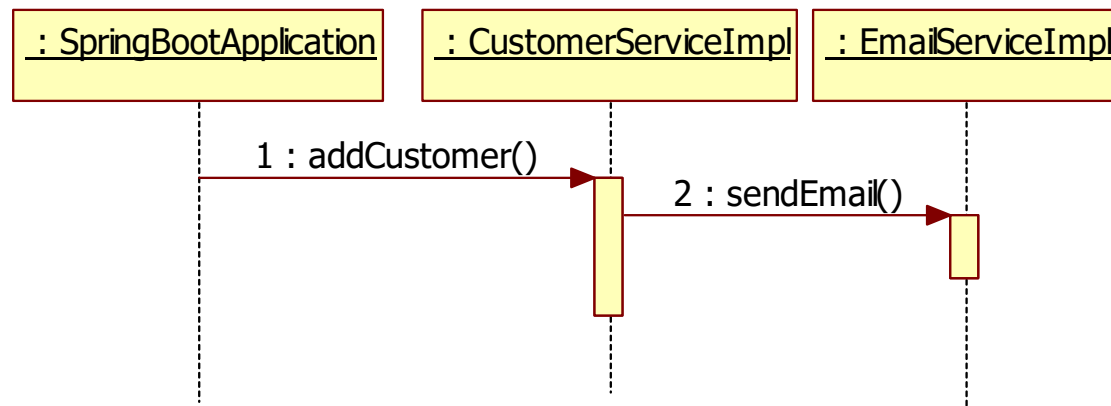
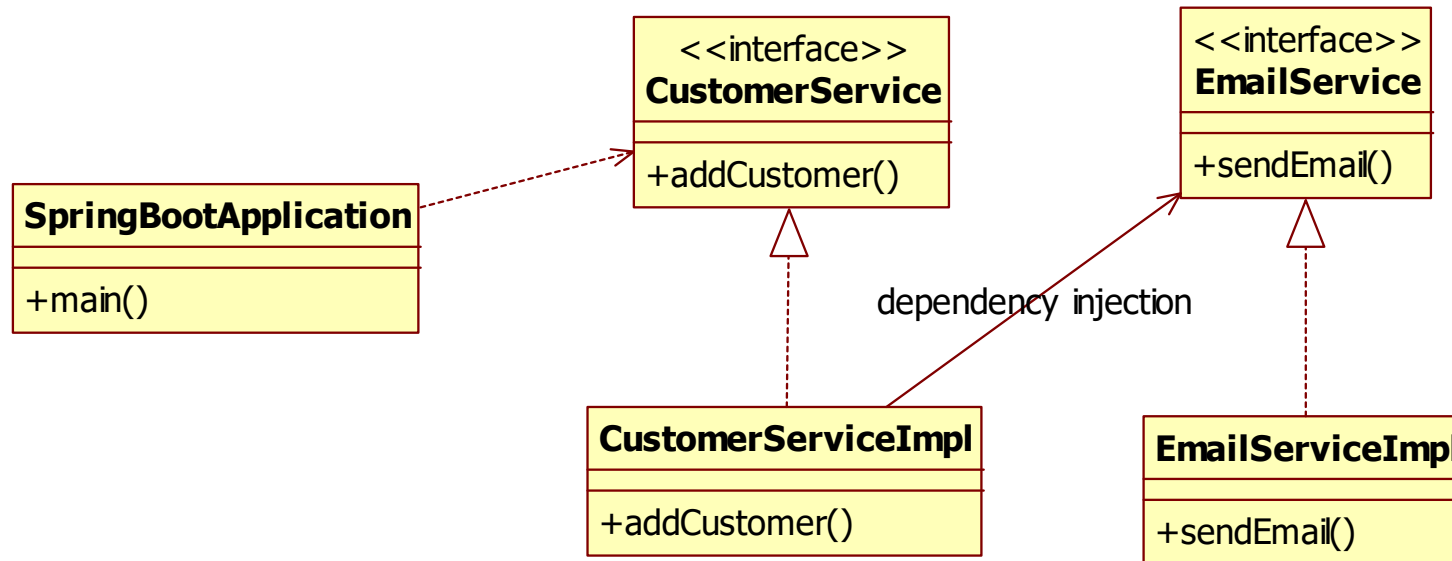
Service object



Layered architecture



Dependency injection




Dependency injection: Setter injection

```
@Service
public class CustomerServiceImpl implements CustomerService {

    private EmailService emailService;

    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

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



Setter injection

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

Dependency injection: Constructor injection

```
@Service
```

```
public class CustomerServiceImpl implements CustomerService {
```

```
    private EmailService emailService;
```

Customer injection

```
    @Autowired
```

```
    public CustomerService(EmailService emailService) {
```

```
        this.emailService = emailService;
```

```
    }
```

```
    public void addCustomer() {
```

```
        emailService.sendEmail();
```

```
    }
```

```
}
```

```
@Service
```

```
public class EmailServiceImpl implements EmailService{
```

```
    public void sendEmail() {
```

```
        System.out.println("Sending email");
```


```
    }
```

```
}
```

Dependency injection: Field injection

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

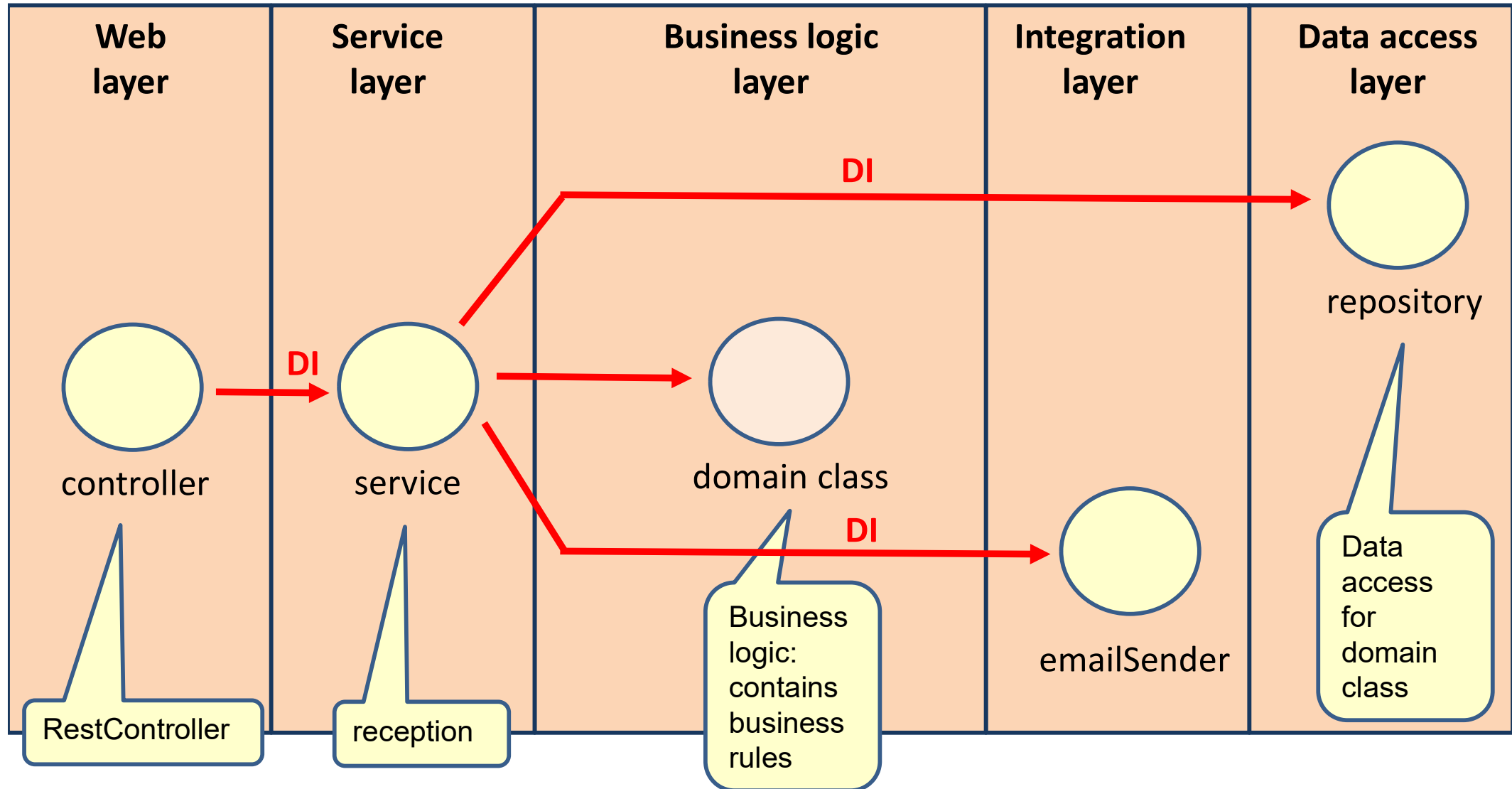
    public void addCustomer() {
        emailService.sendEmail();
    }
}
```



Field injection

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


Layered architecture

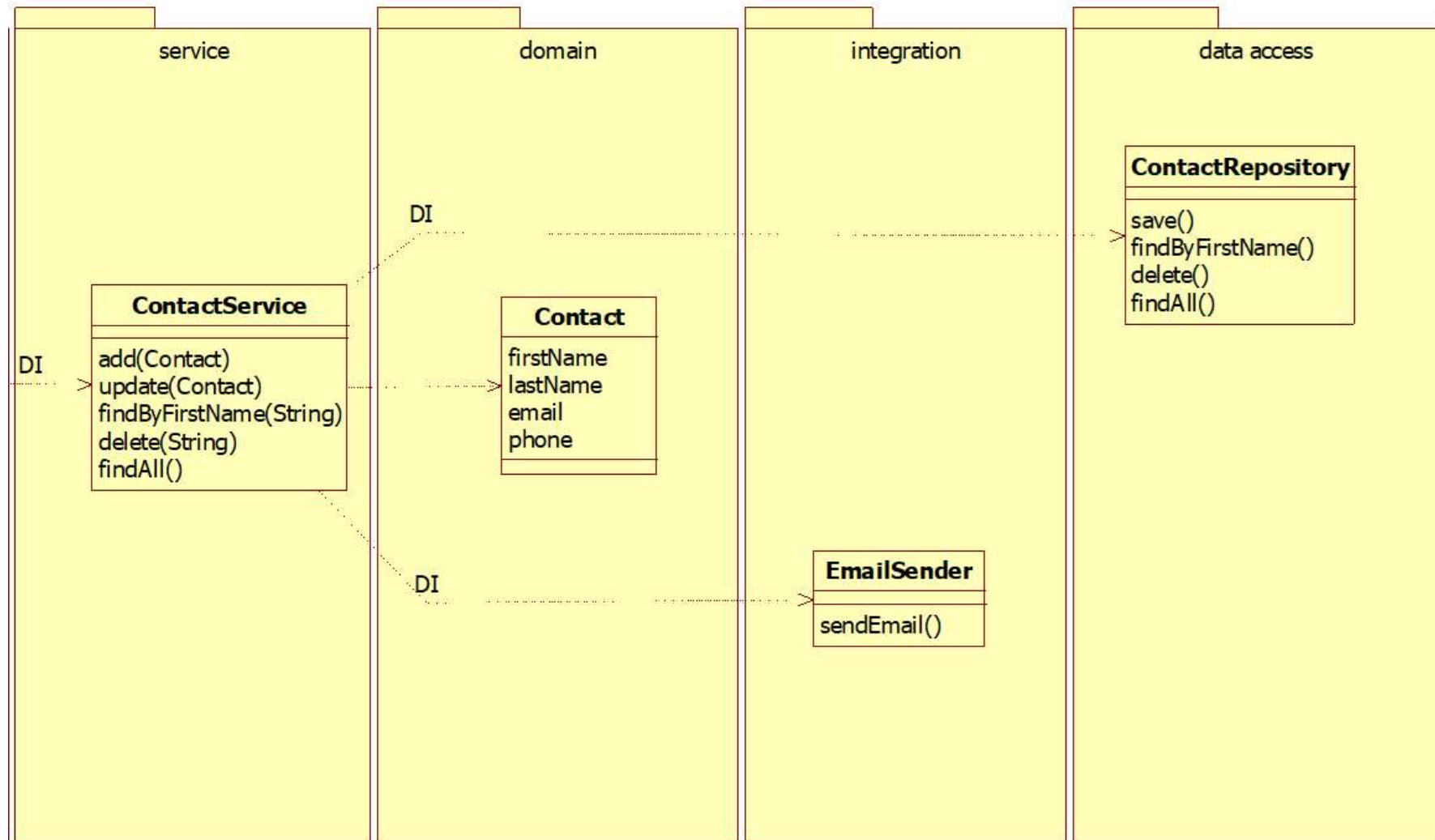


Main point

- An enterprise back-end system is typically divided in different layers.

Science of Consciousness: Life is found in layers.

Spring Boot example



Repository

@Repository

@Repository

```
public class ContactRepository {  
    private Map<String, Contact> contacts = new HashMap<String, Contact>();  
  
    public void save(Contact contact){  
        contacts.put(contact.getFirstName(),contact);  
    }  
  
    public Contact findByFirstName(String firstName){  
        return contacts.get(firstName);  
    }  
  
    public void delete(String firstName){  
        contacts.remove(firstName);  
    }  
  
    public Collection<Contact> findAll(){  
        return contacts.values();  
    }  
}
```

EmailSender

@Component

```
public class EmailSender {  
    public void sendEmail (String message, String emailAddress){  
        System.out.println("Send email message "+ message+" to"+emailAddress);  
    }  
}
```

@Component

Service

@Service

@Service

```
public class ContactService {
```

@Autowired

@Autowired

```
    ContactRepository contactRepository;
```

@Autowired

```
    EmailSender emailSender;
```

```
    public void add(Contact contact){
        contactRepository.save(contact);
        emailSender.sendEmail(contact.getEmail(), "Welcome");
    }
```

```
    public void update(Contact contact){
        contactRepository.save(contact);
    }
```

```
    public Contact findByFirstName(String firstName){
        return contactRepository.findByFirstName(firstName);
    }
```

```
    public void delete(String firstName){
        Contact contact = contactRepository.findByFirstName(firstName);
        emailSender.sendEmail(contact.getEmail(), "Good By");
        contactRepository.delete(firstName);
    }
```

```
    public Collection<Contact> findAll(){
        return contactRepository.findAll();
    }
```

```
}
```

Application



```
@SpringBootApplication
public class SpringBootMVCAApplication implements CommandLineRunner {
    @Autowired
    private ContactService contactService;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootMVCAApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        contactService.add(new Contact("Frank", "Brown", "fbrown@gmail.com", "4723459800"));
        System.out.println(contactService.findByFirstName("Frank"));
    }
}
```

ASPECT-ORIENTED PROGRAMMING

BASICS OF AOP

Crosscutting concern

- Check security for **every** service level method

```
public class CustomerService {  
  
    public void getAllCustomers() {  
        checkSecurity();  
        ...  
    }  
  
    public void getCustomer(long customerNumber) {  
        checkSecurity();  
        ...  
    }  
  
    public void addCustomer(long customerNumber, String firstName) {  
        checkSecurity();  
        ...  
    }  
  
    public void removeCustomer(long customerNumber) {  
        checkSecurity();  
        ...  
    }  
}
```

We have to call
checkSecurity() for all methods
of all service classes

Crosscutting concern

- Log **every** call to the database

```
public class AccountDAO {  
    public void saveAccount(Account account) {  
        ...  
        Logger.log("...");  
    }  
  
    public void updateAccount(Account account) {  
        ...  
        Logger.log("...");  
    }  
  
    public void loadAccount(long accountNumber) {  
        ...  
        Logger.log("...");  
    }  
  
    public void removeAccount(long accountNumber) {  
        ...  
        Logger.log("...");  
    }  
}
```

We have to call
Logger.log() for all methods of
all DAO classes

Good programming practice principles



DRY: Don't Repeat Yourself

- Write functionality at one place, and only at one place
- Avoid code scattering

SoC: Separation of Concern

- Separate business logic from (technical) plumbing code
- Avoid code tangling

AOP concepts



- Joinpoint
- Pointcut
- Aspect
- Advice
- Weaving

AOP concept: Joinpoint

- A specific point in the code

Joinpoint A

Joinpoint B

Joinpoint C

```
public class AccountDAO {  
    public void saveAccount(Account account) {  
        ...  
    }  
    public void updateAccount(Account account) {  
        ...  
    }  
    public void loadAccount(long accountNumber) {  
        ...  
    }  
    public void removeAccount(long accountNumber) {  
        ...  
    }  
}
```

AOP concept: Pointcut

- A collection of 1 or more joinpoints

Pointcut A: All methods of the AccountDAO class

Pointcut B: All methods of the AccountDAO class that have 1 parameter of type long

```
public class AccountDAO {  
    public void saveAccount(Account account) {  
        ...  
    }  
    public void updateAccount(Account account) {  
        ...  
    }  
    public void loadAccount(long accountNumber) {  
        ...  
    }  
    public void removeAccount(long accountNumber) {  
        ...  
    }  
}
```

AOP concept: Advice

- The implementation of the crosscutting concern

```
public class LoggingAdvice {  
    public void log() {  
        ...  
    }  
}
```

```
public class EmailAdvice {  
    public void sendEmailMessage() {  
        ...  
    }  
}
```


AOP concept: Aspect

- What crosscutting concern do I execute (=advice) at which locations in the code (=pointcut)
 - Aspect A: call the log() method of LoggingAdvice before every method call of AccountDAO
 - Aspect B: call the sendEmailMessage() method of EmailAdvice after every method call of AccountDAO that has one parameter of type long

```
public class AccountDAO {
```

```
    public void saveAccount(Account account) {  
        ...  
    }
```

```
    public void updateAccount(Account account) {  
        ...  
    }
```

```
    public void loadAccount(long accountNumber) {  
        ...  
    }
```

```
    public void removeAccount(long accountNumber) {  
        ...  
    }  
}
```

```
public class LoggingAdvice {
```

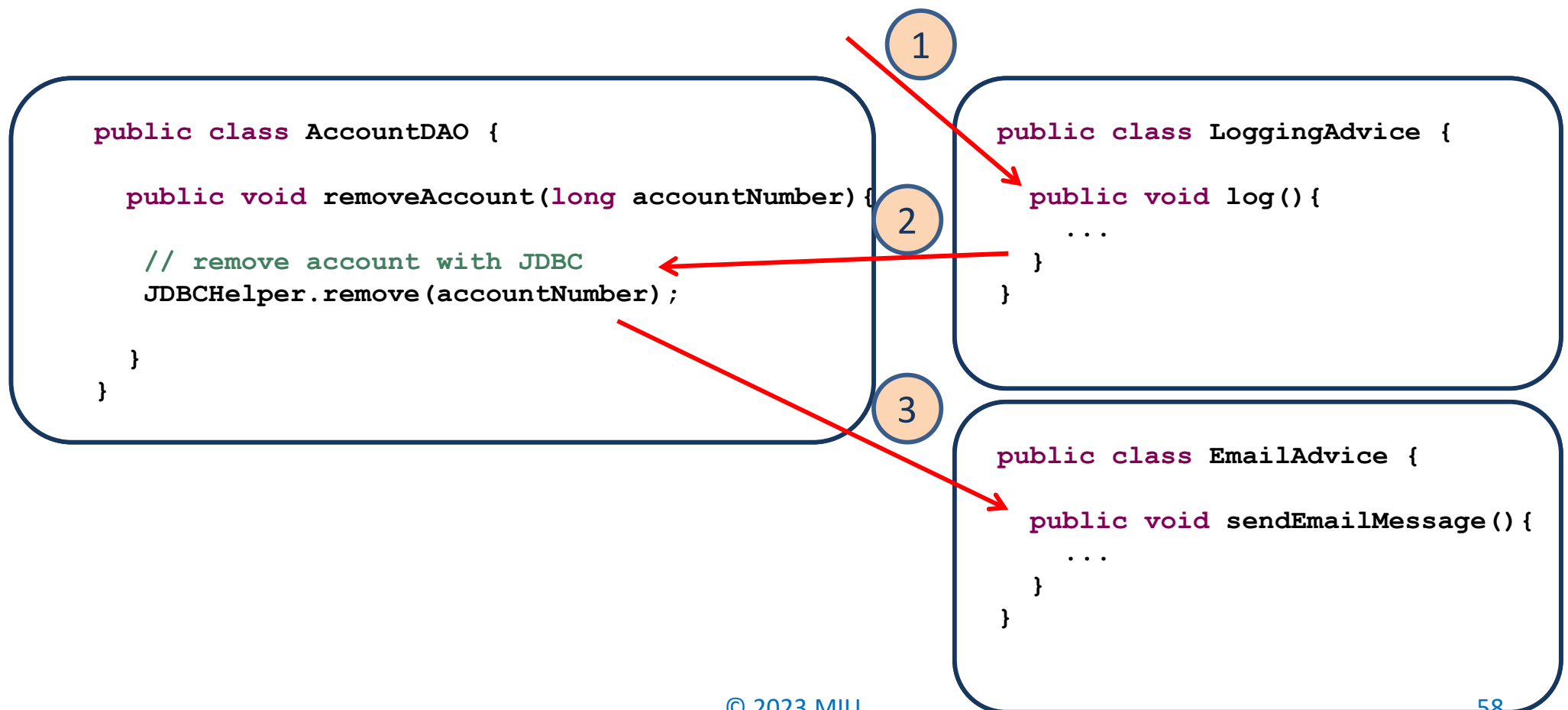
```
    public void log() {  
        ...  
    }
```

```
public class EmailAdvice {
```

```
    public void sendEmailMessage() {  
        ...  
    }
```

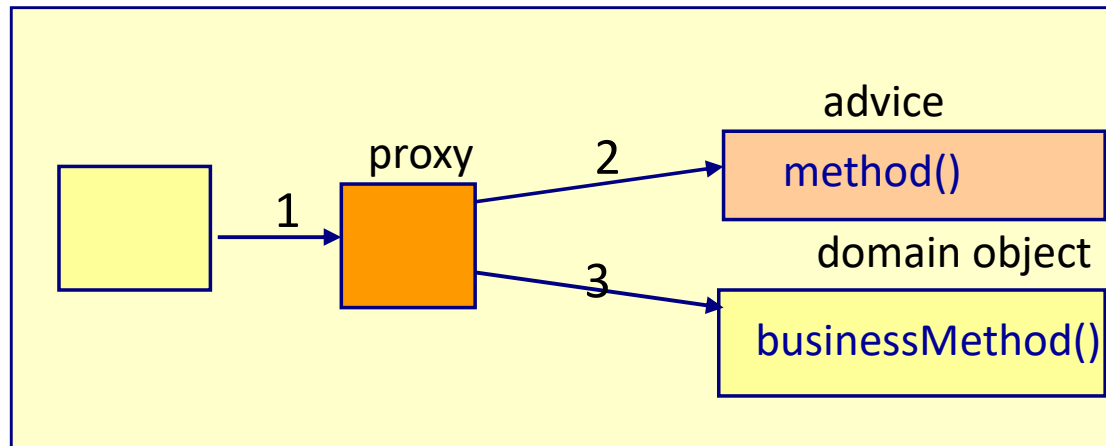
AOP concept: Weaving

- Weave the advice code together with the target code at the corresponding pointcuts such that we get the correct execution

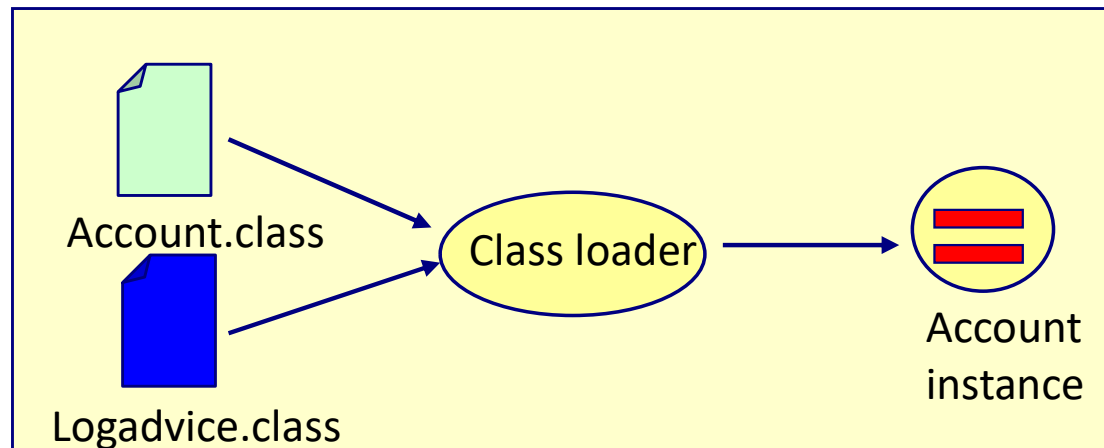


Weaving

Proxy-based weaving



Load time weaving

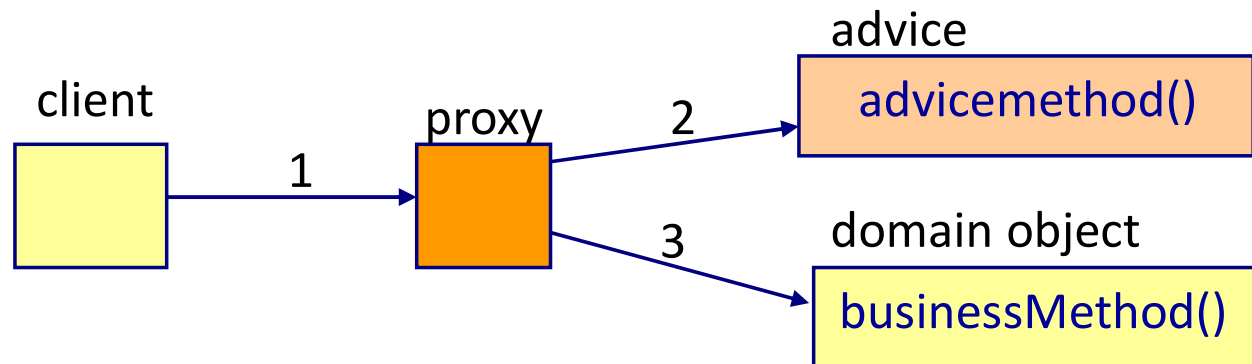


Advice types

- Before
- After returning
- After throwing
- After (finally)
- Around

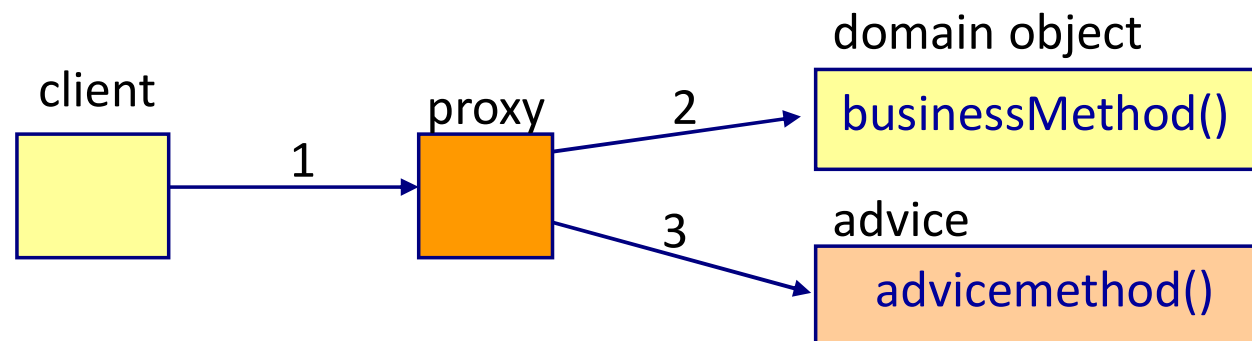
Before advice

- First call the advice method and then the business logic method



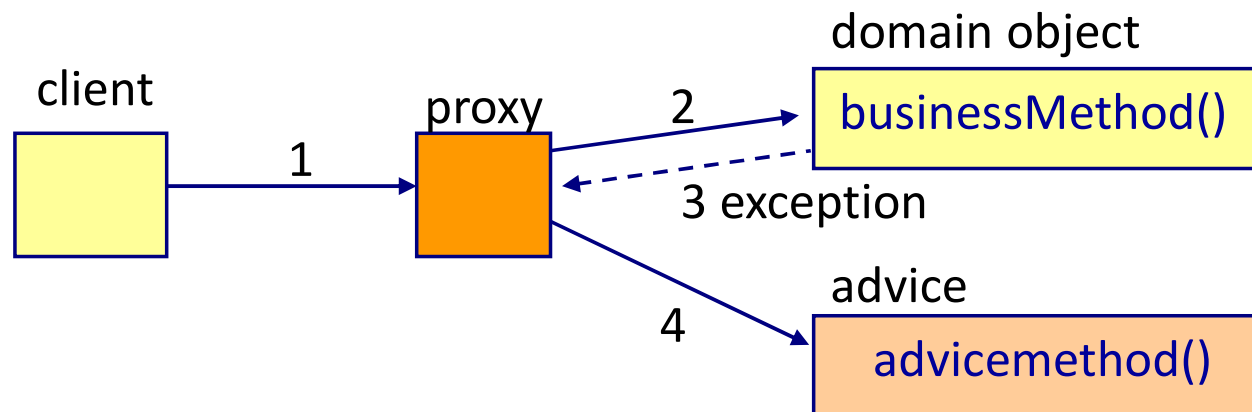
After returning advice

- First call the business logic method and when this business logic method returns normally without an exception, then call the advice method



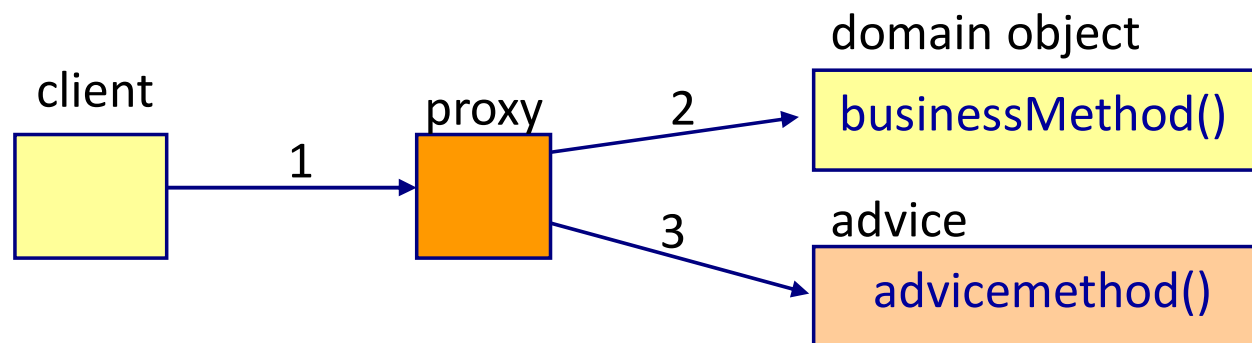
After throwing advice

- First call the business logic method and when this business logic method throws an exception, then call the advice method



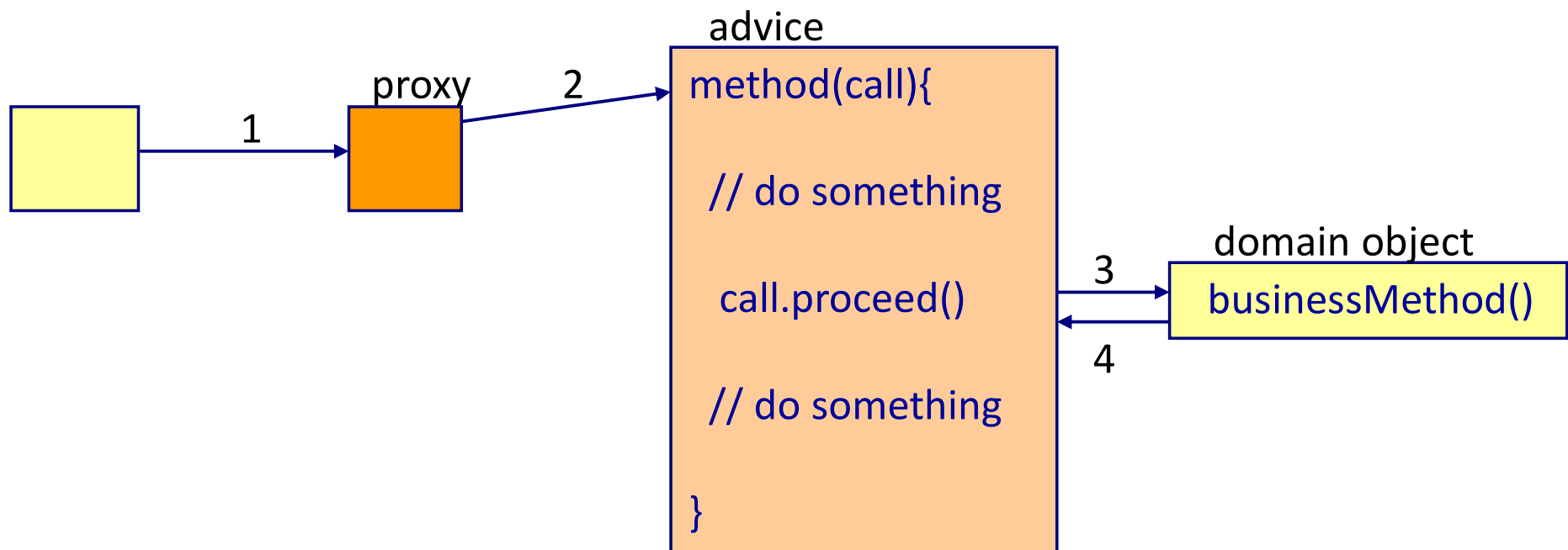
After advice

- First call the business logic method and then call the advice method (independent of how the business logic method returned: normally or with exception)



Around advice

- First call the advice method. The advice method calls the business logic method, and when the business logic method returns, we get back to the advice method



AOP with Spring Boot

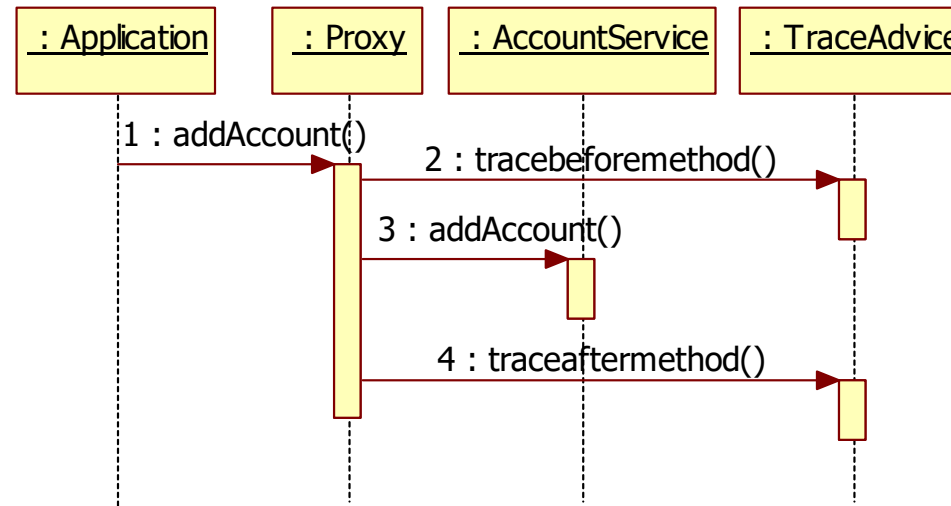
```
public class AccountService implements IAccountService{
    Collection<Account> accountList = new ArrayList();

    public void addAccount(String accountNumber, Customer customer){
        Account account = new Account(accountNumber, customer);
        accountList.add(account);
        System.out.println("in execution of method addAccount");
    }
}
```

@Configuration

```
@Aspect
@Configuration
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

AOP with Spring Boot



```
@Aspect
@Configuration
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

Pointcut execution language

Pointcut execution language

```
@Aspect
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

Pointcut execution language

```
■ @Before("execution(public * *.*(..))")
```

Visibility:

- Possibilities:
 - private
 - public
 - Protected
- Optional
- Cannot be *

Return type:

- The return type of the corresponding method(s)
- Not optional
- Can be *

package.class.method(args):

- Name of the package can also be *
- Name of the class can also be *
- Name of the method can also be *
- Arguments can be ..
- Not optional
- Can also be *.*(..)
- Can also be *(..)

Pointcut execution language examples



```
@After("execution(public * *(..))")
```

All public methods

```
@After("execution(public void *(..))")
```

All public methods
that return void

```
@After("execution(* order.*.*(..))")
```

All methods from all
classes in the order
package

```
@After("execution(* *.*.create*(..))")
```

All methods that
start with create

```
@After("execution(* *.Customer.*(..))")
```

All methods from
the Customer class

Pointcut execution language examples

```
@After("execution(* order.Customer.*(..))")
```

All methods from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(..))")
```

The getPayment () method from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(int))")
```

The getPayment () method with a parameter of type int from the Customer class in the order package

```
@After("execution(* *.*.*(long,String))")
```

All methods from all classes that have 2 parameters, the first of type long, and the second of type String

Around example

```
@Around("execution(* *.*(..))")
public Object profile (ProceedingJoinPoint call) throws Throwable{
    Stopwatch clock = new Stopwatch("");
    clock.start(call.toShortString());

    Object object= call.proceed();

    clock.stop();
    System.out.println(clock.prettyPrint());
    return object;
}
```

Create and start a stopwatch

Call the business logic method

Stop the stopwatch and
print result

```
StopWatch '': running time (millis) = 1
```

```
-----
ms      %      Task name
```

```
-----
00001   100%   execution(addAccount)
```


Getting the return value

■ Works only for @AfterReturning

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

getName() returns a String

The pointcut expression

Add 'returning' parameter

```
@Aspect  
public class TraceAdvice {  
    @AfterReturning(pointcut="execution(* mypackage.Customer.getName(..))", returning="retValue")  
    public void tracemethod(JoinPoint joinpoint, String retValue) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("return value =" + retValue);  
    }  
}
```

Add parameter to the advice method.
The name of the parameter must be the same as the name of the returning parameter of the @AfterReturning annotation

Getting the return value

```
public class Customer {  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

getAge() returns an integer

Add 'returning' parameter

```
@Aspect  
public class TraceAdvice {  
    @AfterReturning(pointcut="execution(* mypackage.Customer.getAge(..))", returning="retValue")  
    public void tracemethod(JoinPoint joinpoint, int retValue) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("return value =" + retValue);  
    }  
}
```

retValue is an int

Getting the exception

- Works only for @AfterThrowing

```
public class Customer {  
    public void myMethod() throws MyException{  
        throw new MyException("myexception");  
    }  
}
```

```
public class MyException extends Exception{  
    private String message;  
  
    public MyException(String message){  
        this.message=message;  
    }  
    public String getMessage(){  
        return message;  
    }  
}
```

```
@Aspect  
public class TraceAdvice {  
    @AfterThrowing(pointcut="execution(* mypackage.Customer.myMethod(..))",throwing="exception")  
    public void tracemethod(JoinPoint joinpoint, MyException exception) {  
        System.out.println("method =" +joinpoint.getSignature().getName());  
        System.out.println("exception message =" +exception.getMessage());  
    }  
}
```

Add 'throwing' parameter

Add parameter to the advice method

Get parameters

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..)) && args(name)")  
    public void tracemethod(JoinPoint joinpoint, String name) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
    }  
}
```

Add 'args' parameter

Add parameter(s) to the advice method

Get parameters

```
public class Customer {  
    private String name;  
    private int age;  
  
    public void setNameAndAge(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

2 parameters

```
@Aspect  
public class TraceAdvice {  
    @Before("execution(* mypackage.Customer.setNameAndAge(..)) && args(name,age)")  
    public void tracemethod(JoinPoint joinpoint, String name, int age) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
        System.out.println("parameter age =" + age);  
    }  
}
```

Add name and age to the args parameter

Add 2 parameters to the advice method

Get parameters from the Joinpoint

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Get the arguments from
the joinpoint

Take the first argument

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..))")  
    public void tracemethodA(JoinPoint joinpoint) {  
        Object[] args = joinpoint.getArgs();  
        String name = (String)args[0];  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
    }  
}
```

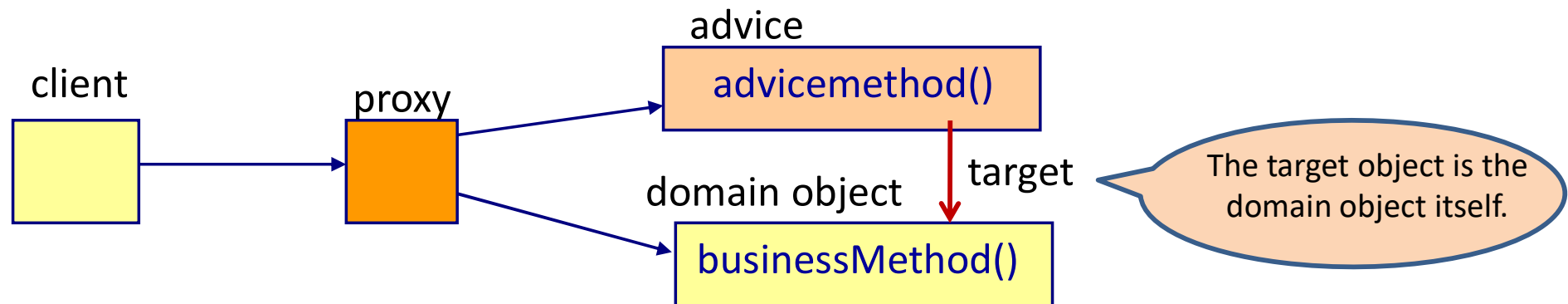
Get multiple parameters from the Joinpoint

```
public class Customer {  
    private String name;  
    private int age;  
  
    public void setNameAndAge(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

2 parameters

```
@Aspect  
public class TraceAdvice {  
    @Before("execution(* mypackage.Customer.setNameAndAge(..))")  
    public void tracemethod(JoinPoint joinpoint) {  
        Object[] args = joinpoint.getArgs();  
        String name = (String)args[0];  
        int age = (Integer)args[1];  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
        System.out.println("parameter age =" + age);  
    }  
}
```

The target class



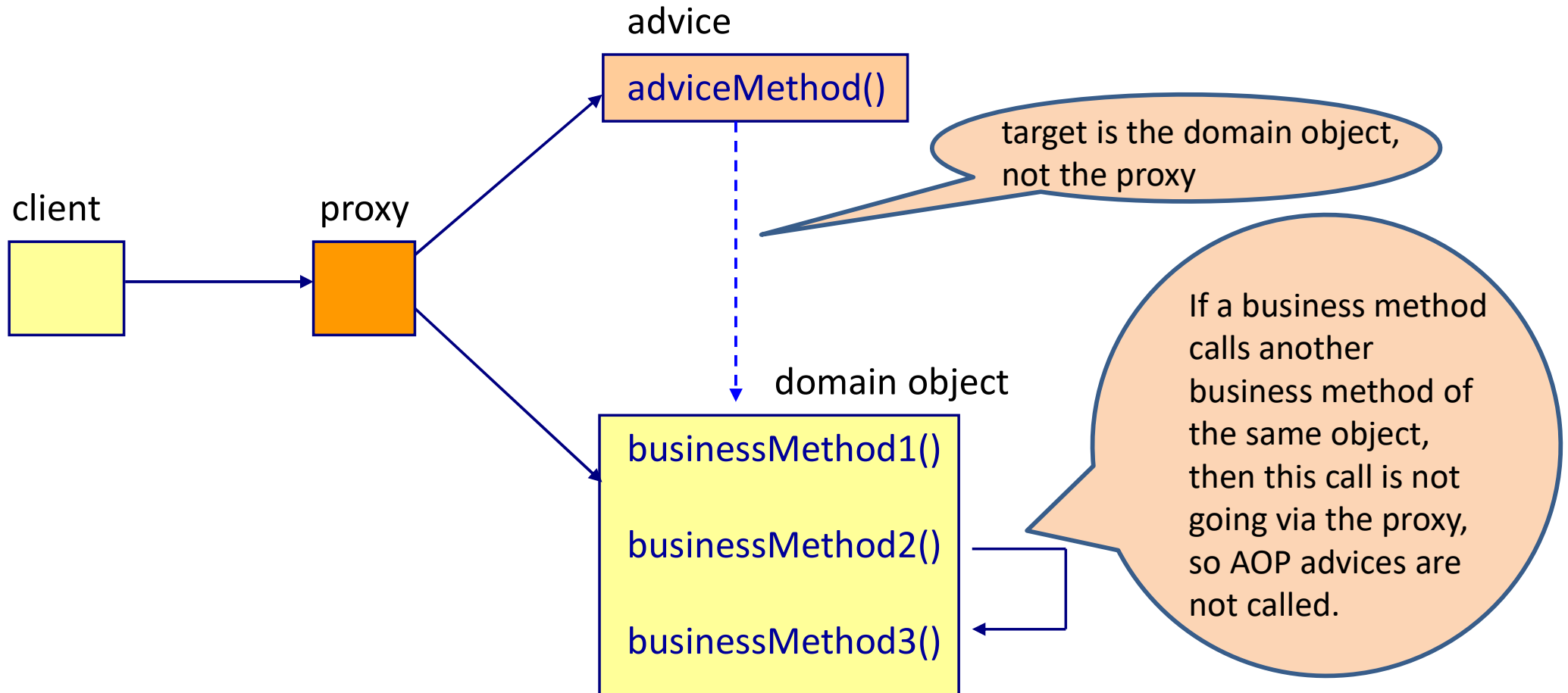
Get the target class

```
public class Customer {  
    private String name;  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Get the target object from
the joinpoint

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..))")  
    public void tracemethod(JoinPoint joinpoint) {  
        Customer customer = (Customer)joinpoint.getTarget();  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("customer age =" + customer.getAge());  
    }  
}
```

Disadvantage of a proxy



Advantages of AOP

- No code tangling
 - Clean separation of business logic and plumbing code
- No code scattering

Disadvantages of AOP

- You don't have a clear overview of which code runs when
- A pointcut expression is a string that is parsed at runtime
 - No compile time checking of the pointcut expression
- You make mistakes easily
- Problems with proxy-based AOP

Be careful with AOP: always use unit testing and integration testing with AOP

Main point

- Aspect Oriented Programming lets us program additional logic in one place, and then declaratively apply that logic to many places.
Science of Consciousness: We create harmony (single implementation), in diversity (applied to many places).

Connecting the parts of knowledge with the wholeness of knowledge

1. Layering is a powerful technique to separate different aspects of a system
 2. The service class is the connection point between the different layers
-

3. **Transcendental consciousness** is the source of all intelligence of creation.
4. **Wholeness moving within itself:** In unity consciousness, one experiences that everything is just an expression of one's own Self.

