

Microservices Development (using Spring Boot)

K. VENKATA RAMANA



Venkata Ramana

Over three decades of IT experience in Corporate Training, Software development and architectural designs involving web technologies, databases, SOA, Microservices & Cloud.

❑ Specification

❑ Framework

❑ Pattern

Specification

Provides API , standards, recommended practices, codes
And technical publications, reports and studies.

JCP - Java Community Process

JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)

The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)

The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)

Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

JSR 311,370 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)

A **framework** is a body of pre-written code that acts as a template or skeleton, which a developer can then use to create an application by filling in their own code as needed to get the app to work as they intend it to.

A framework is created to be used over and over so that developers can program their application without the manual overhead of creating every line of code from scratch.

Java frameworks are bodies of prewritten code used by developers to create apps using the Java programming language.

A Java framework is a type of framework specific to the Java programming language, used as a platform for developing software applications and Java programs.

Ex: Spring, Hibernate, Spring Boot etc.,

Design Pattern

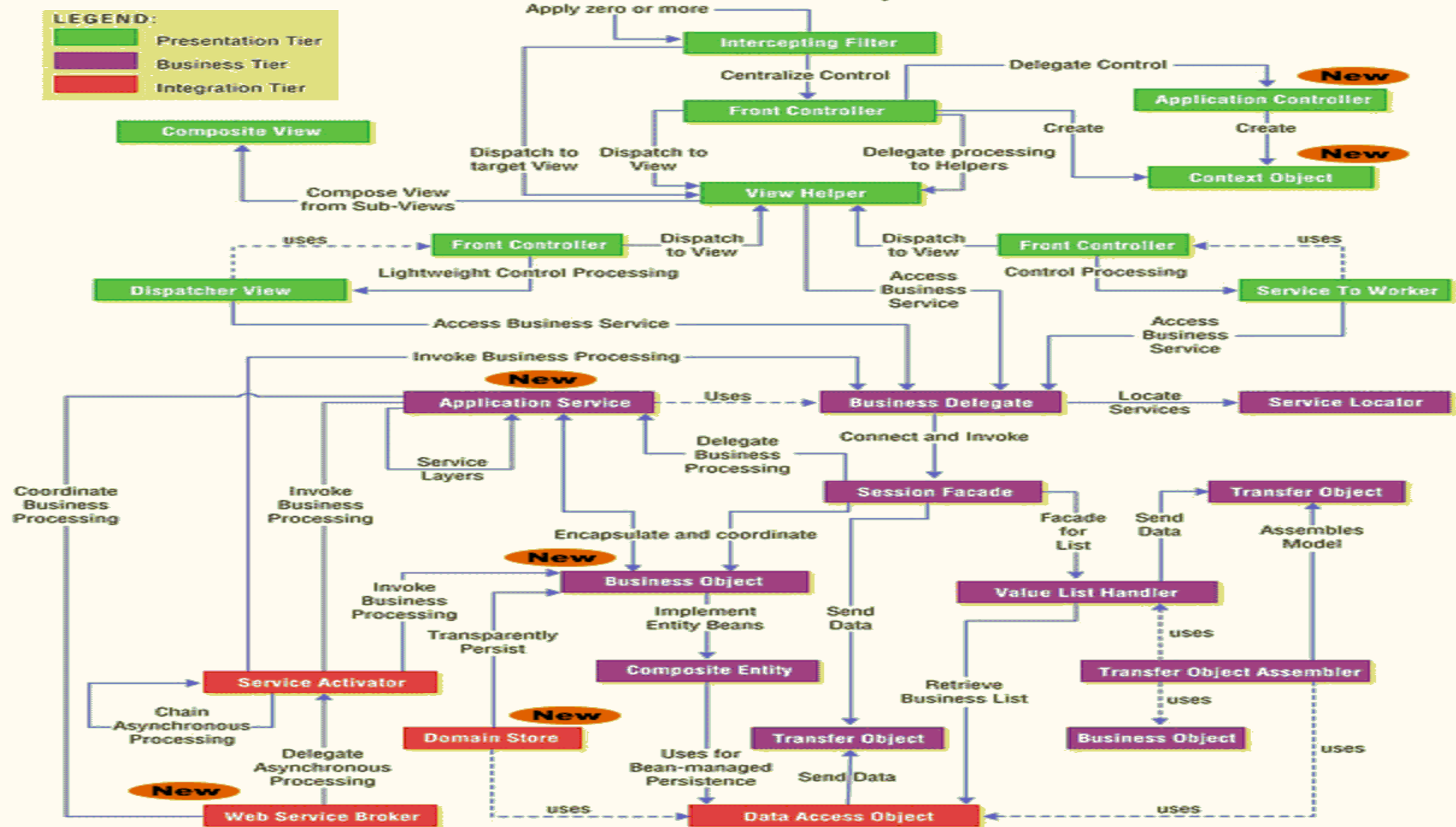
In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design

GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
SCOPE	Class	Factory Method	Class Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Core J2EE Patterns, 2nd Edition

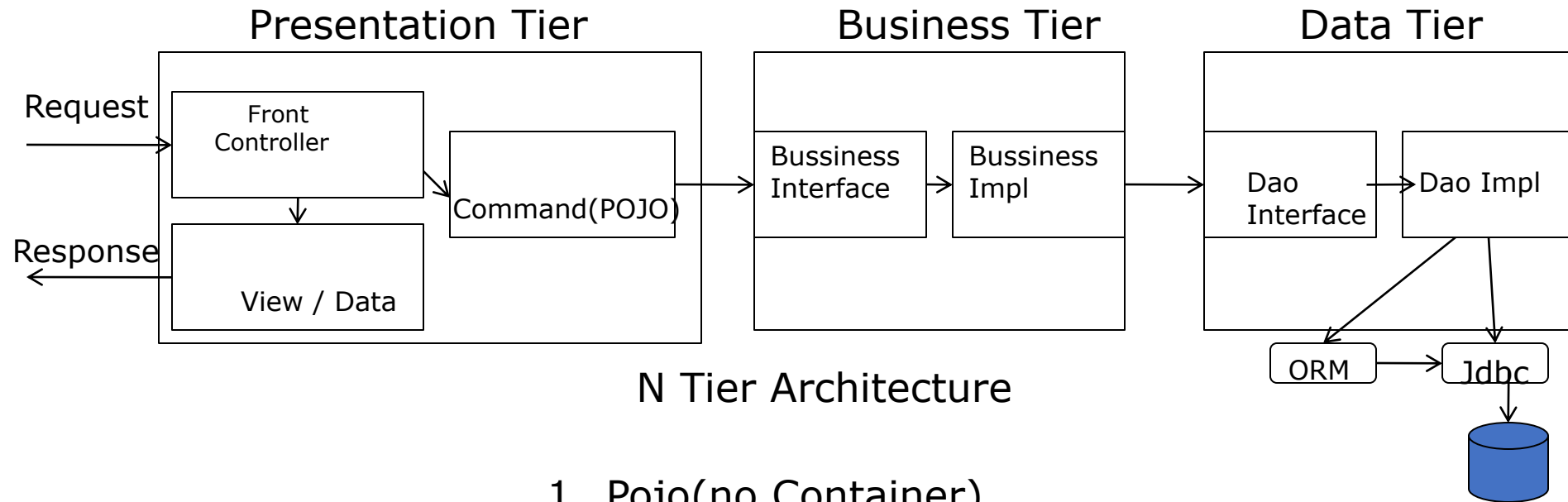


RESTful API Patterns

- ❑ Statelessness
- ❑ Content Negotiation
- ❑ URI Templates
- ❑ Pagination
- ❑ Versioning
- ❑ Authorization
- ❑ API facade
- ❑ Discoverability
- ❑ Idempotent
- ❑ Circuit breaker

Microservice Patterns:

- ☐ API gateway
- ☐ Service registry
- ☐ Circuit breaker
- ☐ Messaging
- ☐ Database per Service
- ☐ Access Token
- ☐ Saga
- ☐ Event Sourcing & CQRS



N Tier Architecture

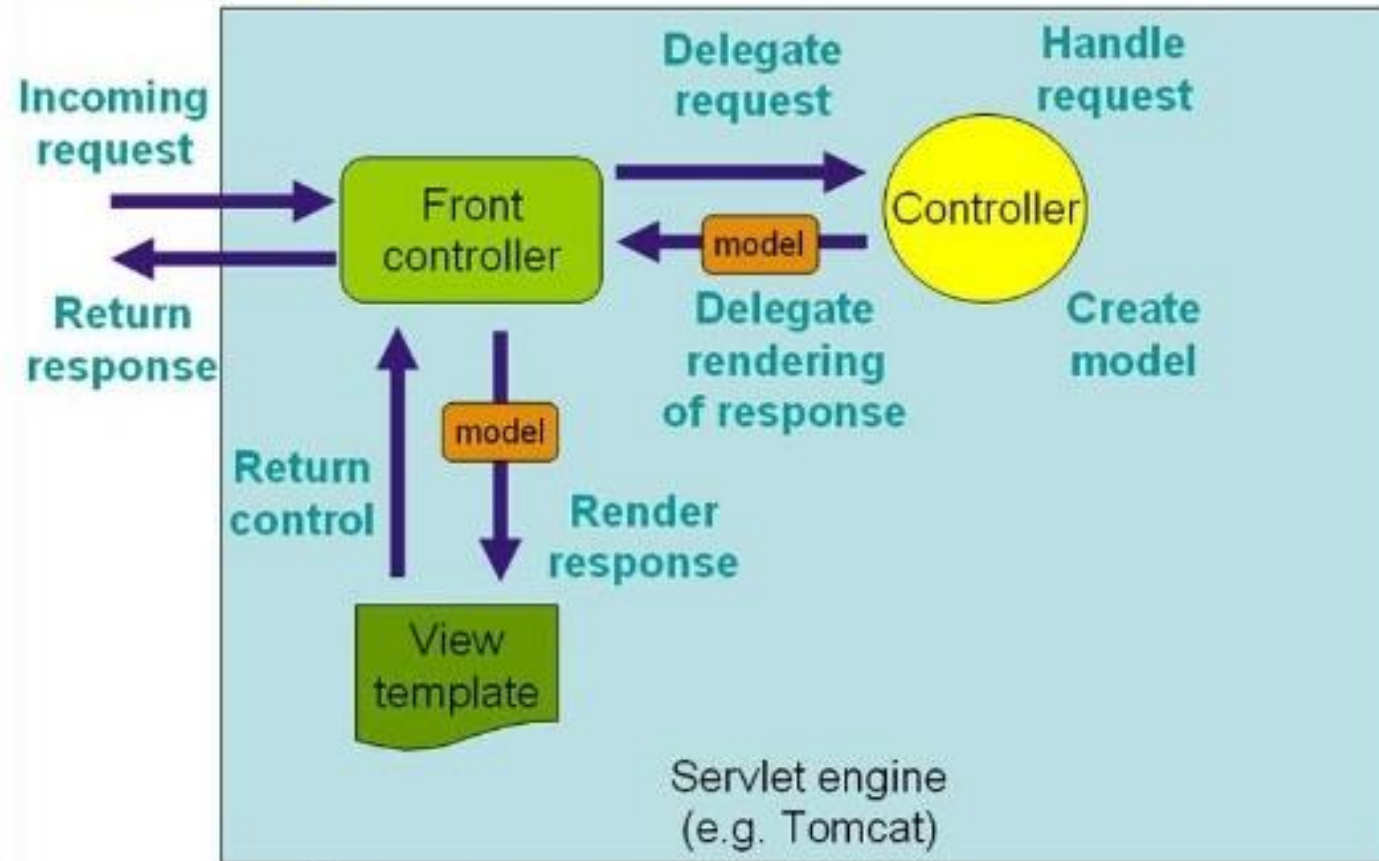
1. Servlet/jsp
2. MVC
 - Struts
 - JSF
 - Flex
 - Gwt
 - Spring MVC
 - ...

1. Pojo(no Container)
2. Ejb 2.x(HW Container)
 - Session Bean
 - Mdb
3. Pojo + LW Container
 - Spring
 - Microcontainer
 - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
 - Hibernate
 - Kodo
 - Toplink
 - MyBatis
5. JPA

+ Spring Templates

Spring MVC Workflow



Source: www.springsource.org

Web Application:

It is an end-to-end solution for a user.

Which means, User can:


- ❑ Open it using a browser Interact with it.
- ❑ User can click on something and after some processing, its result will be reflected in the browser screen. Human-System interaction.

Web API / Web service

With Web APIs alone, a user can not interact with it, because it only returns data, not views.

- ❑ It is a system which interacts with another system
- ❑ It does not return views, it returns data
- ❑ It has an endpoint set, which can be hit by other systems to get data which it provides.

← → ↻ spicejet.com

 **BOOK** ADD-ONS DEALS GIFT CARD SP

✈ Flights 🏨 Hotels 🌴 Holiday Packages 🕒 Flight S

☒ One Way ☐ Round Trip ☐ Multicity

*FROM *TO *DEPART DATE

← → ↻ yatra.com

yatra Flights Hotels

Book Flights, Hotels and Holiday Packages

Depart From
New Delhi
DEL

<html>

Web MVC

Web API

<xml />

{JSON}



Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- ✓ It is used to achieve abstraction.
- ✓ By interface, we can support the functionality of multiple inheritance.
- ✓ It can be used to achieve loose coupling.

It is used to achieve abstraction.

1

2

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

3

Loose Coupling

While using an interface, we define the method separately and the signature separately.

This way, all the methods, and classes are entirely independent and achieves Loose Coupling.

```
interface Drawable{  
    void draw();  
}
```

```
class Rectangle implements Drawable{  
    public void draw()  
    {System.out.println("drawing rectangle");}  
}
```

```
class Circle implements Drawable{  
    public void draw()  
    {System.out.println("drawing circle");}  
}
```

```
interface Account{  
    float rateOfInterest();  
}
```

```
class SavingAccount implements Account{  
    public float rateOfInterest(){return 9.15f;}  
}
```

```
class CurrentAccount implements Account{  
    public float rateOfInterest(){return 9.7f;}  
}
```

```
interface ICustomerDao
{
    List<Customer> getCustomers();
}
```

```
class CustomerJdbcDao implements ICustomerDao
{
    List<Customer> getCustomers() { }
```

```
class CustomerHibernateDao implements ICustomerDao
{
    List<Customer> getCustomers() { }
```

Java 7, we can have abstract methods and variables. It cannot have a method body.

Java 8, we can have default and static methods in an interface.

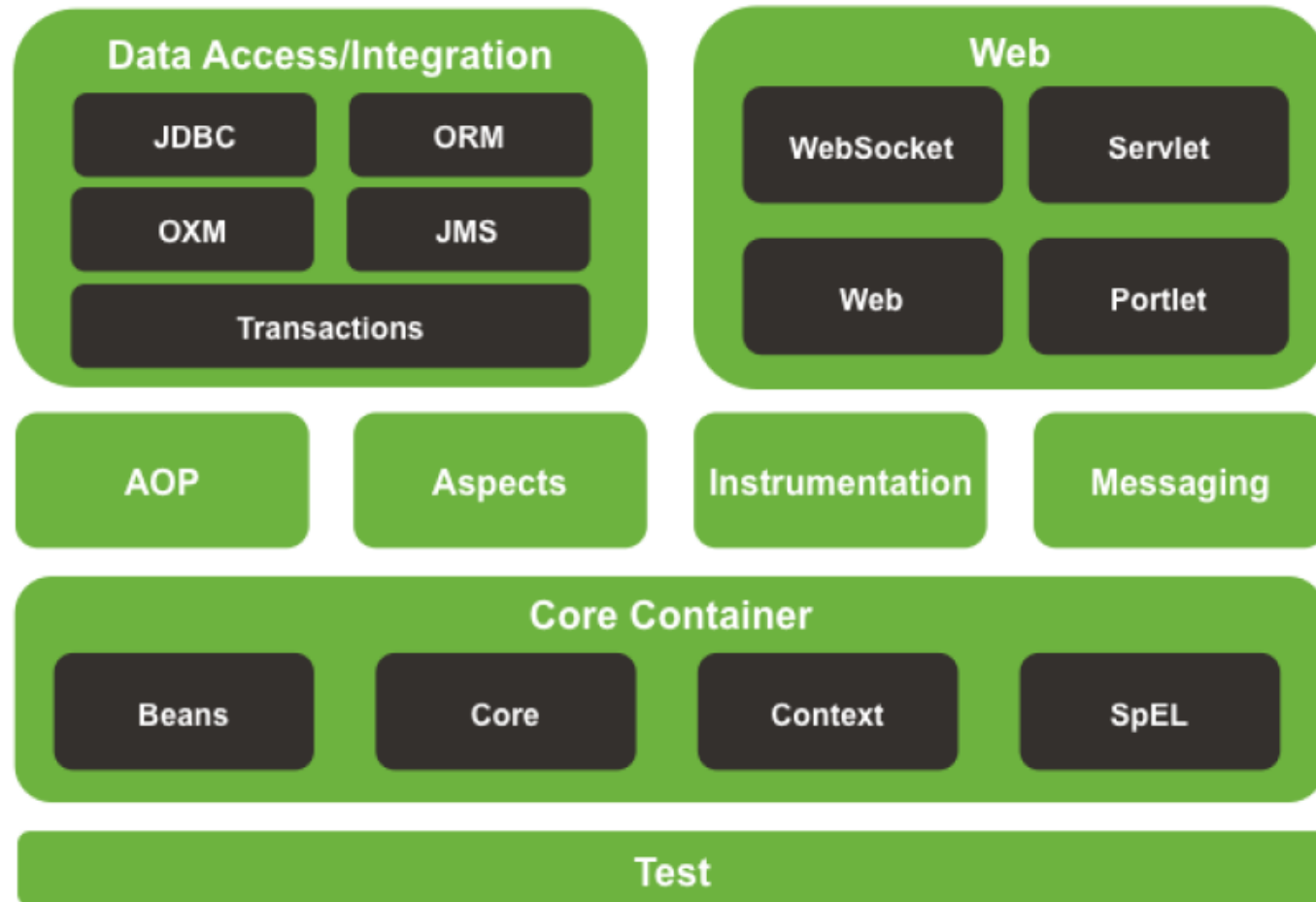
Java 9, we can have private methods in an interface.

Spring Framework Features:

1. Core Container - IOC, DI & AOP
Application Context (Object creation – Singleton/Prototype)
Dependency Injection (Object Graph Creation)
2. Data Access Layer (Template pattern)
3. Spring MVC & REST
4. Other features like Security, Messaging, Spring Integration, Spring Batch, Spring Cloud etc.,



Spring Framework Runtime



Core Container

The Core Container consists of the spring-core, spring-beans, spring-context, springcontext-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern.

The Context (spring-context) module builds on the solid base provided by the Core and Beans modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization. The ApplicationContext interface is the focal point of the Context module.

spring-context-support provides support for integrating common third-party libraries into a Spring application context for caching (EhCache, Guava, JCache), mailing (JavaMail), scheduling (CommonJ, Quartz) and template engines (FreeMarker, JasperReports, Velocity).

The spring-expression module provides a powerful Expression Language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification.

The spring-aop module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

The separate spring-aspects module provides integration with AspectJ. The spring-instrument module provides class instrumentation support and classloader implementations to be used in certain application servers.

Messaging

Spring Framework 4 includes a spring-messaging module with key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

The spring-jdbc module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The spring-tx module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

The spring-orm module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, and Hibernate. Using the spring-orm module you can use all of these O/R mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature.

The spring-oxm module provides an abstraction layer that supports Object/XML mapping implementations such as JAXB, Castor, XMLBeans, JiBX and XStream.

The spring-jms module (Java Messaging Service) contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the spring-messaging module.

Web

The Web layer consists of the spring-web, spring-webmvc, spring-websocket, and springwebmvc-portlet modules.

The spring-web module provides basic web-oriented integration features such as multipart file upload functionality and the initialization of the IoC container using Servlet listeners and a web-oriented application context. It also contains an HTTP client and the web-related parts of Spring's remoting support.

The spring-webmvc module (also known as the Web-Servlet module) contains Spring's modelview-controller (MVC) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

The spring-webmvc-portlet module (also known as the Web-Portlet module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the Servlet-based spring-webmvc module.

Test

The spring-test module supports the unit testing and integration testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation

IOC is used to decouple common task from implementation.

Six basic techniques to implement Inversion of Control.

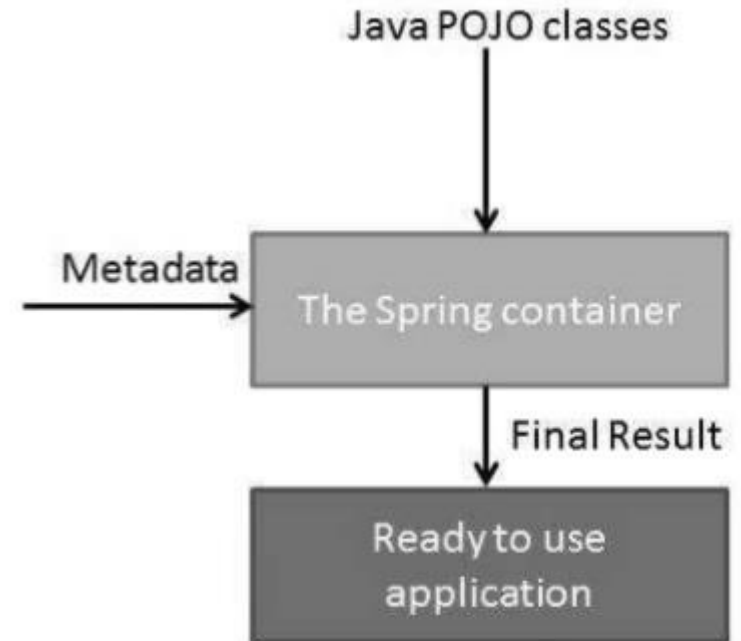
These are:

- 1.using a factory pattern
- 2.using a service locator pattern
- 3.using a constructor injection
- 4.using a setter injection
- 5.using an interface injection
- 6.using a contextualized lookup

Constructor, setter, and interface injection are all aspects of Dependency injection.

What is a container?

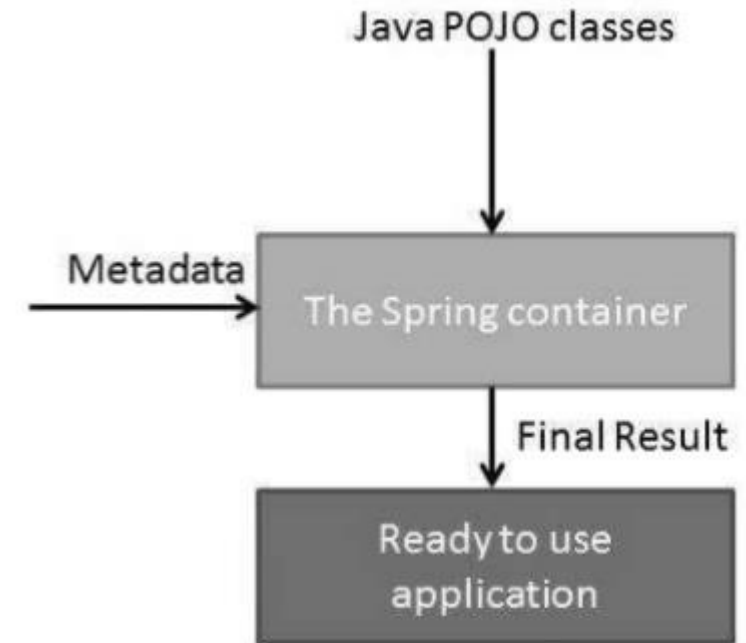
- ✓ The container will create the objects,
- ✓ wire them together,
- ✓ configure them,
- ✓ and manage their complete life cycle from creation till destruction.



The container gets its instructions on

- ✓ what objects to instantiate,
- ✓ configure,
- ✓ and assemble by reading the configuration metadata provided.
- ✓ The configuration metadata can be represented either by XML or Annotation.

- ❑ Apache Tomcat is a Servlet Container.
- ❑ Weblogic/WebSphere/JBoss provides EJB Container
- ❑ Spring is a POJO container.



Spring XML Configuration

```
package demo.web;
```

```
Class AccountDaoJdbc implements AccountDao  
{
```

```
DataSource dataSource;
```

```
Public void setDataSource(DataSource datasource)  
{  
dataSource = datasource;  
}  
..  
}
```

```
<bean id = "accountDao" class="demo.web.AccountDaoJdbc">  
  <property name="datasource">  
    <ref bean="dataSource"/>  
  </property>  
</bean>
```

Spring Annotation Configuration

```
package demo.web;
```

```
@Repository("accountDao")
```

```
Class AccountDaoJdbc implements AccountDao
```

```
{
```

```
@Autowired
```

```
DataSource dataSource;
```

```
Public void setDataSource(DataSource datasource)
```

```
{
```

```
dataSource = datasource;
```

```
}
```

```
..
```

```
}
```

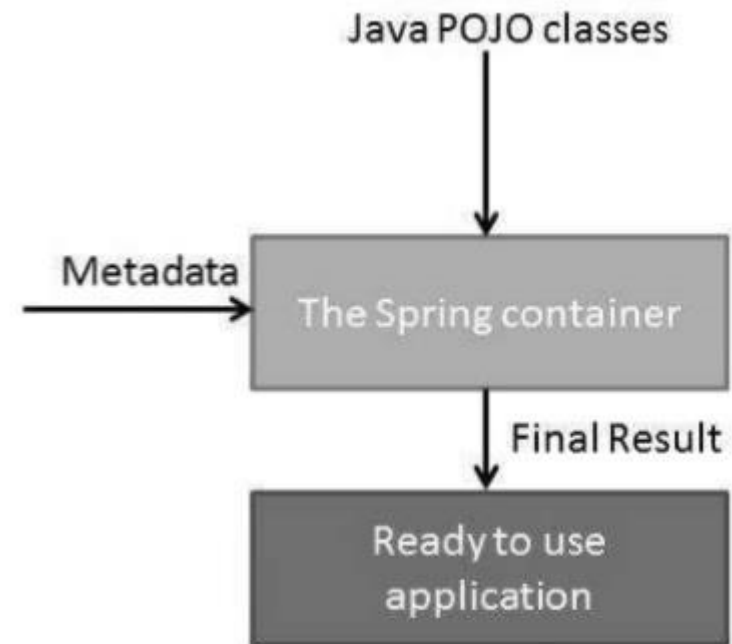
```
<bean id="accountDao" class="demo.web.AccountDaoJdbc">  
  <property name="dataSource">  
    <ref bean="dataSource"/>  
  </property>  
</bean>
```


Lifecycle Management

Bean Managed Life cycle

```
AccountBean acc = new AccountBean();  
acc.setAmt(5000);  
....  
acc=null;
```

Container Managed Life cycle



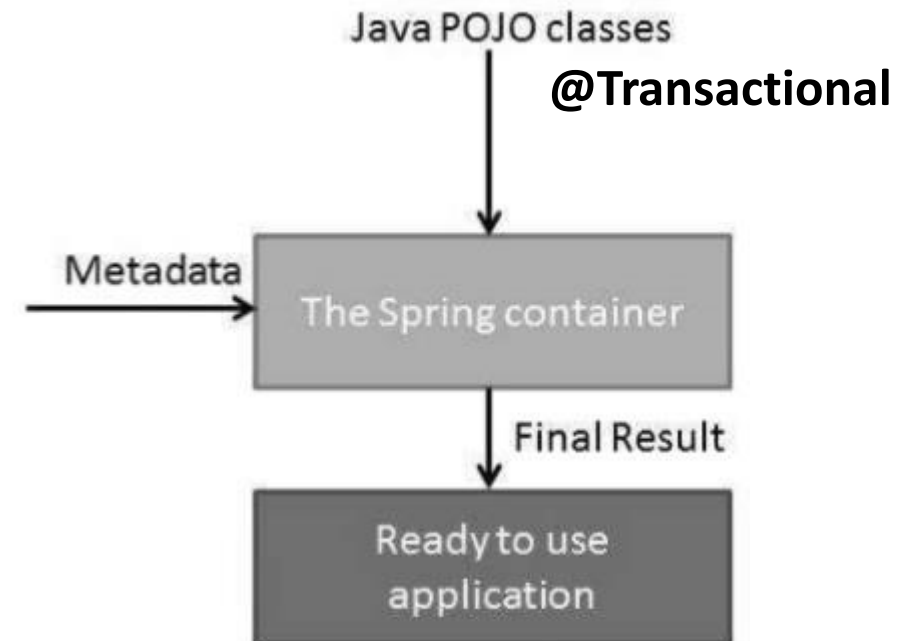
```
ApplicationContext ctx = .....  
AccountBean acc = ctx.getBean("account");
```

Container Managed Services

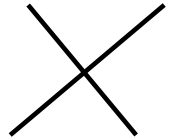
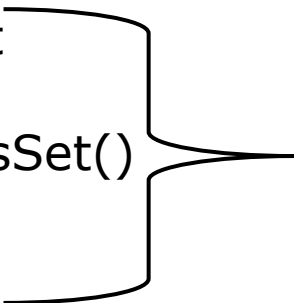
Bean managed transactions

~~Transaction tx=
tx.begin;
database operations;
tx.commit() / tx.rollback();~~

Container managed transactions

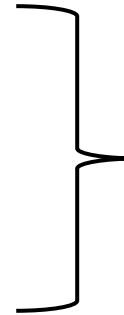


Life cycle of a bean within Spring application context

1. Instantiate
2. Populate properties(DI)
3. Aware interfaces [BeanNameAware's setBeanName()
ApplicationContextAware's setApplicationContext () 
4. **Pre-initialization Bean Post processors**
5. Any method with @PostConstruct
6. InitializingBean's afterPropertiesSet()  (use @PostConstruct)
7. Call custom init-method
8. **Post-initialization BeanPostProcessors**
9. Now Bean is ready to use

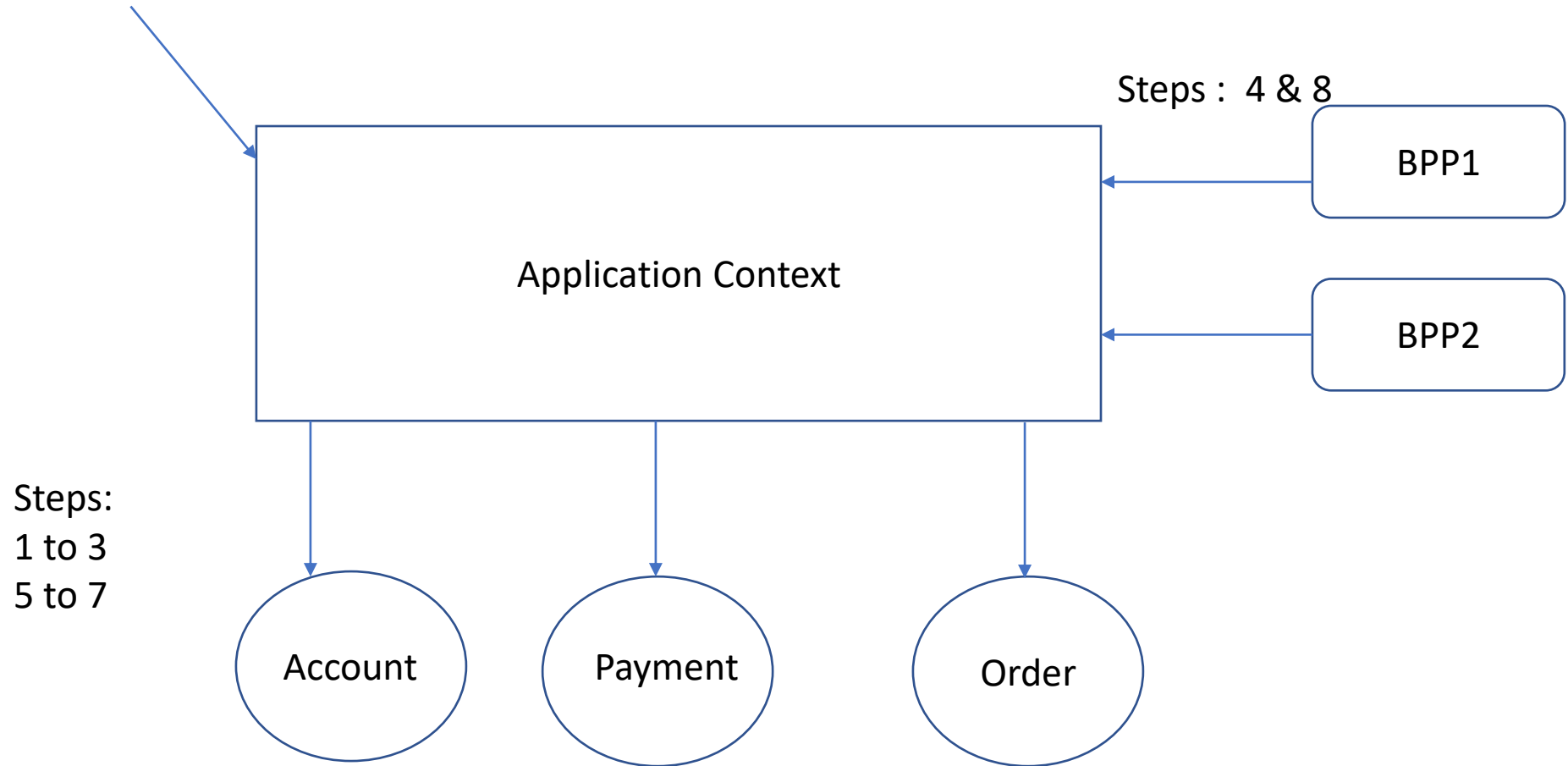
Container is shutdown

- 1.Any method with @PreDestroy
- 2.DisposableBean's destroy
- 3.Call custom destroy-method



Use @PreDestroy

```
ApplicationContext context = new .....  
Account account = context.getBean("account")
```



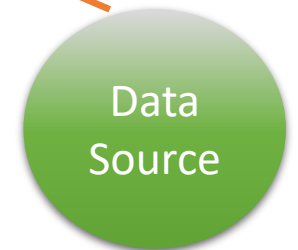
Presentation Tier



Business Tier



Data Tier



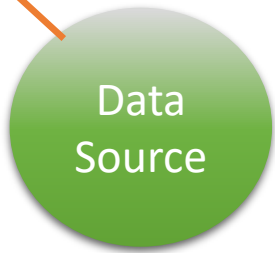
Presentation Tier



Business Tier



Data Tier



@RestController

```
Class AccountController
{
  @Autowired
  AccountService accService;
  ....
}
```

@Service

```
Class AccountServiceBean
  implements AccountService
{
  @Autowired
  AccountDao accountDao;
  ....
}
```

@Repository

```
Class AccountDaoJdbc
  implements AccountDao
{
  @Autowired
  DataSource dataSource;
  ....
}
```

Spring AOP

Spring AOP

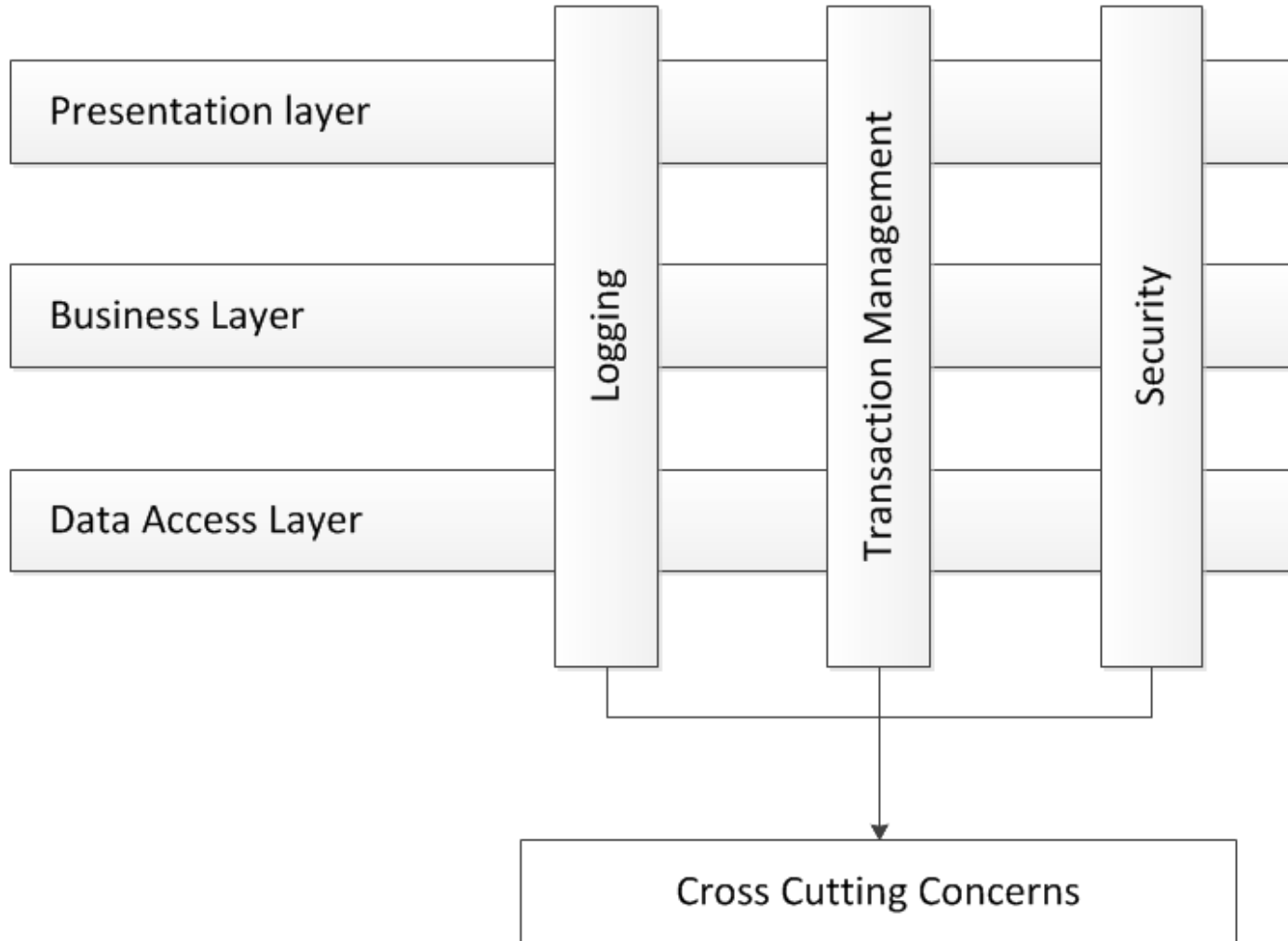
Spring AOP enables Aspect-Oriented Programming in spring applications.

In AOP, aspects enable the modularization of concerns such as :

- ☐ transaction management,
- ☐ Logging,
- ☐ Exception Handling
- ☐ Security etc.,

(often termed crosscutting concerns)

Cross Cutting Concerns



Client Application

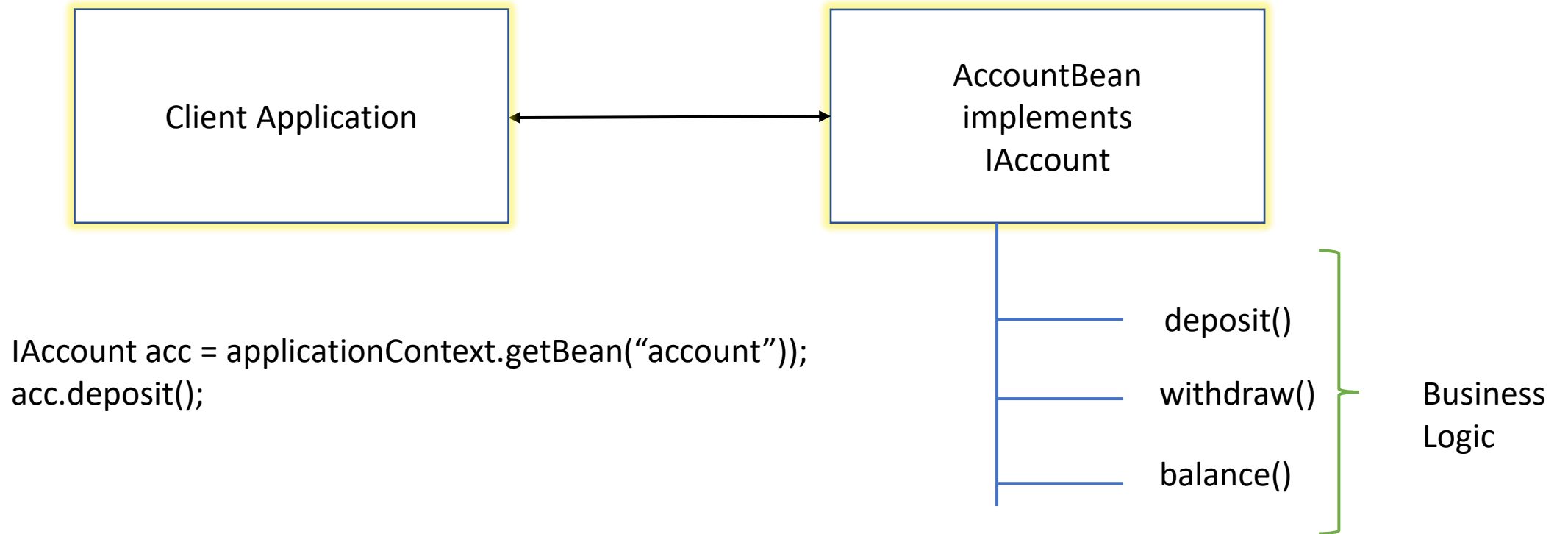
```
@Autowired  
IAccount acc ;  
acc.deposit();
```

Business Tier

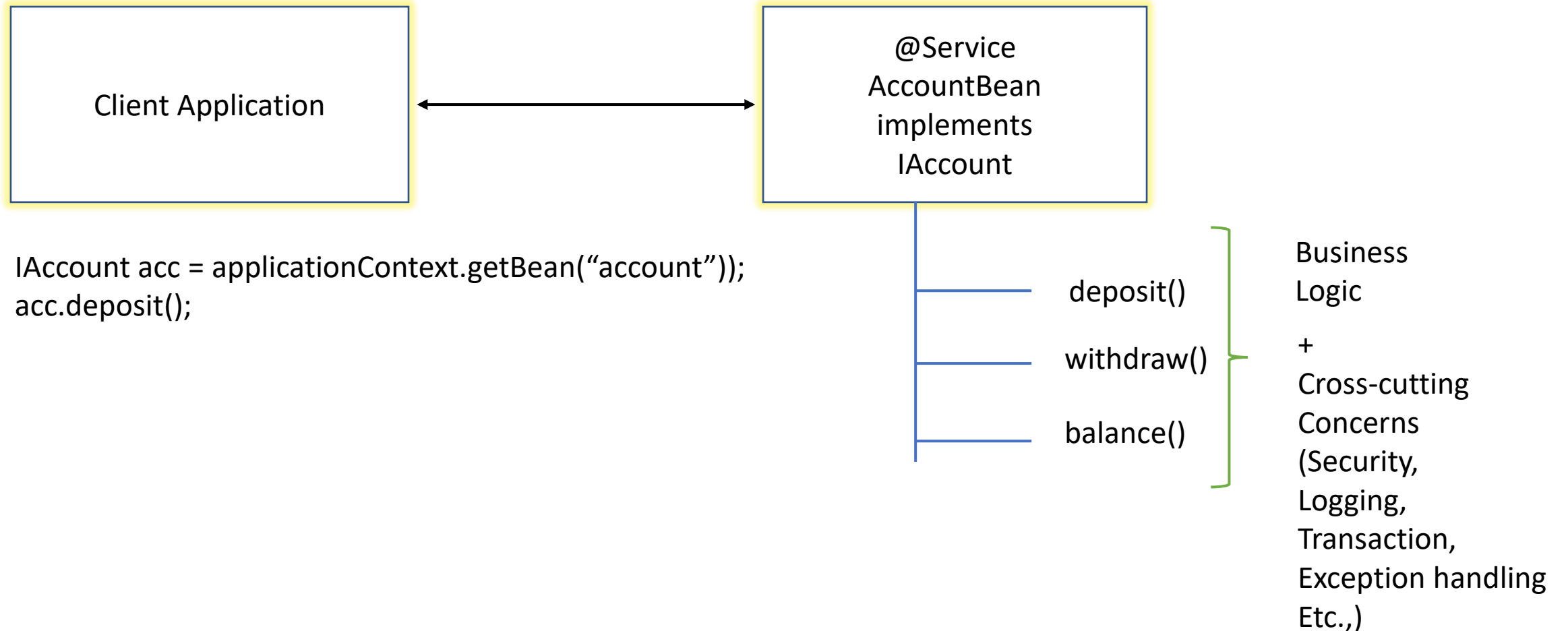
```
Interface IAccount  
{  
    deposit();  
    withdraw();  
    balance();  
}
```

```
@Service  
Class AccountBean  
    implements IAccount  
{  
    deposit() {....}  
    withdraw() {....}  
    balance() {....}  
}
```

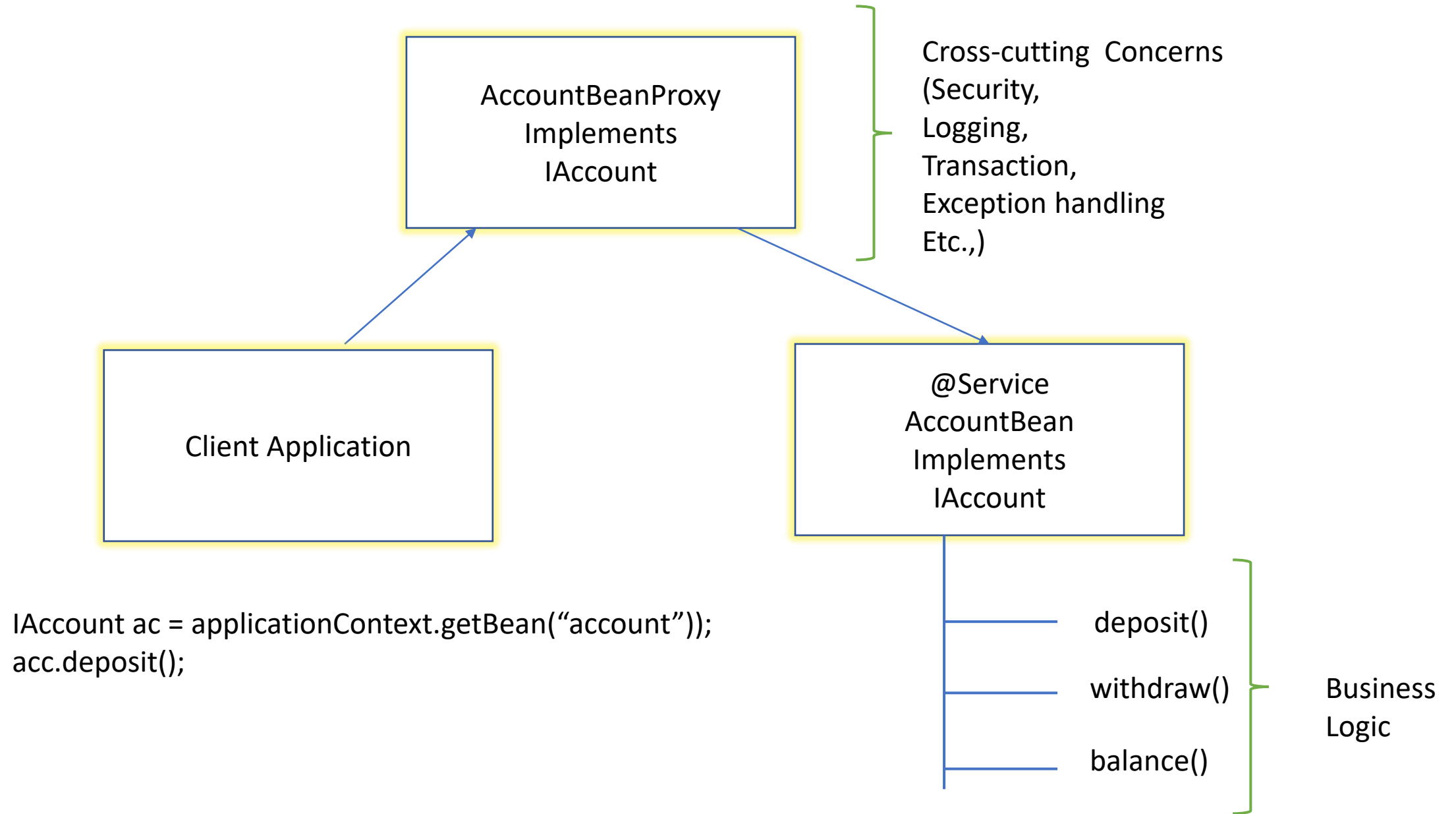
Account Bean with business logic



Account Bean with business & Cross-Cutting Concerns (programmatic)

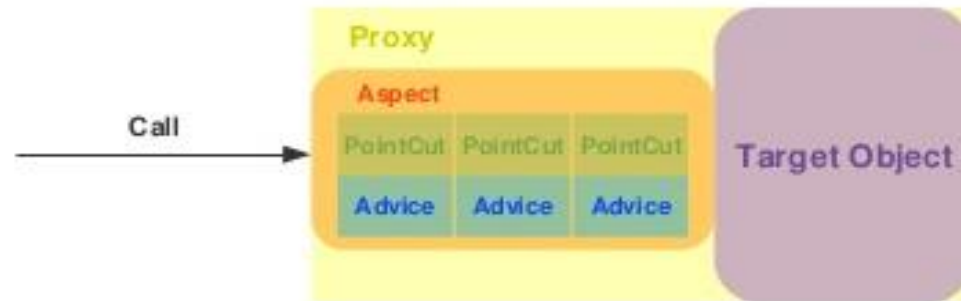


Account Bean with business & Cross-Cutting Concerns (Declarative – using proxy)



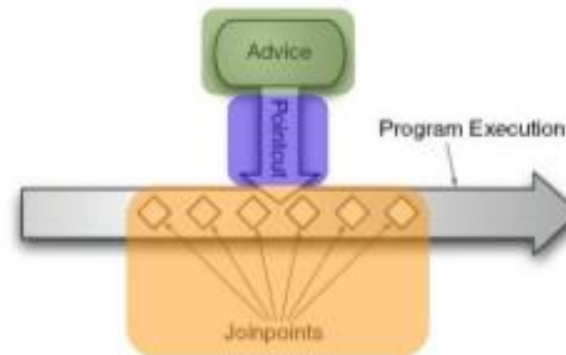
AOP Terminology

- **Aspect** – a modularization of a Crosscutting concern.
- **Target object** - object being advised by one or more aspects.
- **AOP proxy** – Will manage the way of Applying Aspects at particular Pointcuts.



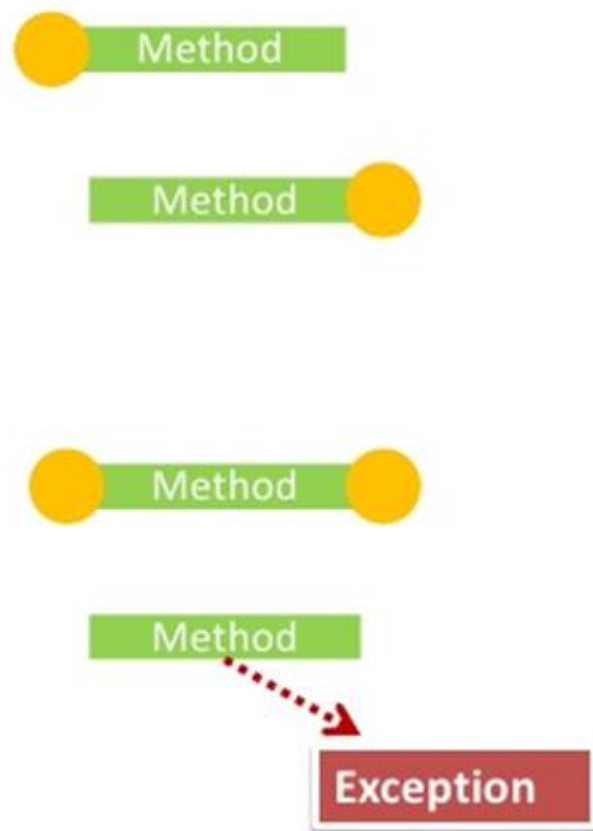
AOP Terminology

- **Joinpoint** – Defines A Point During the Execution of a program. We can insert Additional logics at Joinpoints
- **Advice** – Action taken(Functionality) at a Joinpoint.
- **Pointcut** – Combination of Joinpoints where the Advice need to be Applied.



Advice Types

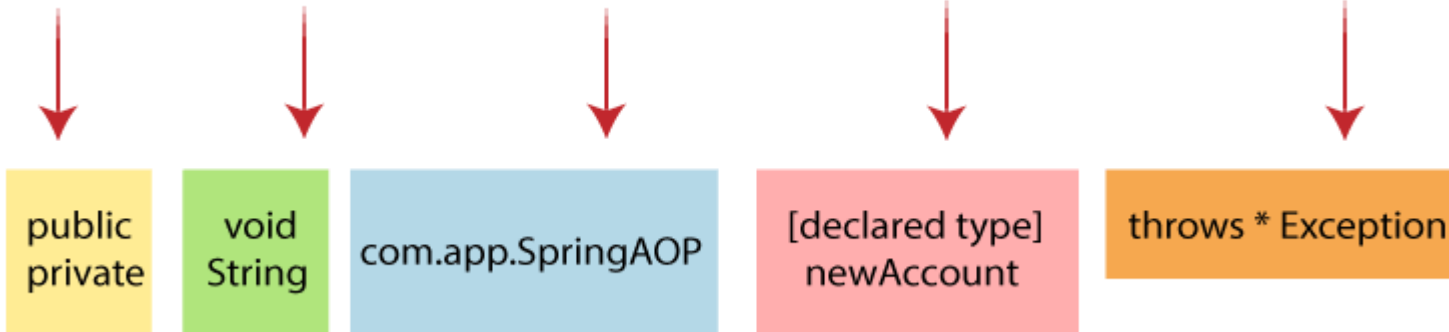
- Before Advice
- After returning Advice
- Around Advice
- Throws Advice



Springs Advice



execution(modifiers-pattern? return-type-pattern declaring-type-pattern?method-name-pattern(param-pattern) throws-pattern?)



Any return type package class method any type and number of arguments

```
@Pointcut("execution(* aspects.trace.demo.*(..))")  
public void traceMethodsInDemoPackage() {}
```

Diagram illustrating the components of the `@Pointcut` annotation and the method signature:

- `Any return type` points to `public`.
- `package` points to `aspects.trace.demo`.
- `class` points to `.*`.
- `method` points to `.*`.
- `any type and number of arguments` points to `(..)`.

Some examples of common pointcut expressions are given below.

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(..))
```

the execution of any method defined in the service package or a sub-package:

```
execution(* com.xyz.service..*.*(..))
```

any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

any join point (method execution only in Spring AOP) within the service package or a sub-package:

```
within(com.xyz.service..*)
```

any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

`this(com.xyz.service.AccountService)`

any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

`target(com.xyz.service.AccountService)`

Spring AOP

Spring AOP

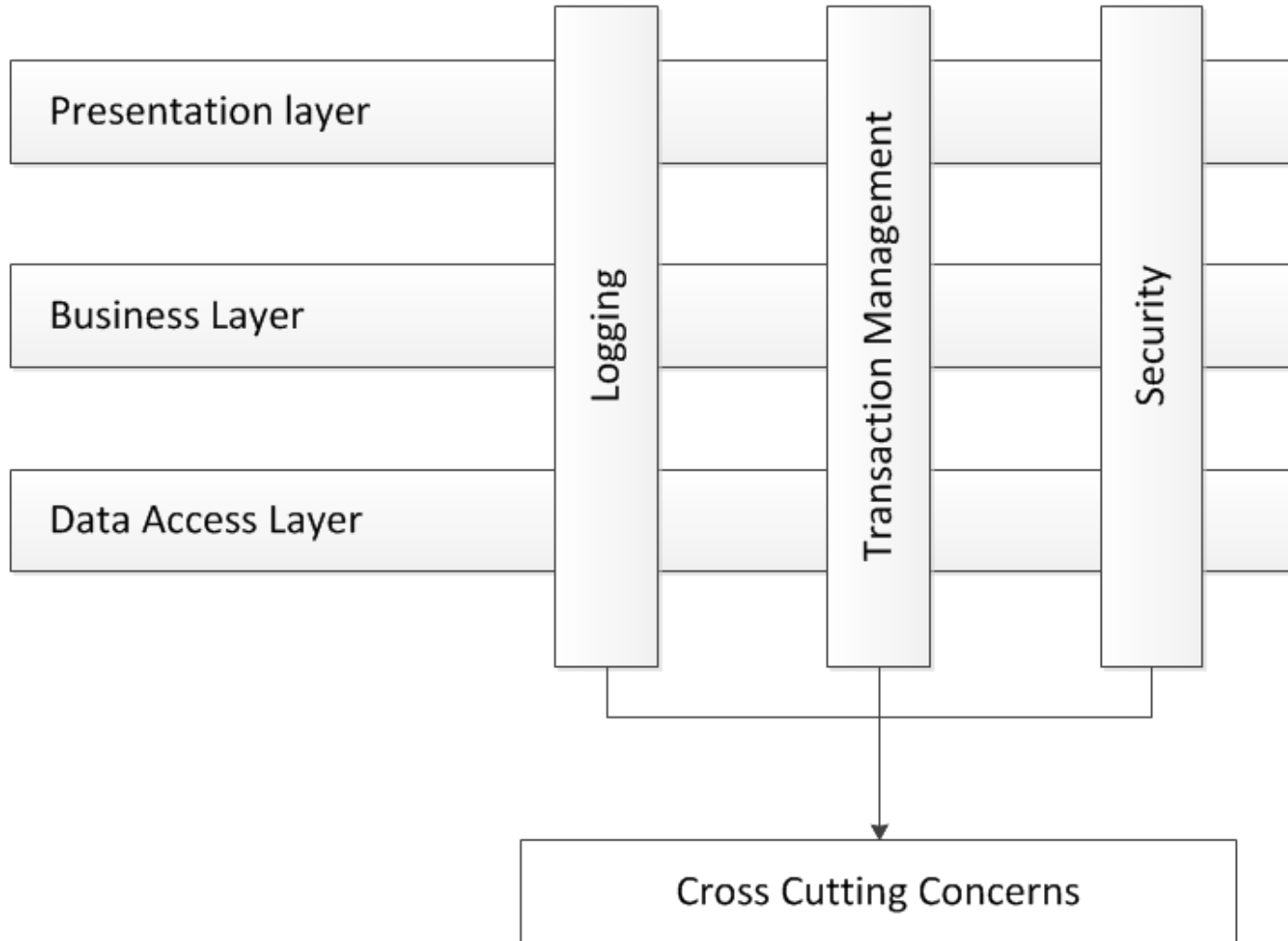
Spring AOP enables Aspect-Oriented Programming in spring applications.

In AOP, aspects enable the modularization of concerns such as :

- ☐ transaction management,
- ☐ Logging,
- ☐ Exception Handling
- ☐ Security etc.,

(often termed crosscutting concerns)

Cross Cutting Concerns



Client Application

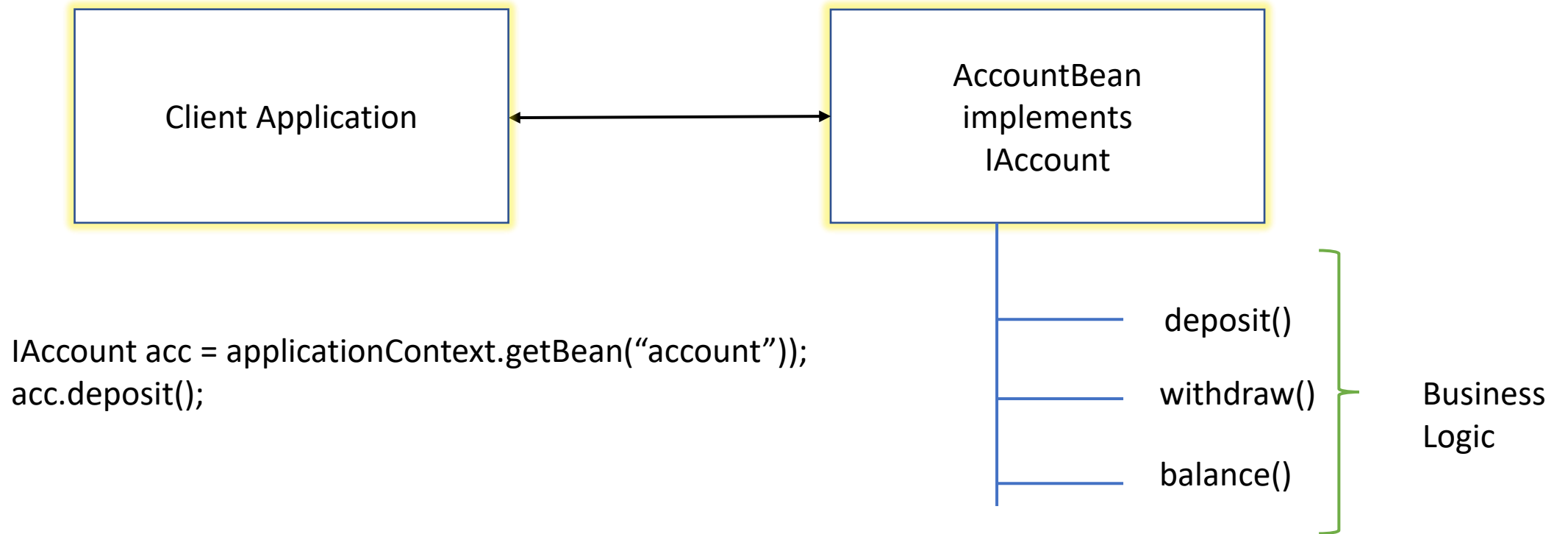
```
@Autowired  
IAccount acc ;  
acc.deposit();
```

Business Tier

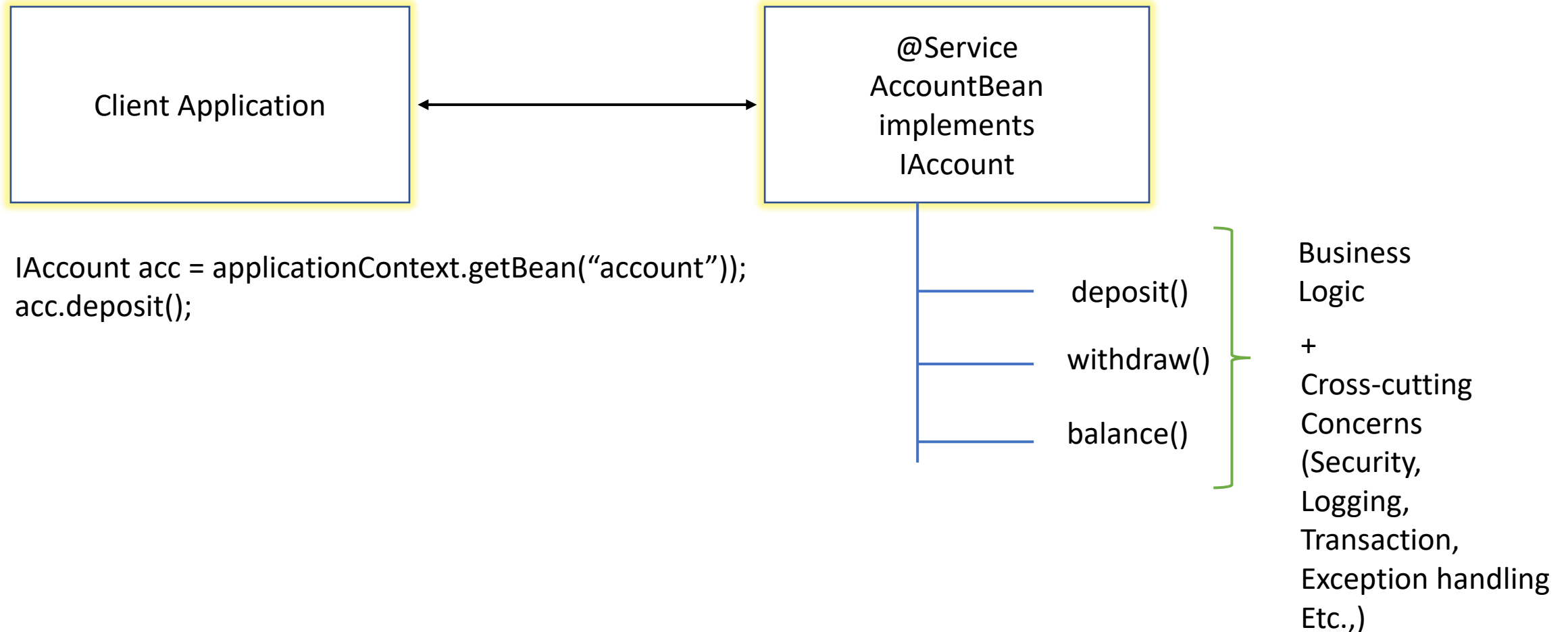
```
Interface IAccount  
{  
    deposit();  
    withdraw();  
    balance();  
}
```

```
@Service  
Class AccountBean  
    implements IAccount  
{  
    deposit() {....}  
    withdraw() {....}  
    balance() {....}  
}
```

Account Bean with business logic



Account Bean with business & Cross-Cutting Concerns (programmatic)



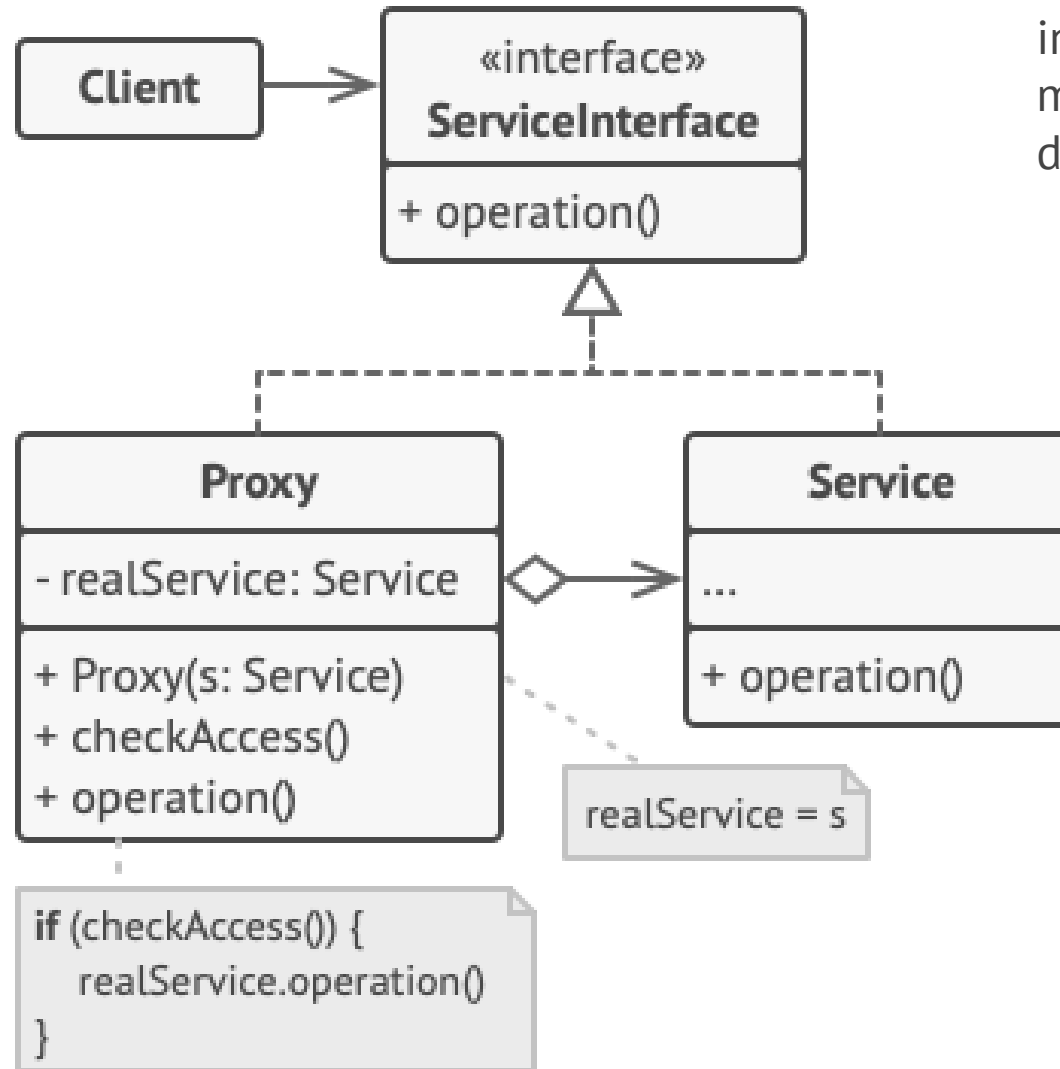
GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
SCOPE	Class	Factory Method	Class Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

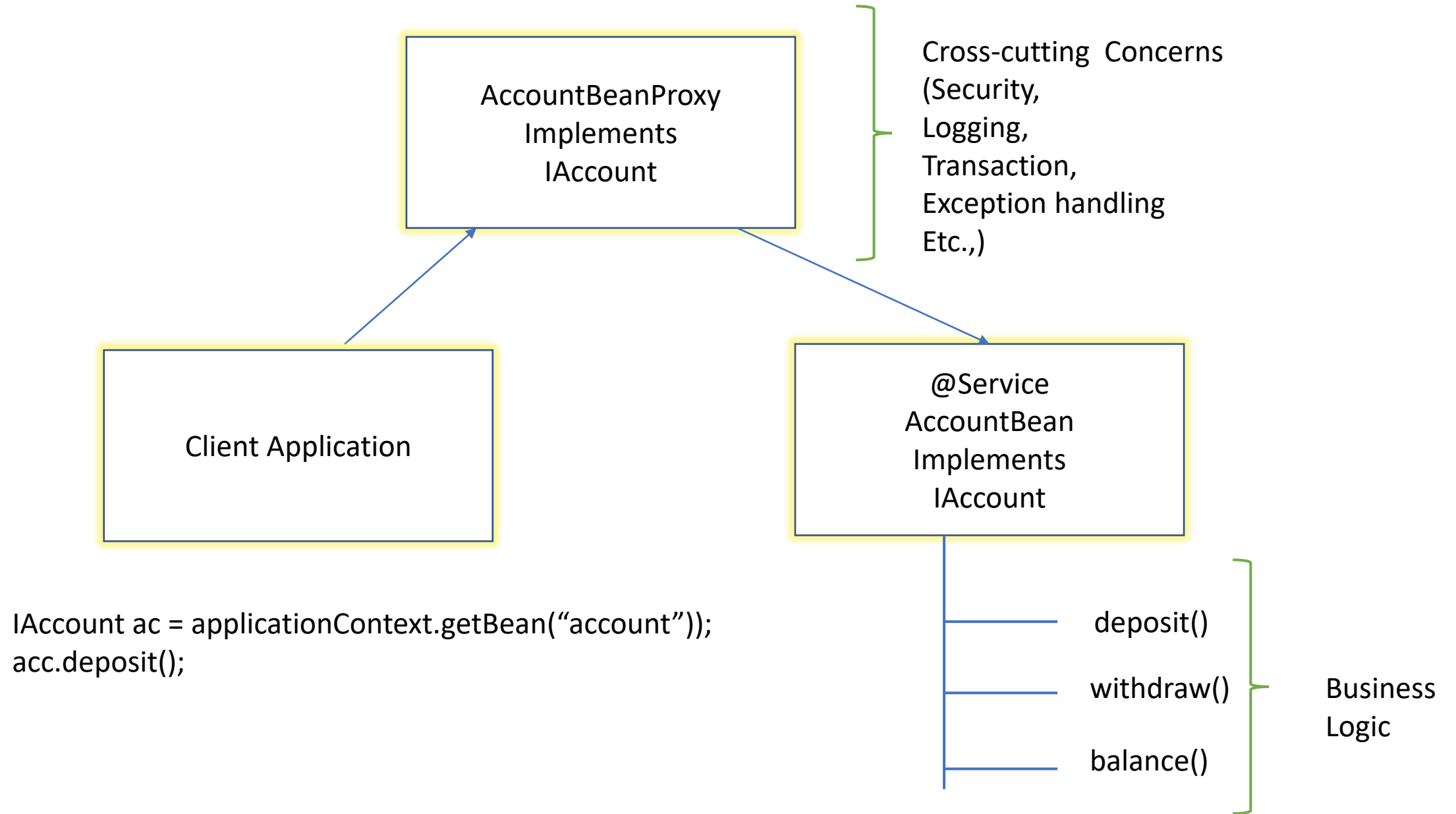
The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object. Usually, proxies manage the full lifecycle of their service objects.



The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

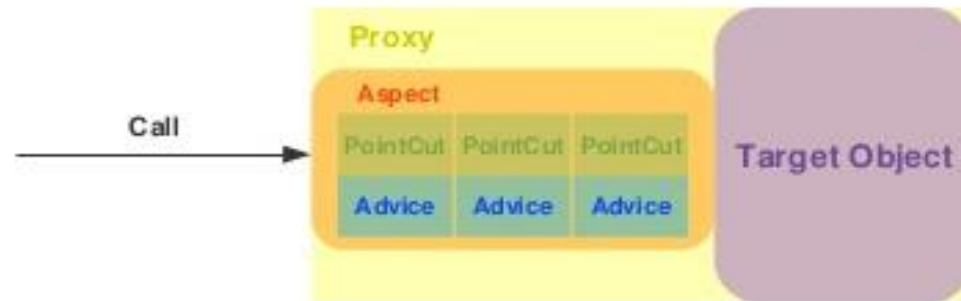
The **Service** is a class that provides some useful business logic.

Account Bean with business & Cross-Cutting Concerns (Declarative – using proxy)



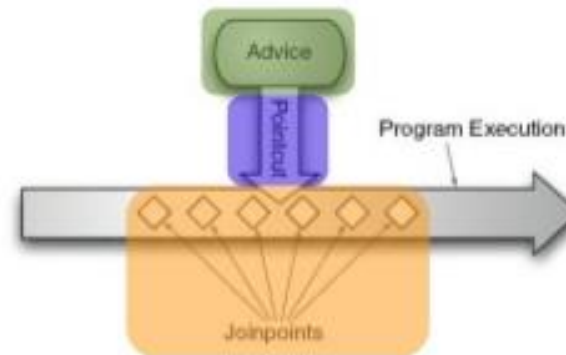
AOP Terminology

- **Aspect** – a modularization of a Crosscutting concern.
- **Target object** - object being advised by one or more aspects.
- **AOP proxy** – Will manage the way of Applying Aspects at particular Pointcuts.



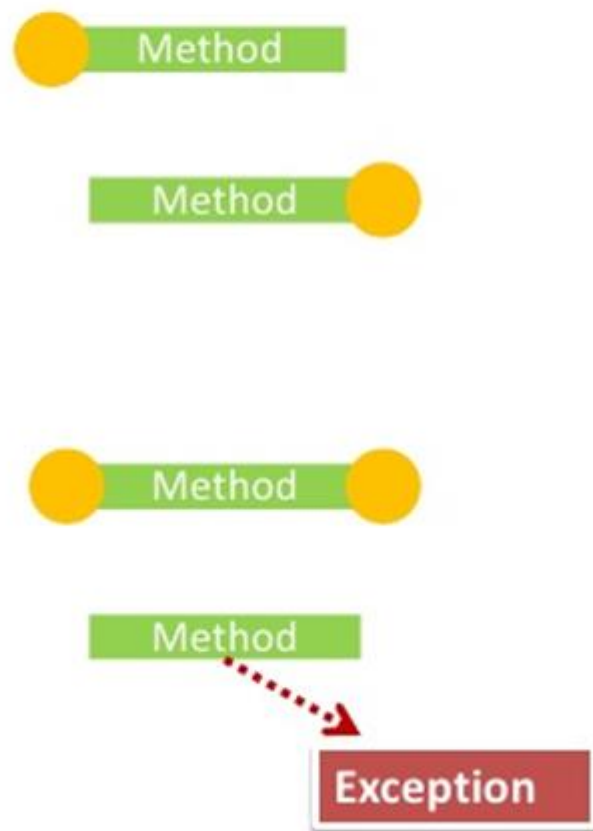
AOP Terminology

- **Joinpoint** – Defines A Point During the Execution of a program. We can insert Additional logics at Joinpoints
- **Advice** – Action taken(Functionality) at a Joinpoint.
- **Pointcut** – Combination of Joinpoints where the Advice need to be Applied.



Advice Types

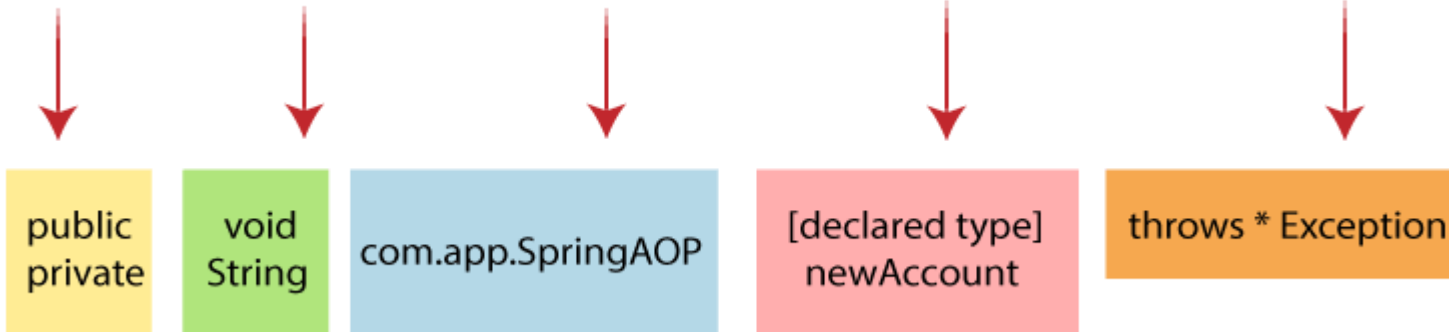
- Before Advice
- After returning Advice
- Around Advice
- Throws Advice



Springs Advice



execution(modifiers-pattern? return-type-pattern declaring-type-pattern?method-name-pattern(param-pattern) throws-pattern?)



Any return type

package

class

method

any type and number of arguments

```
@Pointcut("execution(* aspects.trace.demo.*(..))")  
public void traceMethodsInDemoPackage() {}
```

Some examples of common pointcut expressions are given below.

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(..))
```

the execution of any method defined in the service package or a sub-package:

```
execution(* com.xyz.service..*.*(..))
```

any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

any join point (method execution only in Spring AOP) within the service package or a sub-package:

```
within(com.xyz.service..*)
```

any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

`this(com.xyz.service.AccountService)`

any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

`target(com.xyz.service.AccountService)`

@ConditionalOnClass and @ConditionalOnMissingClass:

Scenario: We want to configure a bean that depends on the availability or absence of a specific library or class.

Example: If your application uses an external caching library like Redis, you can conditionally enable a caching bean only if the Redis library is present on the classpath (@ConditionalOnClass(Redis.class)).

Conversely, we can conditionally enable an alternative caching strategy if Redis is not available(@ConditionalOnMissingClass("org.springframework.data.redis.core.RedisTemplate")).

@ConditionalOnBean and @ConditionalOnMissingBean:

Scenario: We want to configure beans based on the presence or absence of other beans in the application context.

Example: If we have multiple implementations of a data repository interface, we can conditionally configure a service bean to use a specific repository implementation only if the corresponding repository bean is present (@ConditionalOnBean(UserRepository.class)).

Alternatively, we can conditionally configure a fallback implementation if the repository bean is missing (@ConditionalOnMissingBean(UserRepository.class)).

@ConditionalOnProperty:

Scenario: We want to conditionally configure a bean based on the values of application properties.

Example: Suppose we have an email notification service, and you want to enable or disable it based on a configuration property. We can use @ConditionalOnProperty to conditionally configure the email service bean only if a specific property is set to a certain value.

@ConditionalOnResource:

Scenario: We want to conditionally configure a bean based on the presence of a specific resource on the classpath.

Example: Let's say we have a module that provides additional features, and you want to conditionally enable it only if a specific configuration file (my-module-config.properties) is present on the classpath. we can use @ConditionalOnResource to ensure the module's beans are configured only when the resource is available.

@ConditionalOnWebApplication and
@ConditionalOnNotWebApplication:

Scenario: We want to configure beans specifically for web or non-web environments.

Example: Suppose we have a bean that provides web-specific functionality, such as an API documentation generator. We can use @ConditionalOnWebApplication to configure the bean only if the application is running as a web application.

Conversely, we might have a bean that should be created only for non-web environments, such as a background job scheduler, which can be configured using @ConditionalOnNotWebApplication.