Spring Basics

Philippe Collet

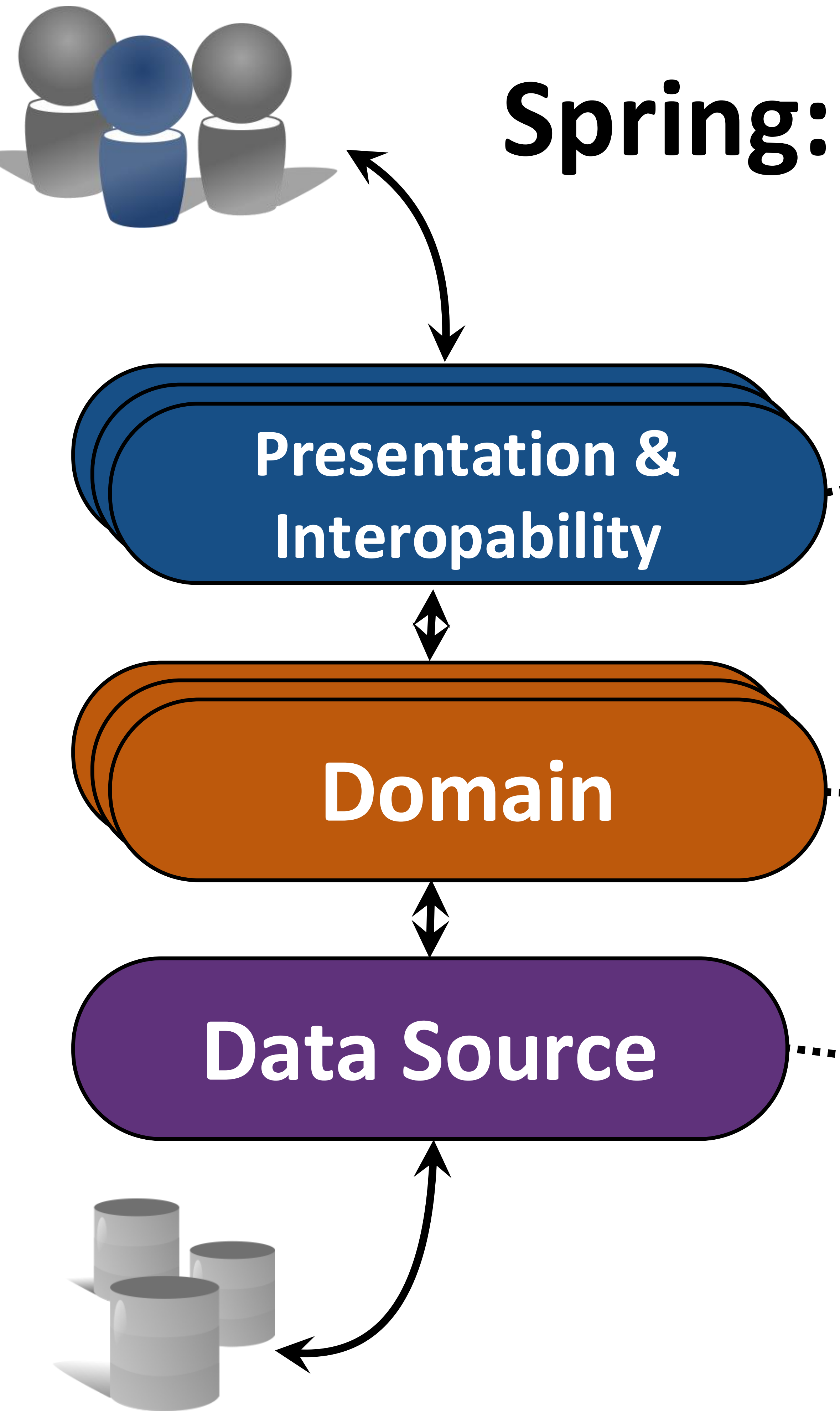
pictures: sxc.hu, pixabay

Spring History

–August 2003	Spring 1.0 M1
–October 2006	Spring 2.0
–December 2009	Spring 3.0
–December 2013	Spring 4.0
–June 2014	**SpringBoot 1.1**
–February 2017	**SpringBoot 1.5**
–October 2017	Spring 5 (reactive streams)
–November 2017	SpringBoot 2.0 (Spring 5 + JDK9)
–November 2022	SpringBoot 3.0 (Spring 6 + JDK17)
–**May 2025**	**SpringBoot 3.5**
–December 2025	SpringBoot 4.0 (Spring 7)

LATEST stable SpringBoot we are using : 3.5.9

Spring: a N-tiers architecture



Handle users & interop
(Several layers possible)

Business functionality
(Several layers possible)

Handle data storage
(1 layer + objects)

Spring N-tiers architecture (and Benefits)

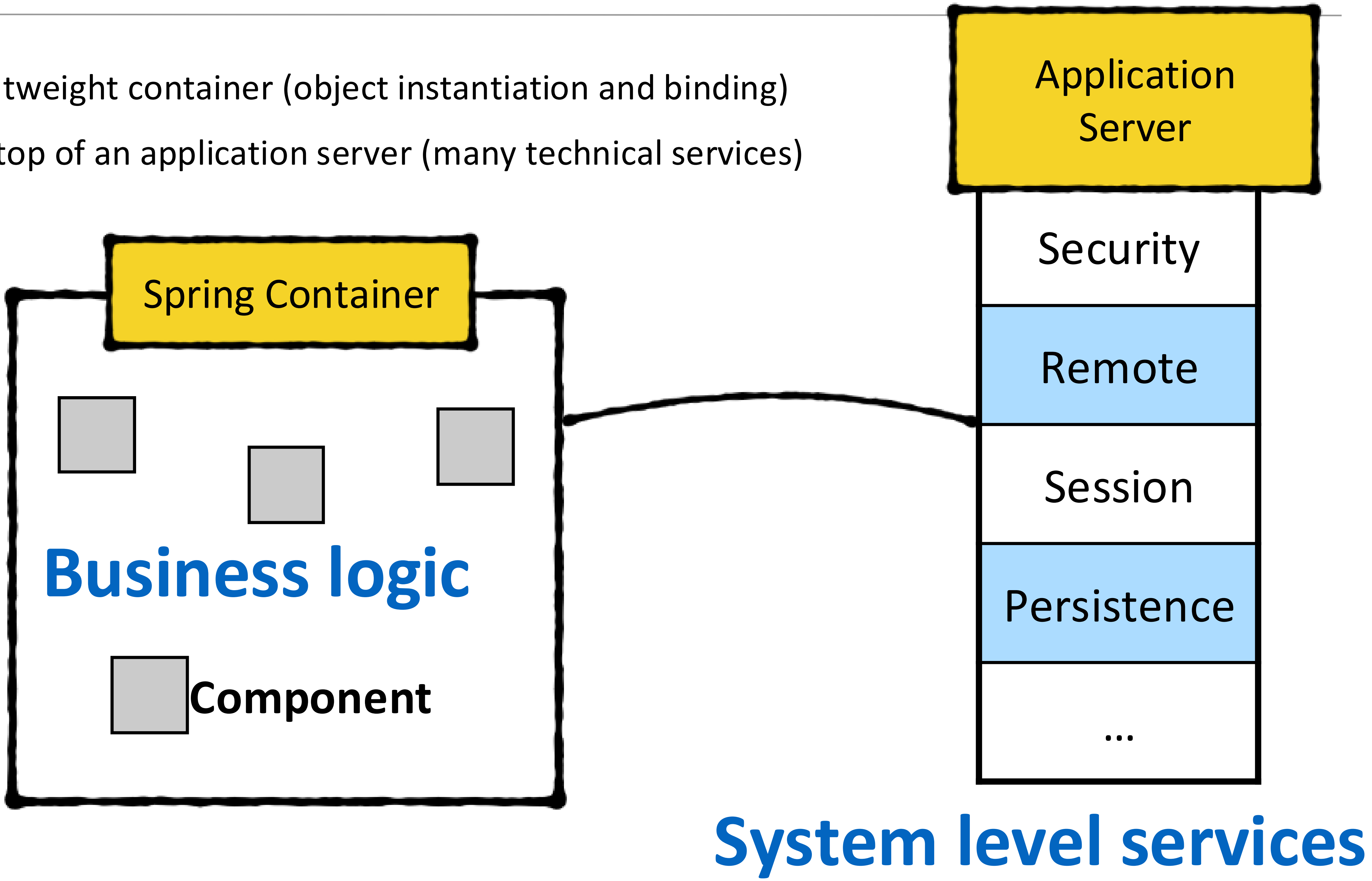
3-tiers architecture

Benefits

- Modularity (at the layer level)
- Separation of concerns (between layers)
- Extensibility / Replaceability
- Optimization (cache between layers)

Spring Principles (and Benefits)

- Lightweight container (object instantiation and binding)
- On top of an application server (many technical services)



Spring Principles (and Benefits)

Lightweight container (object instantiation and binding)

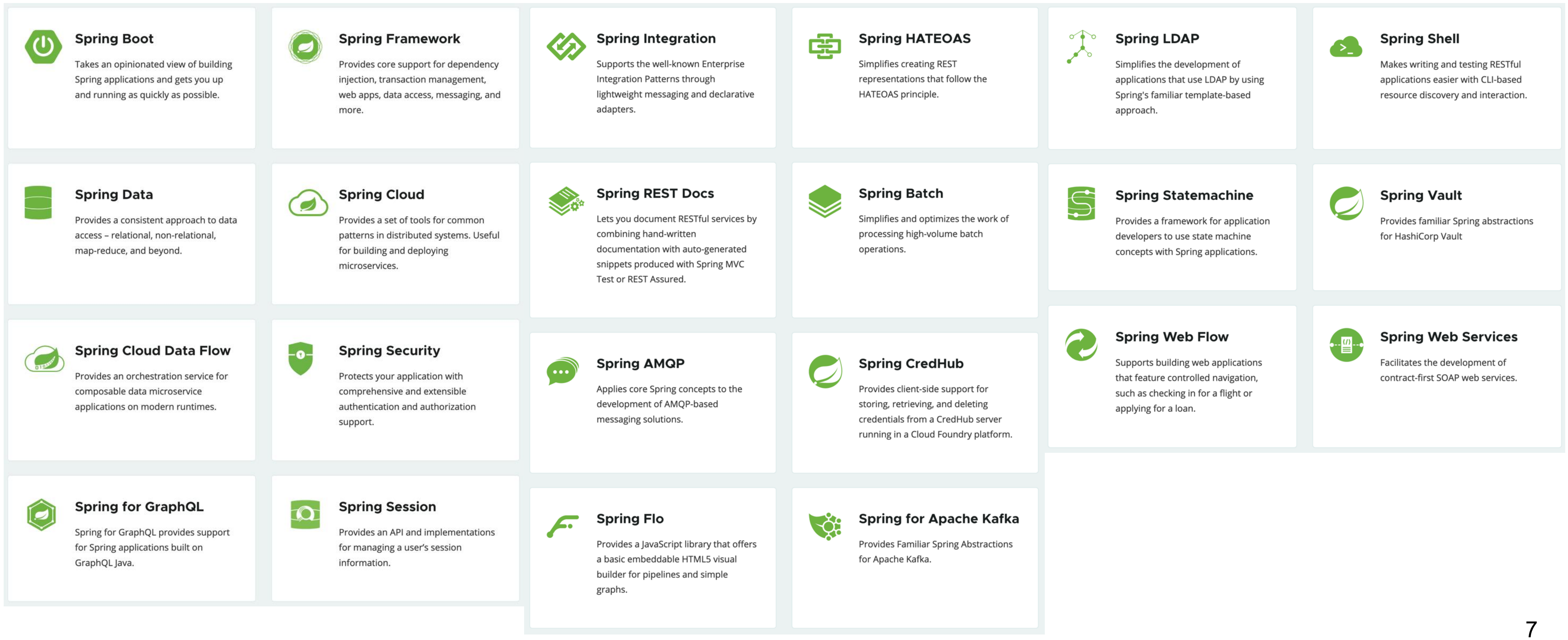
On top of an application server (many technical services)

Benefits

- Modularity (business-oriented components)
- Separation of concerns (between business and technical services)
- Reuse (of interfaces)

Spring Principles (and Benefits)

- A range of services to integrate Web frameworks, persistence frameworks and many librairies or external systems



Spring Boot
Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.

Spring Framework
Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

Spring Integration
Supports the well-known Enterprise Integration Patterns through lightweight messaging and declarative adapters.

Spring HATEOAS
Simplifies creating REST representations that follow the HATEOAS principle.

Spring LDAP
Simplifies the development of applications that use LDAP by using Spring's familiar template-based approach.

Spring Shell
Makes writing and testing RESTful applications easier with CLI-based resource discovery and interaction.

Spring Data
Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.

Spring Cloud
Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

Spring REST Docs
Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.

Spring Batch
Simplifies and optimizes the work of processing high-volume batch operations.

Spring Statemachine
Provides a framework for application developers to use state machine concepts with Spring applications.

Spring Vault
Provides familiar Spring abstractions for HashiCorp Vault

Spring Cloud Data Flow
Provides an orchestration service for composable data microservice applications on modern runtimes.

Spring Security
Protects your application with comprehensive and extensible authentication and authorization support.

Spring AMQP
Applies core Spring concepts to the development of AMQP-based messaging solutions.

Spring CredHub
Provides client-side support for storing, retrieving, and deleting credentials from a CredHub server running in a Cloud Foundry platform.

Spring Web Flow
Supports building web applications that feature controlled navigation, such as checking in for a flight or applying for a loan.

Spring Web Services
Facilitates the development of contract-first SOAP web services.

Spring for GraphQL
Spring for GraphQL provides support for Spring applications built on GraphQL Java.

Spring Session
Provides an API and implementations for managing a user's session information.

Spring Flo
Provides a JavaScript library that offers a basic embeddable HTML5 visual builder for pipelines and simple graphs.

Spring for Apache Kafka
Provides Familiar Spring Abstractions for Apache Kafka.

Spring Principles (and Benefits)

A range of services to integrate Web frameworks, persistence frameworks and many librairies or external systems

Benefits

- Modularity (between technical libraries)
- Separation of concerns (between business and technical services)
- Reuse (of technical services)

Spring Components | 101

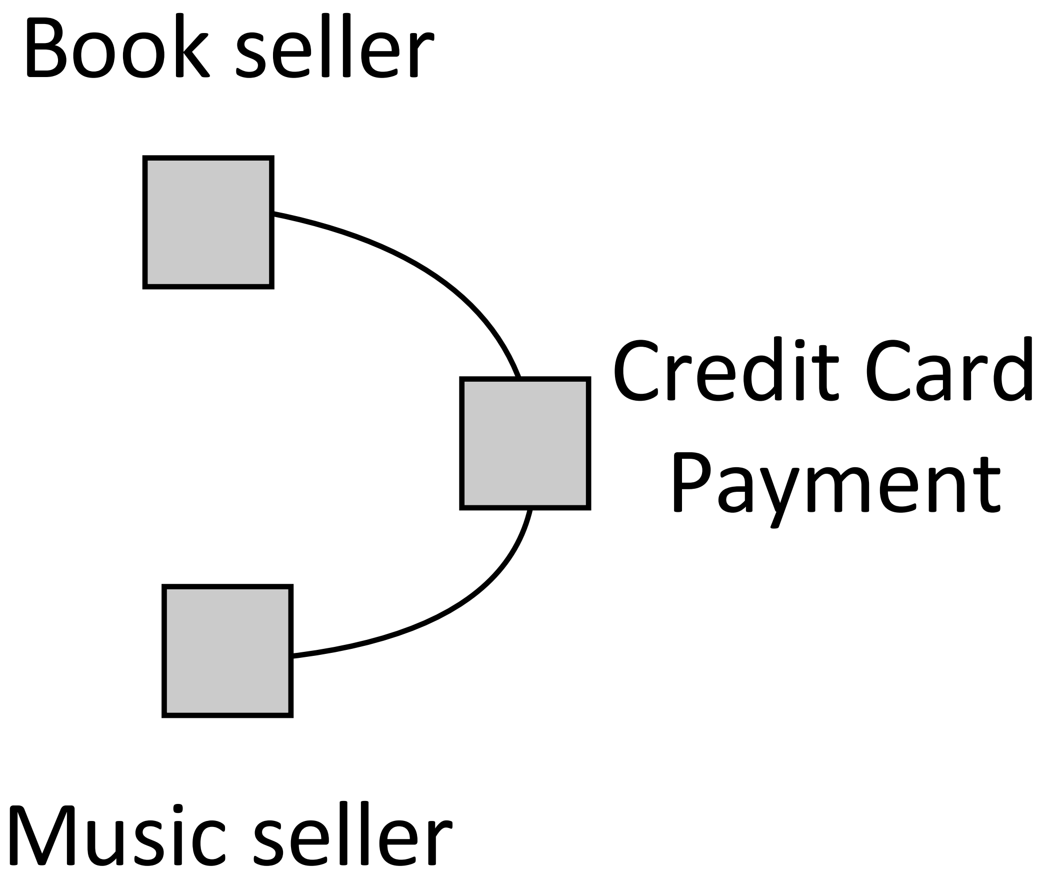
Spring **Components**

Interface **reuse**

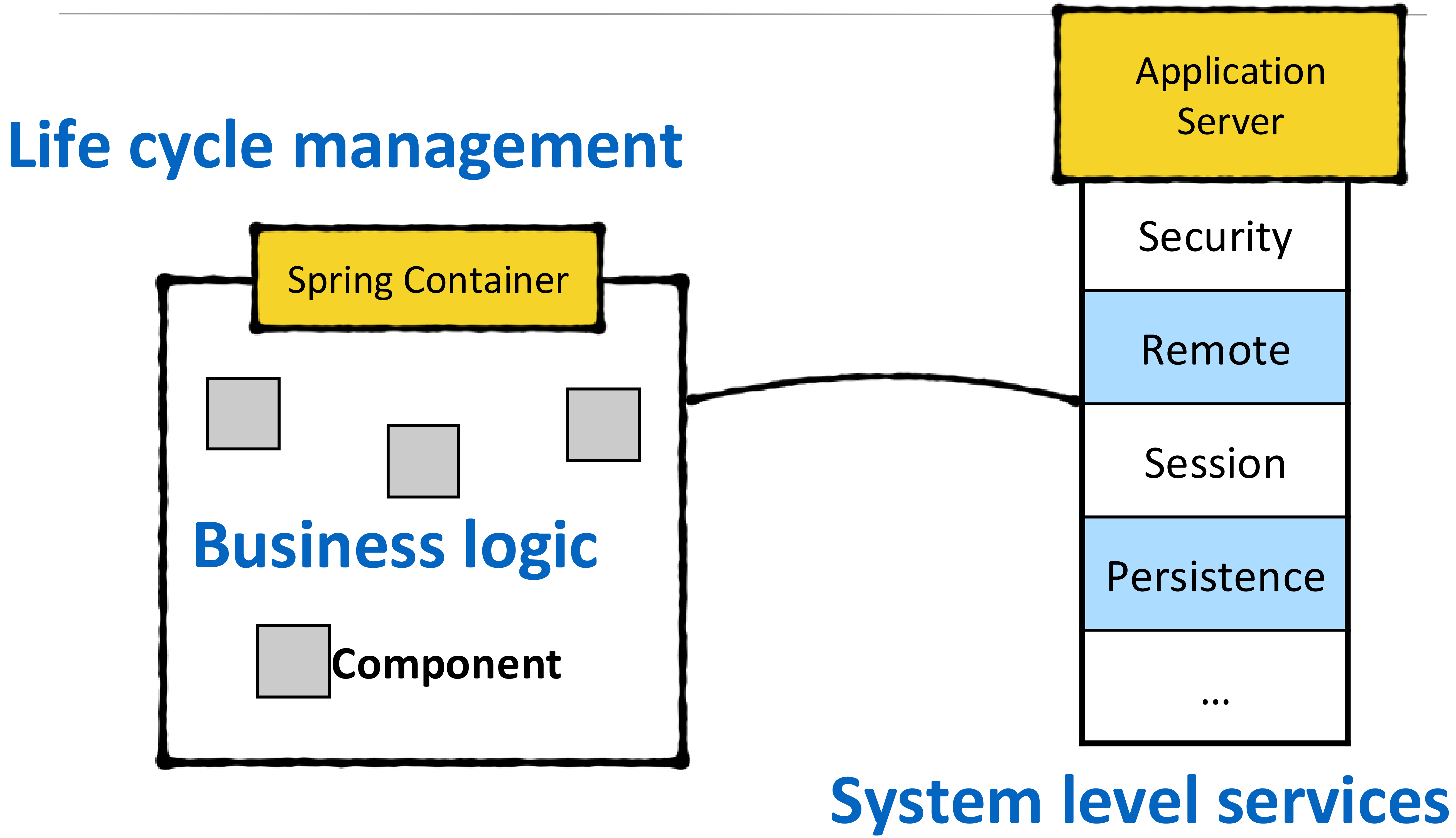
Encapsulate

application

Behavior

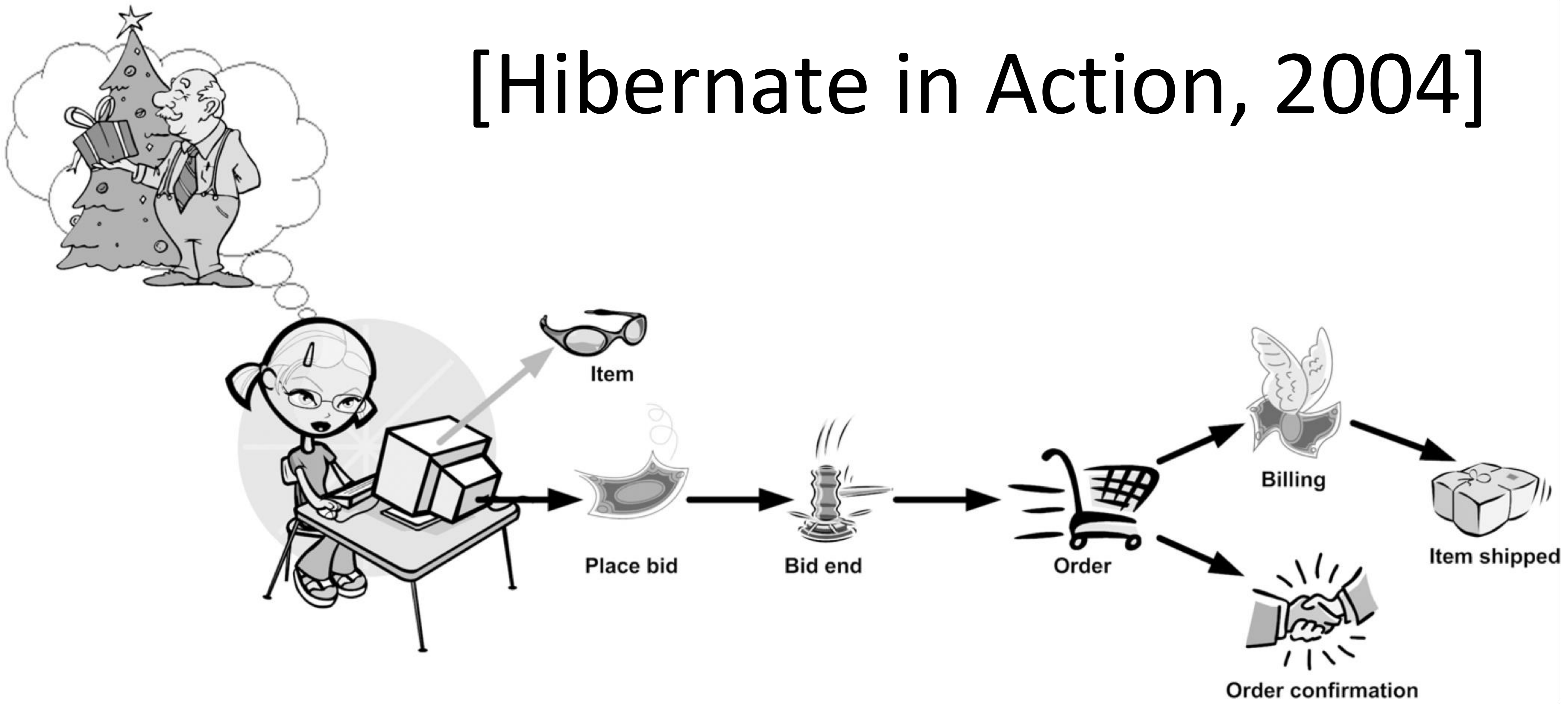


Spring infrastructure



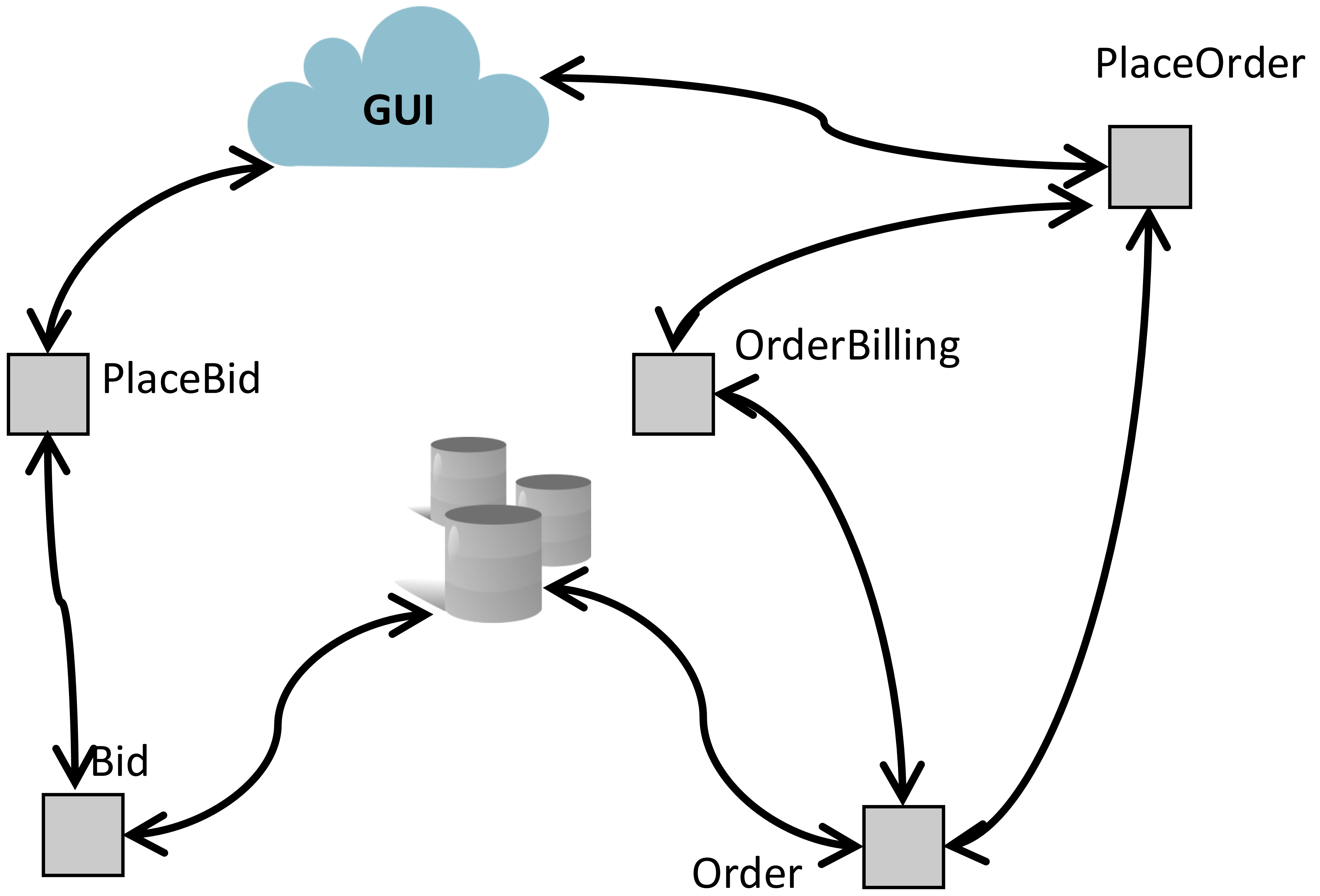
The ActionBazaar example

[Hibernate in Action, 2004]

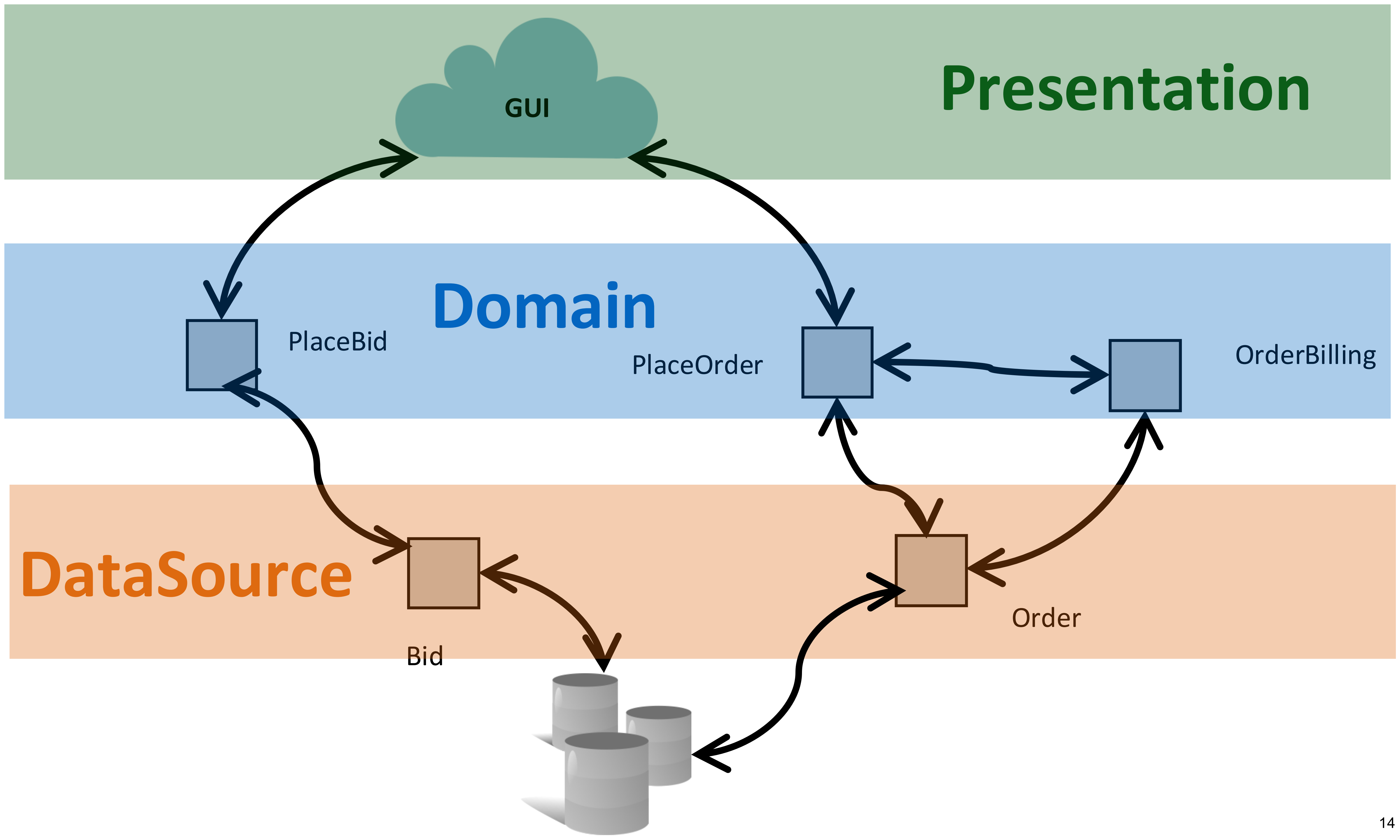


[EiA]

Identifying **Components & Objects**



3-tiers Architecture



Rule of Thumb

Domain Beans interfaces as **Verbs**

DataSource Beans as **Nouns**

What is a **POJO**?

An ordinary Java Object

Not bound to any restriction such as

Extending some classes

Implementing some interfaces

Containing some annotations

What is a **Bean**?

A POJO that

is **serializable**

has **no-argument constructor**

allows access to properties using **getters / setters** with a **simple naming convention**

What is a **Spring Component** ?

POJO

+

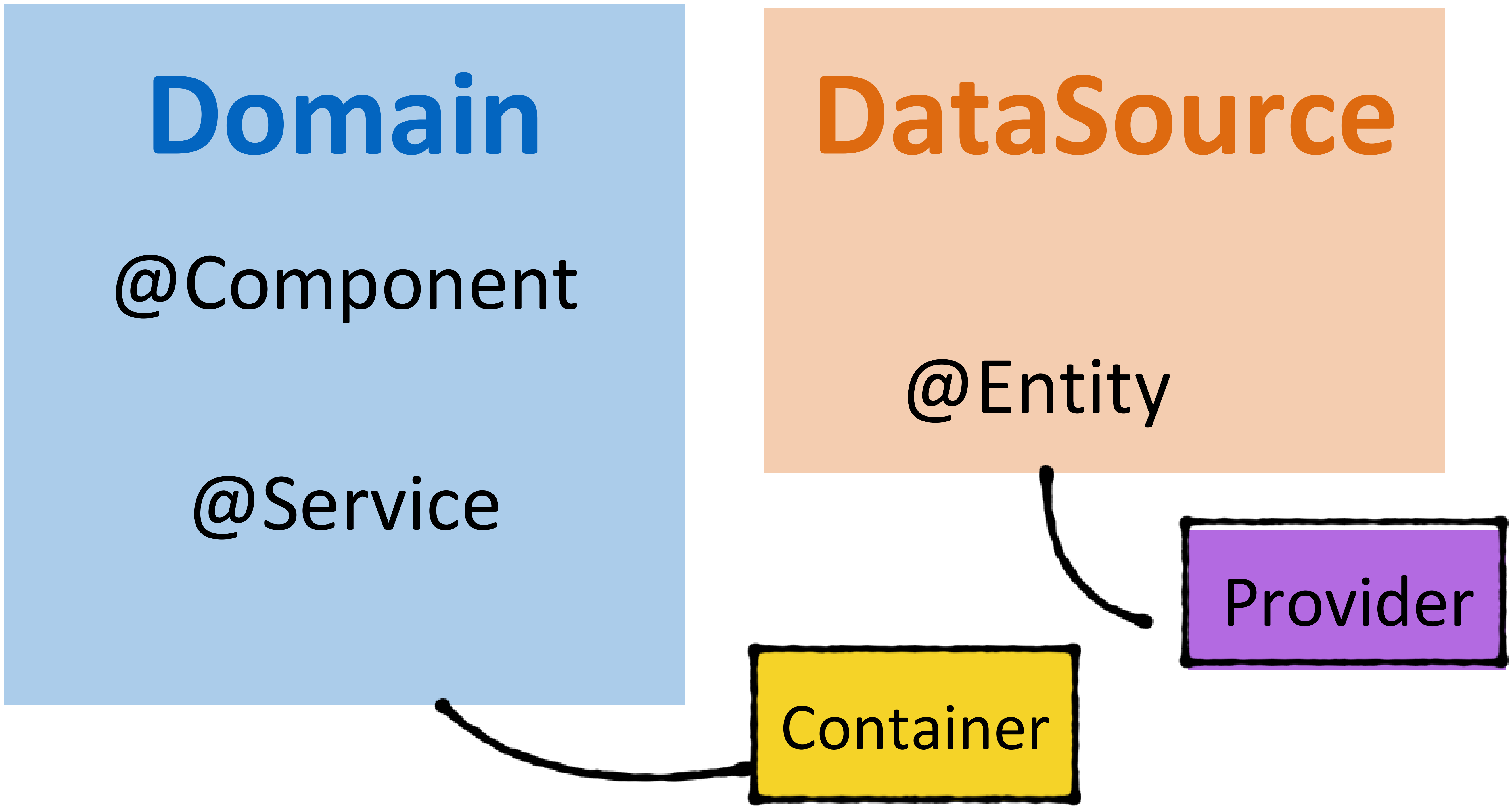
@nnotations **Méta-données de configuration**

Spring Component

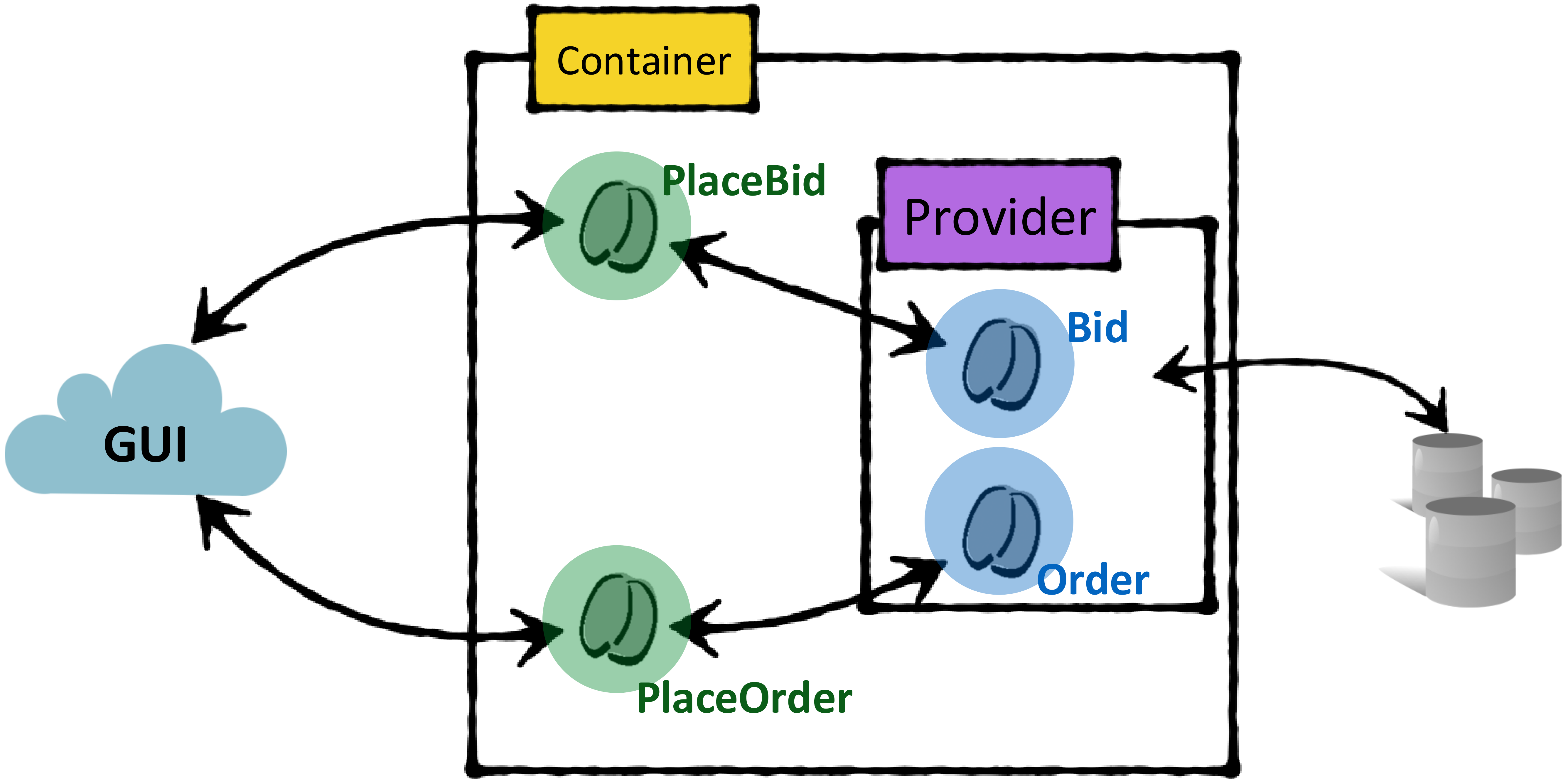
```
public interface HelloUser {
   public void sayHello(String n);
}
```

```
@Component
public class HelloUserBean implements HelloUser{
  public void sayHello(String n) {
    System.out.println("Hello, " + n "!");
  }
}
```

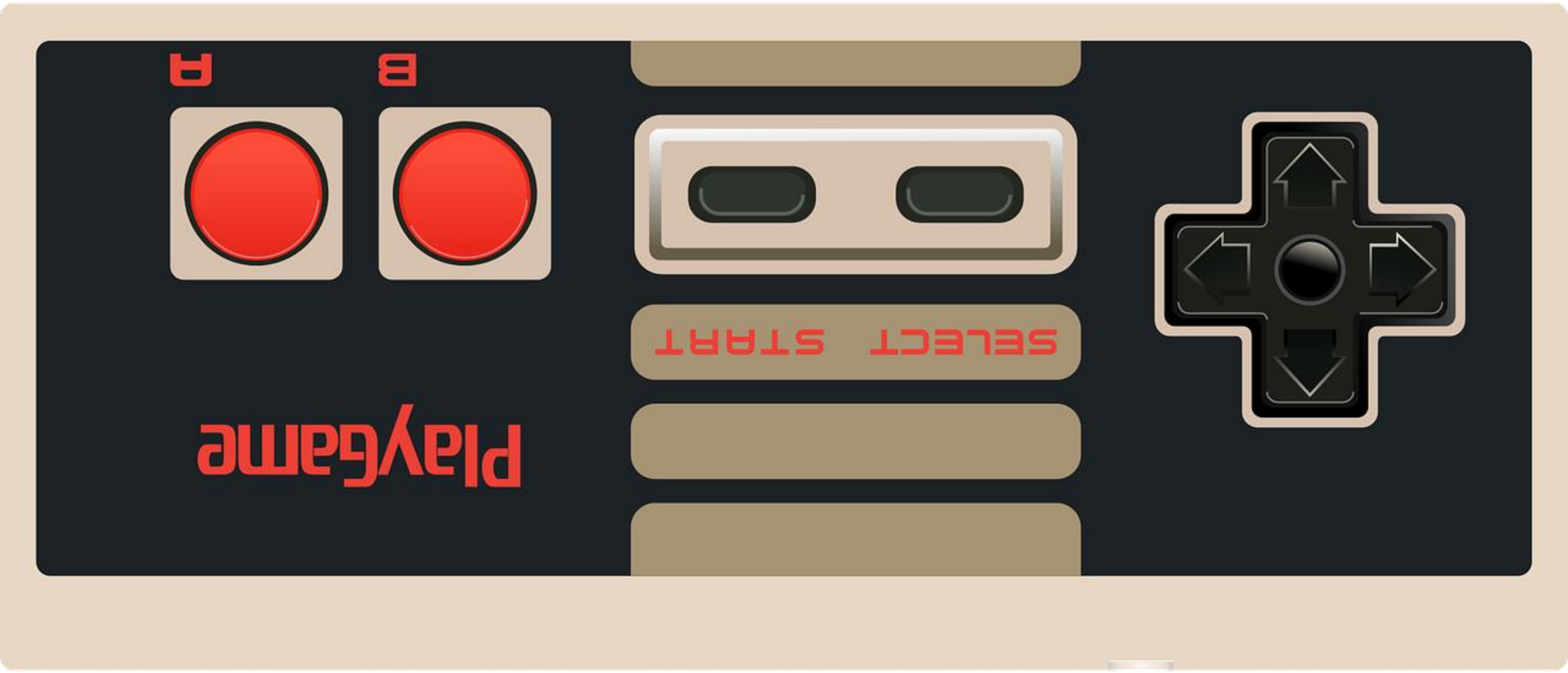
Different **Flavors**



Components and entities

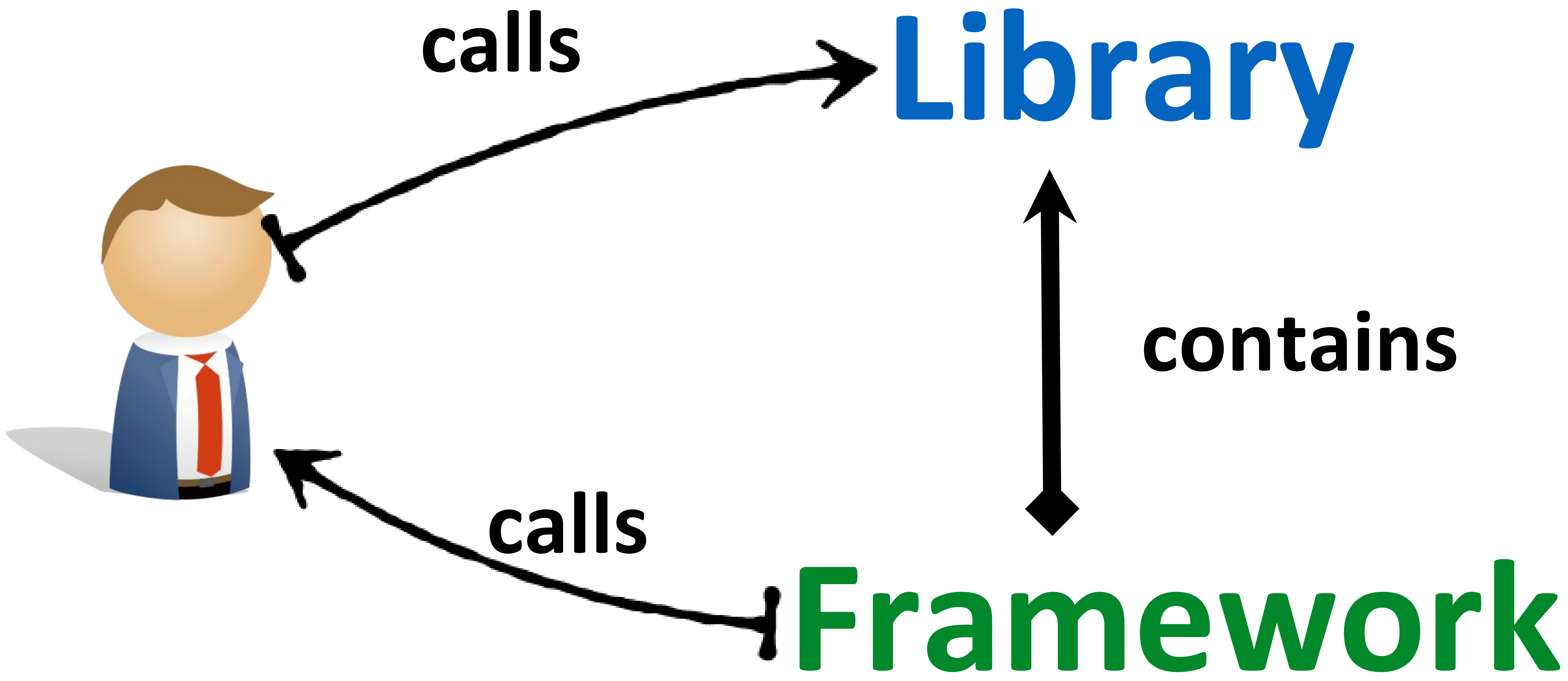


[EiA]



Inversion of Control 101

Library versus **Framework**?



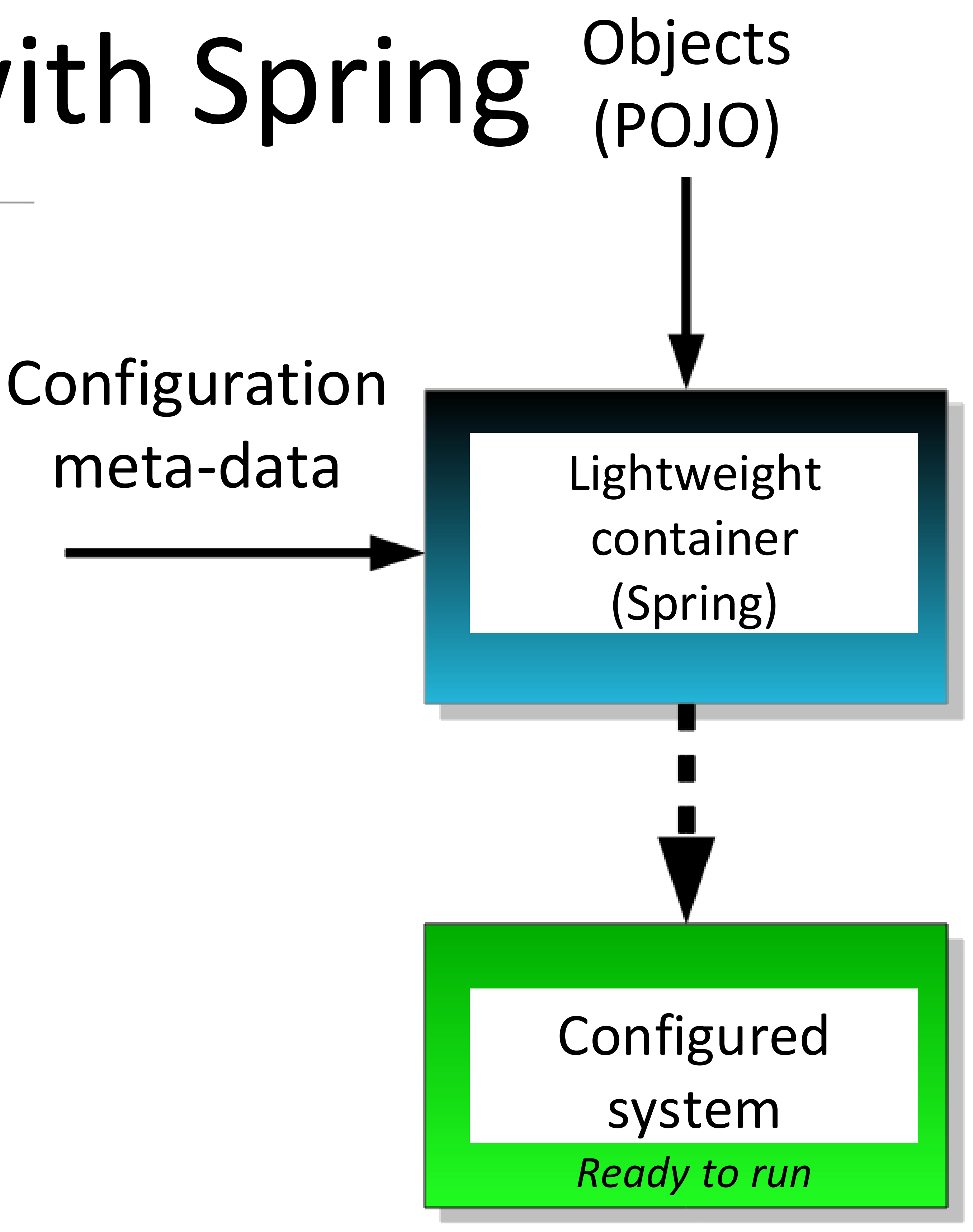
Inversion of Control

aka the Hollywood pattern

Your **code** reuses a **library**

A **framework** reuses your **code**

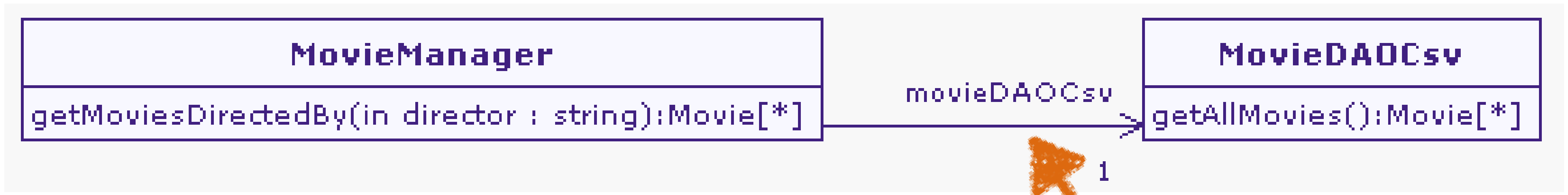
IoC with Spring



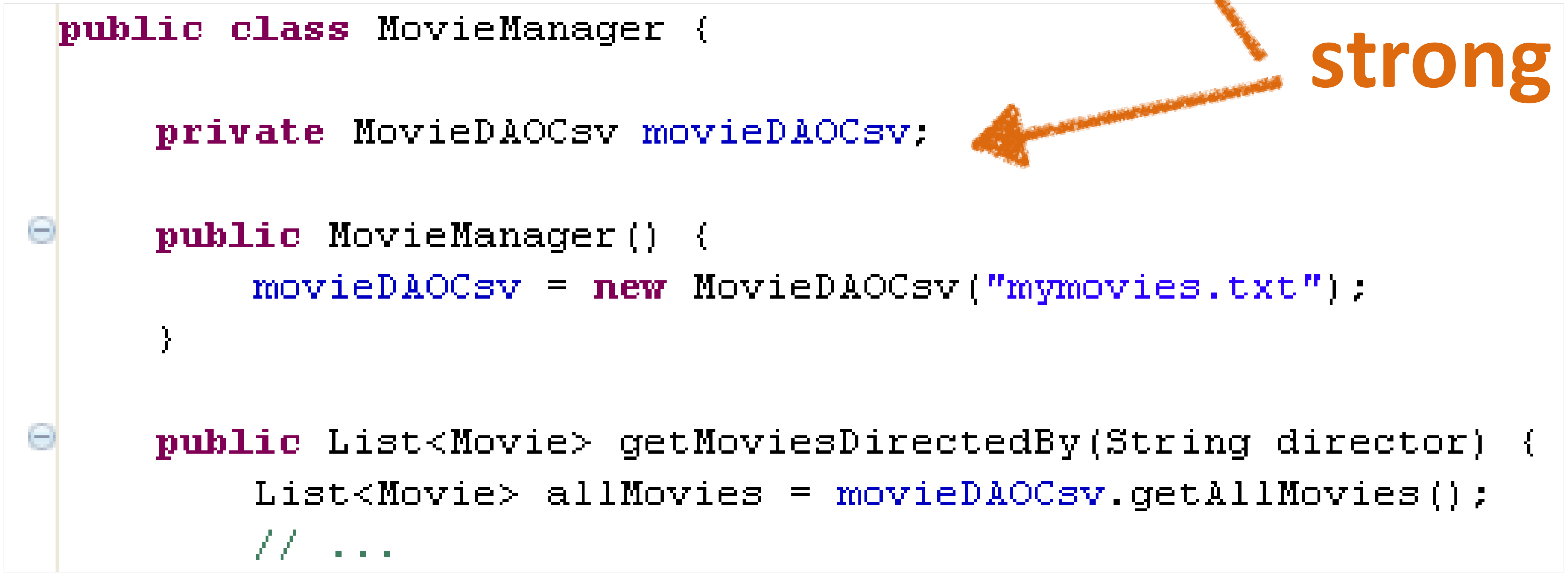
Dependency Injection

Illustration

Based in part on Martin Fowler's reference article on dependency injection

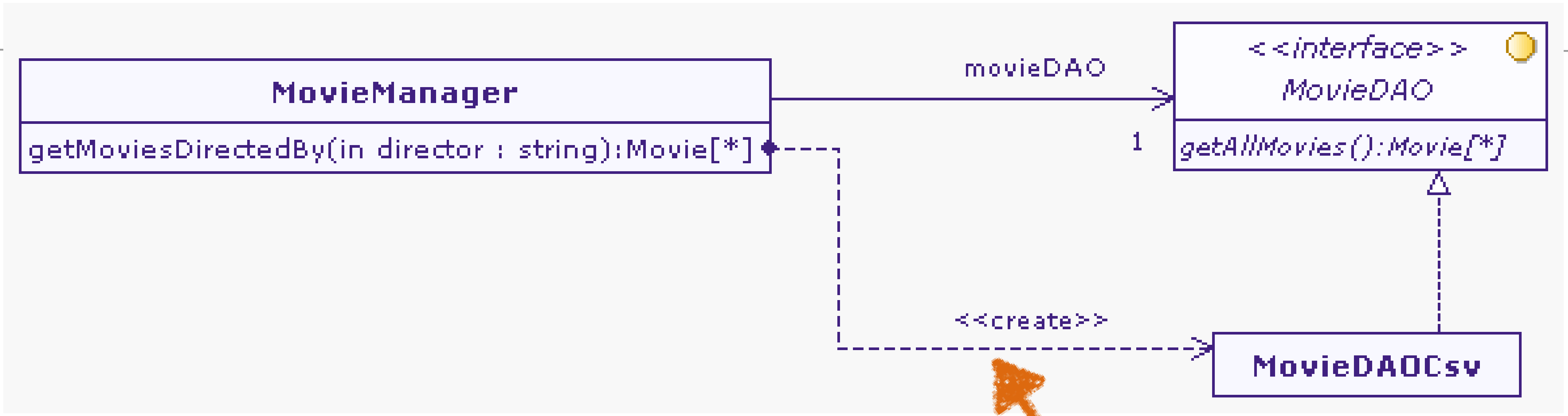


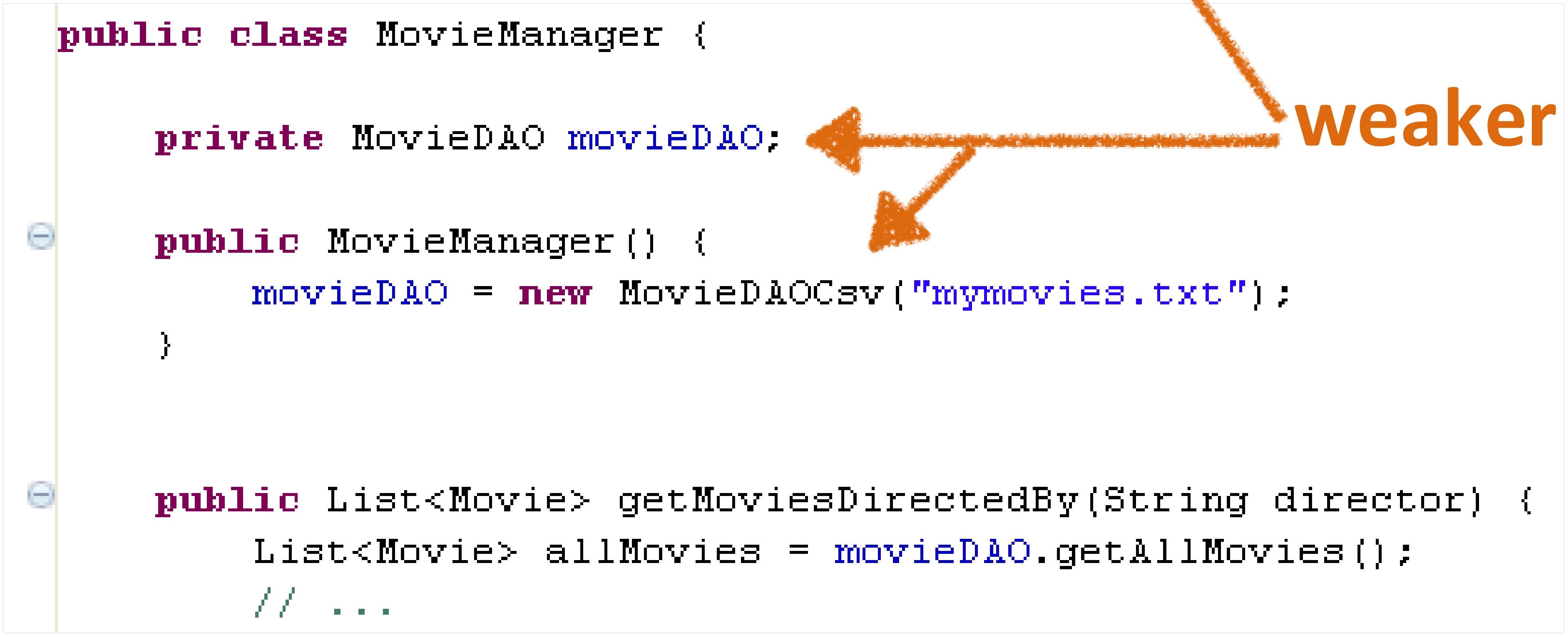
strong coupling



```
public class MovieManager {

    private MovieDAOCsv movieDAOCsv;

    public MovieManager() {
        movieDAOCsv = new MovieDAOCsv("mymovies.txt");
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAOCsv.getAllMovies();
        // ...
```

Using an interface

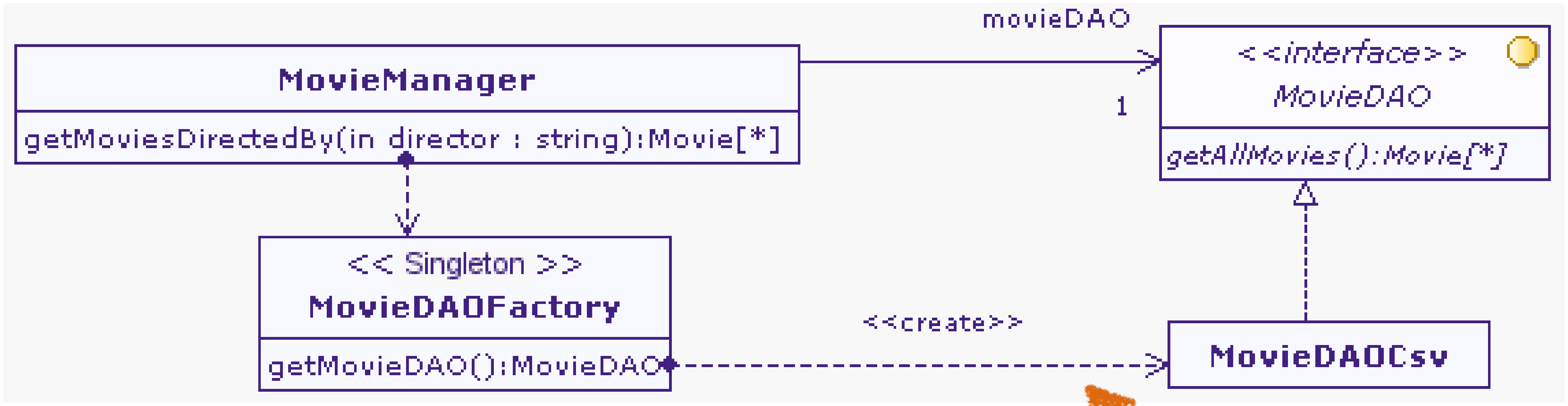


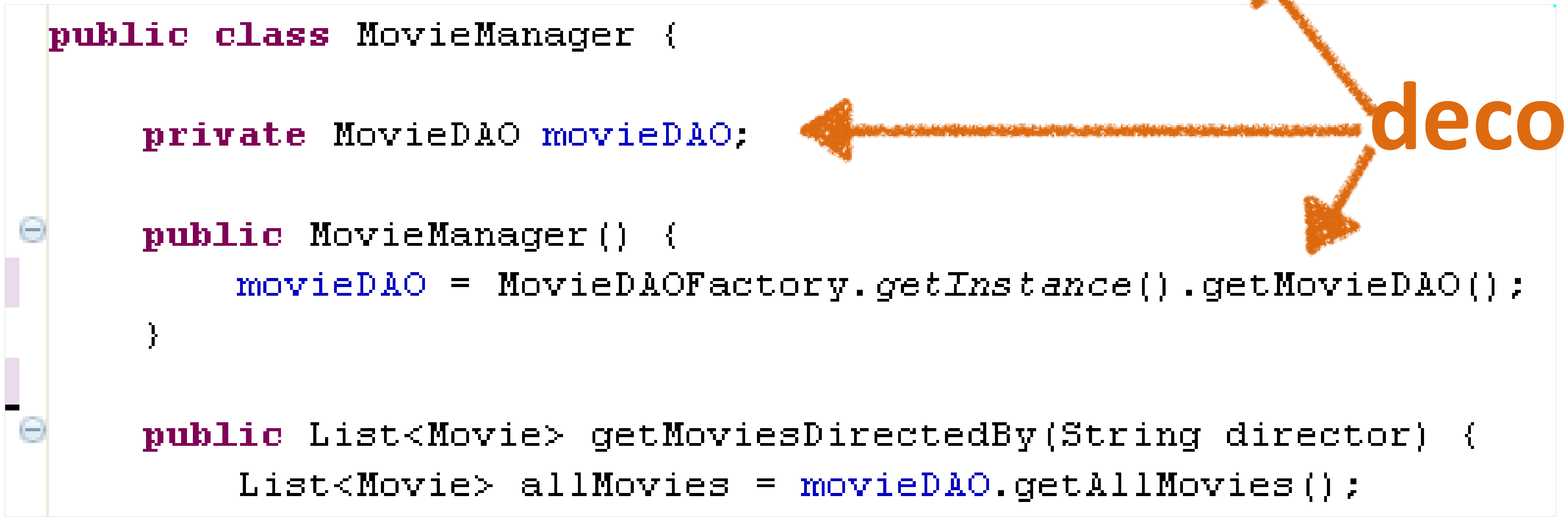


```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = new MovieDAOCsv("mymovies.txt");
    }


    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
        // ...
```

weaker coupling

Using a factory





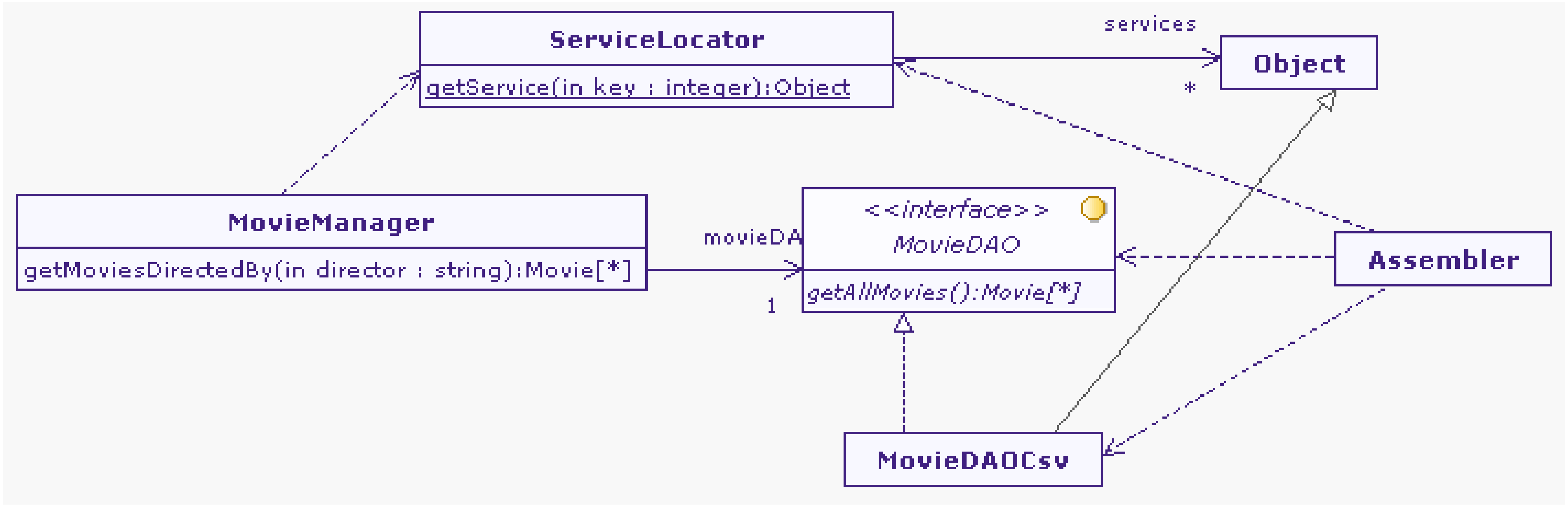
```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = MovieDAOFactory.getInstance().getMovieDAO();
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
```

decoupling

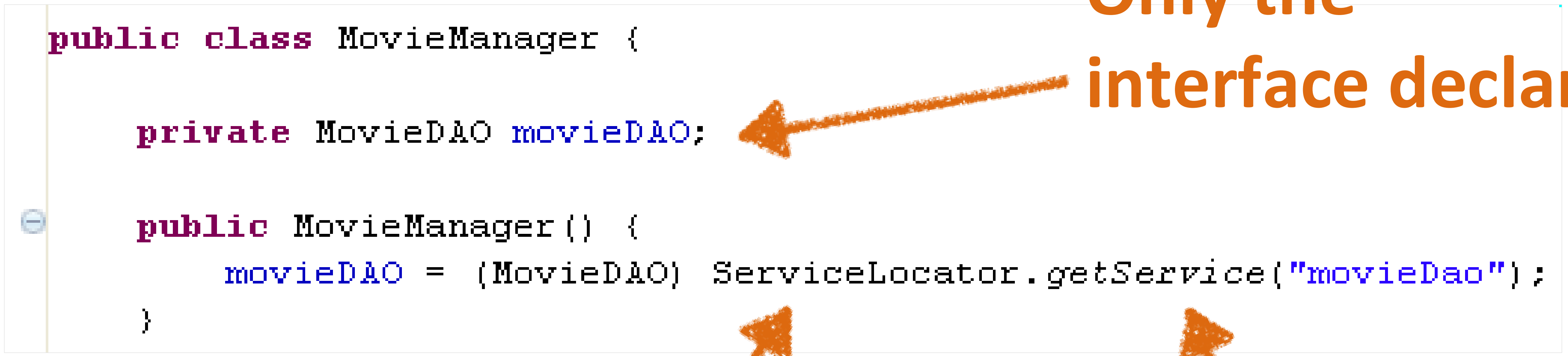
But the factory is a class in our application…

Looking for dependencies

An external object is responsible for creating instances and add them to a list of services
These services are managed by a provider by associating them with key (string)



Looking for dependencies



```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = (MovieDAO) ServiceLocator.getService("movieDao");
    }
```

Only the interface declaration

Lookup

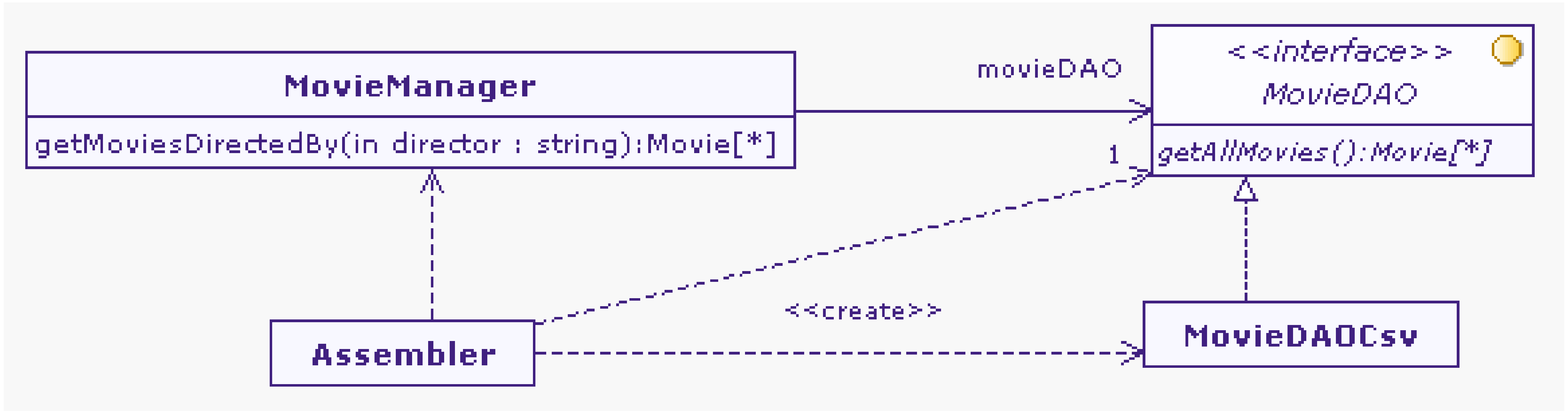
ServiceLocators is known as an Architectural Pattern

Dependency Injection

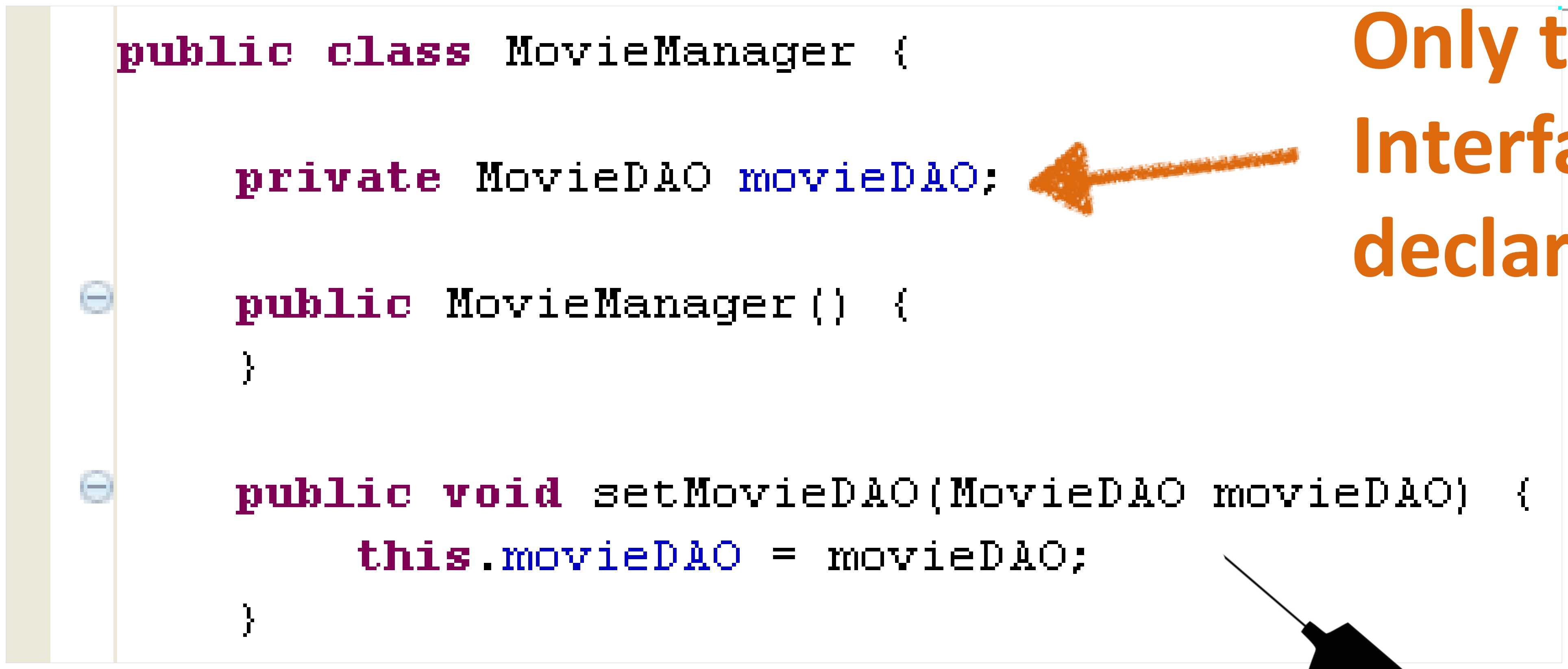
An external object is responsible for creating the application

It creates instances

It injects them in the classes it uses



Dependency Injection

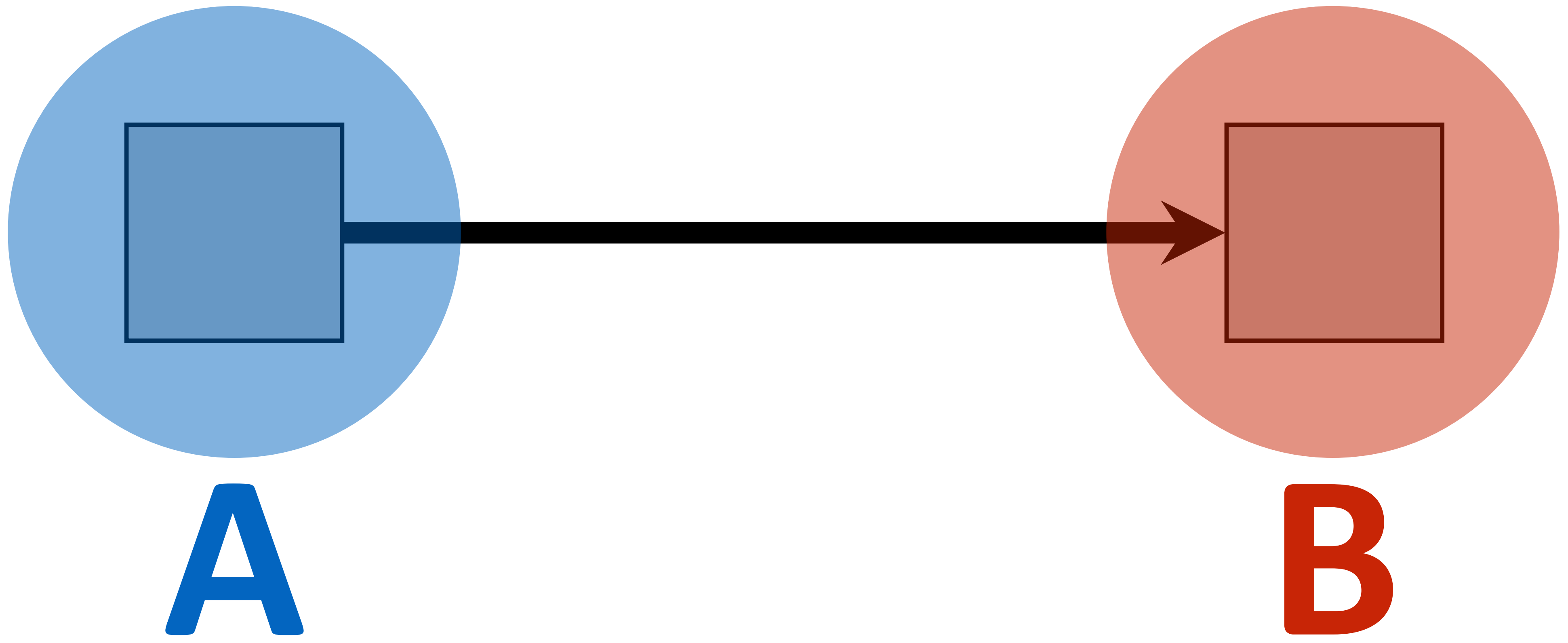


```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
    }

    public void setMovieDAO(MovieDAO movieDAO) {
        this.movieDAO = movieDAO;
    }
```

Only the Interface declaration

creates movieDAO
calls setMovieDAO

Problem: Component Dependencies



Application To Spring

(Spring) Beans

- **The container manages beans**
  - Created from the metadata
- **A bean is defined primarily by**
  - one (or several) unique identifiers -> **Unique within the container**
  - a complete class name Ex : *io.spring.hellospring.HelloWorld*
  - behavior within the container: scope, lifecycle, …
  - references to its dependencies

Naming

- In the container, a bean has different identifiers
  - **A unique identifier with the attribute id**
  - Aliases with the attribute name

- Naming is not mandatory
  - In this case, **the container generates a unique name**

- **The standard Java convention is used**
  - Ex : userDAO, authenticationService, …

Bean scope

Defines the strategy to create and store bean instances

- singleton : 1 instance per container (default)
- prototype : 1 instance per bean retrieval
- request : 1 instance per http request
- session : 1 instance per http session
- global : 1 instance per global http session

Configuration

- With XML (!)
  - Avantage (anyway) : non intrusive…
- With annotations in Java classes
  - Simplification of the configuration process (convention over configuration)
- With specific configuration classes
  - @Configuration

@Autowired on an attribute (DON’T)

But you should use it in test cases

Using @Autowired to inject helloWorld

```
import org.springframework.beans.factory.annotation.Autowired;

@Component
public class HelloWorldCaller {

  @Autowired
   // The attribute name does not matter, only account type does
  private HelloWorld helloWorldService;

  public void callHelloWorld(){
        helloWorldService.sayHello();
  }
}
```

@Autowired through constructor (DO)

Using @Autowired to inject helloWorld

```
import org.springframework.beans.factory.annotation.Autowired;

@Component
public class HelloWorldCaller {

    private HelloWorld helloWorldService;

    @Autowired
    public HelloWorldCaller(HelloWorld helloWorldService){
            this.helloWorldService=helloWorldService;
    }

    public void callHelloWorld(){
            helloWorldService.sayHello();
    }
}
```

(DO)

No @Autowired needed if there is only one constructor that injects all required components

```
import org.springframework.beans.factory.annotation.Autowired;

@Component
public class HelloWorldCaller {

  private HelloWorld helloWorldService;


  public HelloWorldCaller(HelloWorld helloWorldService){
        this.helloWorldService=helloWorldService;
  }

  public void callHelloWorld(){
        helloWorldService.sayHello();
  }
}
```

@Value

- Injection of a value into a bean property
- Located before an attribute, setter, constructor parameter
- Example on a attribute

```
@Component
public class MovieDAOCsv implements MovieDAO {

   @Value("mymovies.txt")
   private String filename;
```

@Component

- **Defines a POJO class as a bean**
- The bean identifier can be specified
  - If not, it is inferred from the class name

```java
import org.springframework.stereotype.Component;

@Component("myComponent")
public class MyComponent {

   ...

}
```

@Configuration

- Annotation on a class that **only** regroups configuration directives and bean definitions
- **@Configuration inherits from @Component**
  - Possible instantiation in the standard way (<bean>, <context:component-scan>, …)
  - Autowiring can be used for attributes
- @Bean annotated methods : bean definition

```
@Configuration
public class AppConfig {

  @Bean
  public MovieDAO movieDao() {
    return new MovieDAOCsv("movies.txt");
  }

}
```

Richer @Configuration

- Automatic detection of beans : ***@ComponentScan***

```
@Configuration
 @ComponentScan(basePackages = { "com.training.spring.service" })
 public class AppConfig {
```

- Auto-detection on the whole classpath : ***@EnableAutoConfiguration***

```
 @Configuration
 @EnableAutoConfiguration(exclude={MovieDAOJdbc.class})
 public class AppConfig {
```

Even easier with SpringBoot

- **@SpringBootApplication**
    - @Configuration +
    - @EnableAutoConfiguration

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
// same as @Configuration + @EnableAutoConfiguration
public class Application {

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

What is SpringBoot after all…

- A simplification platform to code, build, and deploy quickly application server (from monoliths to microservices)
- Easy configuration
  - One single dependency to start in pom.xml + starter projects
  - Full Java configuration with default behavior (scan all components,etc.)
  - Embedded Application server (Tomcat): **@SpringBootApplication** class
    - **A single HUGE jar that contains Tomcat + the war (web archive) of the project**