P.PORTO

Design Pattern and Spring Framework

Paradigmas Emergentes para Desenvolvimento Web e Mobile

Index

Design Patterns for Web Development

Singleton Design Pattern

Strategy Design Pattern

Observer Design Pattern

Decorator Design Pattern

Factory Design Pattern

Design Patterns for Web Development

Singleton Design Pattern

```java
public class SingletonPattern {

    private SingletonPattern() {}

    private static SingletonPattern instance;

    synchronized public static SingletonPattern getInstance() {
        if (instance == null)
        {
            instance = new SingletonPattern();
        }
        return instance;
    }

}
```

Design Patterns for Web Development

Observer Design Pattern

```
public interface Subject {

    public void registerObserver(Observer o);
    public void unregisterObserver(Observer o);
    public void notifyObservers();

}
```

```
public interface Observer {

    public void update(String data);

}
```

Design Patterns for Web Development

Observer Design Pattern

```
public class DataClass implements Subject {

    String message;
    ArrayList<Observer> observerList;

    public DataClass() {
        this.observerList = new ArrayList<>();
    }

    public void setdataChanged(String msg)
    {
        this.message = msg;

        notifyObservers();
    }
```

```
    @Override
    public void notifyObservers()
    {
        for (Iterator<Observer> it =
             observerList.iterator(); it.hasNext();)
        {
            Observer o = it.next();
            o.update(message);
        }
    }

    @Override
    public void registerObserver(Observer o) {
        observerList.add(o);
    }

    @Override
    public void unregisterObserver(Observer o) {
        observerList.remove(observerList.indexOf(o));
    }
}
```

Design Patterns for Web Development

Observer Design Pattern

```java
public class MessageDisplay implements Observer {

    String message;

    @Override
    public void update(String data) {
        this.message = data;
        display();
    }

    public void display()
    {
        System.out.println("Message: "+message);
    }
}
```

```java
public class ToUpperMessageDisplay implements Observer {
    String message;

    @Override
    public void update(String data) {
        this.message = data.toUpperCase();
        display();
    }

    public void display()
    {
        System.out.println("Message: "+message);
    }

}
```

Design Patterns for Web Development

Observer Design Pattern

```
public class Main {

    public static void main(String[] args){
        MessageDisplay md = new MessageDisplay();
        ToUpperMessageDisplay tump = new ToUpperMessageDisplay();

        DataClass dt = new DataClass();
        dt.registerObserver(md);
        dt.registerObserver(tump);

        dt.setdataChanged("new message!");
    }

}
```

Design Patterns for Web Development

Strategy Design Pattern

```java
public interface Strategy {

    int solve(int a, int b);

}
```

```java
public class OperationAdd implements Strategy{
    @Override
    public int solve(int num1, int num2) {
        return num1 + num2;
    }
}
```

```java
public class Context {

    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.solve(num1, num2);
    }

}
```

Design Patterns for Web Development

Strategy Design Pattern

```java
public class OperationAdd implements Strategy{
    @Override
    public int solve(int num1, int num2) {
        return num1 + num2;
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

```java
public class OperationMultiply implements Strategy{
    @Override
    public int solve(int num1, int num2) {
        return num1 * num2;
    }
}
```

Design Patterns for Web Development

Decorator Design Pattern

```java
public interface Shape {

    void draw();

}
```

```java
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape){
        this.decoratedShape = decoratedShape;
    }

    public void draw(){
        decoratedShape.draw();
    }
}
```

```java
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape){
        System.out.println("Border Color: Red");
    }

}
```

Design Patterns for Web Development

Decorator Design Pattern

```java
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}
```

```java
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        Shape circle = new Circle();

        Shape redCircle = new RedShapeDecorator(new Circle());

        Shape redRectangle = new RedShapeDecorator(new Rectangle());
        System.out.println("Circle with normal border");
        circle.draw();

        System.out.println("\nCircle of red border");
        redCircle.draw();

        System.out.println("\nRectangle of red border");
        redRectangle.draw();
    }

}
```

Design Patterns for Web Development

Factory Design Pattern

```java
public interface Shape {

    void draw();

}
```

```java
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

```java
public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }

}
```

Design Patterns for Web Development

Factory Design Pattern

```java
public class ShapeFactory {

    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase( anotherString: "RECTANGLE")){
            return new Rectangle();

        } else if(shapeType.equalsIgnoreCase( anotherString: "SQUARE")){
            return new Square();
        }

        return null;
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        Shape shape2 = shapeFactory.getShape( shapeType: "RECTANGLE");
        shape2.draw();

        Shape shape3 = shapeFactory.getShape( shapeType: "SQUARE");
        shape3.draw();

    }
}
```

Design Patterns for Web Development

Factory Design Pattern

```
public class Main {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        Shape shape2 = shapeFactory.getShape( shapeType: "RECTANGLE");
        shape2.draw();

        Shape shape3 = shapeFactory.getShape( shapeType: "SQUARE");
        shape3.draw();

    }
}
```

Design Patterns for IoC Containers

- Inversion of Control (IoC)
- Dependency Inversion Principle (DIP)
- Dependency Injection (DI)
- Inversion of Control Containers (IoC Containers)

Inversion of Control

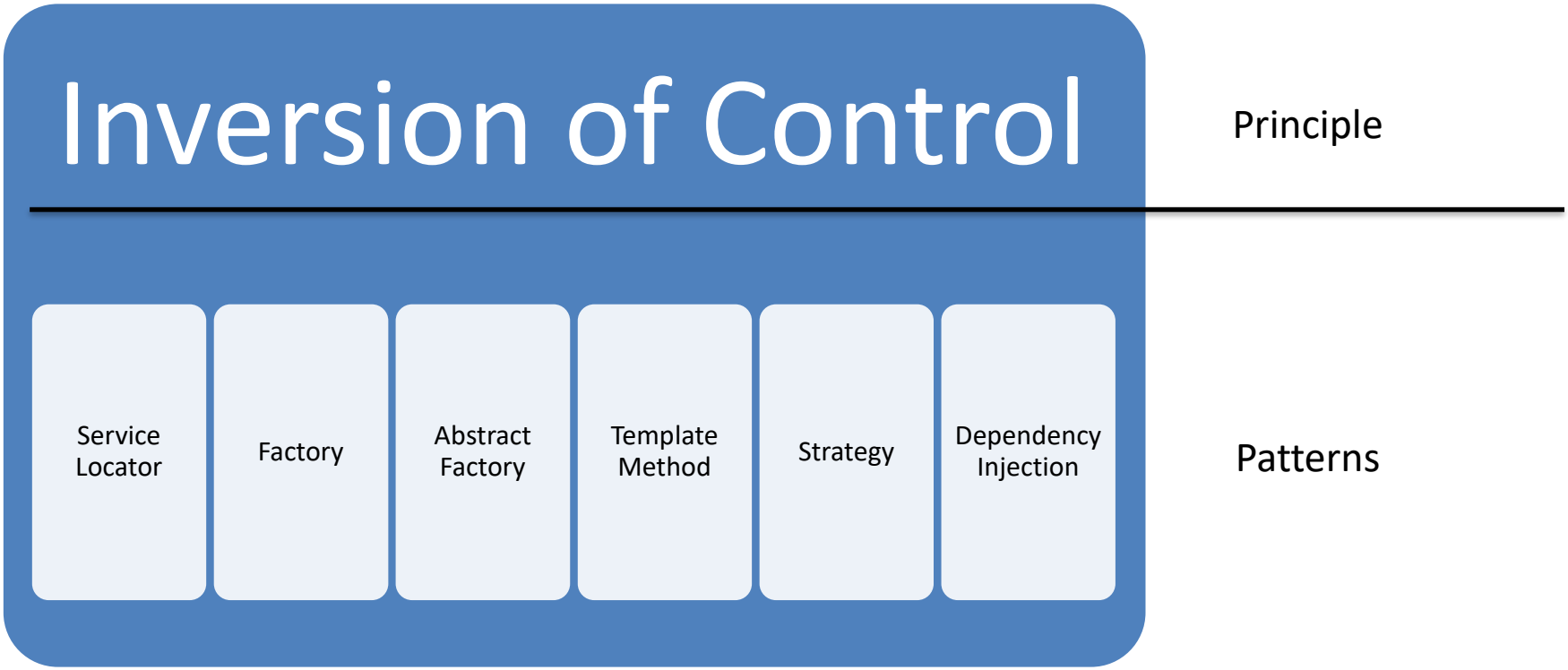
IoC is a design principle which recommends the inversion of control in object-oriented design to achieve loose coupling between classes.

Control refers to any additional responsibilities a class has, other than its main responsibility:

- control over the flow of an application
- control over the dependent object creation
- binding

Remember SRP - Single Responsibility Principle

Inversion of Control



Inversion of Control

```
public class BusinessLogic {

    DataAccess _dataAccess;

    public BusinessLogic()
    {
        _dataAccess = new DataAccess();
    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }
}
```

Tight Coupled

```
public class BusinessLogic {

    DataAccess _dataAccess;

    public BusinessLogic()
    {
        _dataAccess = DataAccessFactory.GetDataAccessObj();
    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }
}
```

```
public class DataAccessFactory {

    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }

}
```

Loose Coupled

Dependency Injenction Principle

The DIP principle was invented by Robert Martin. He is a founder of the SOLID principles.

DIP suggests that high-level modules should not depend on low level modules. Both should depend on abstraction.

The DIP principle also helps in achieving loose coupling between classes

It is recommended to use DIP and IoC together

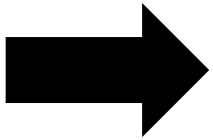
Dependency Injenction Principle (DIP)

DIP Definition

1. High-level modules should not depend on low-level modules. Both should depend on the abstraction.
2. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Injenction Principle

```
public class BusinessLogic {

    DataAccess _dataAccess;

    public BusinessLogic()
    {
        _dataAccess = new DataAccess();
    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }
}
```



```
public class BusinessLogic {

    IDataAccess _dataAccess;

    public BusinessLogic()
    {
        _dataAccess = DataAccessFactory.GetDataAccessObj();
    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }
}
```

```
public interface IDataAccess {

        String GetBusinessData(int id);

}
```

```
public class DataAccessFactory {

    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }

}
```

Dependency Injection (DI)

DI is a design pattern which implements the IoC principle to invert the creation of dependent objects

It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways

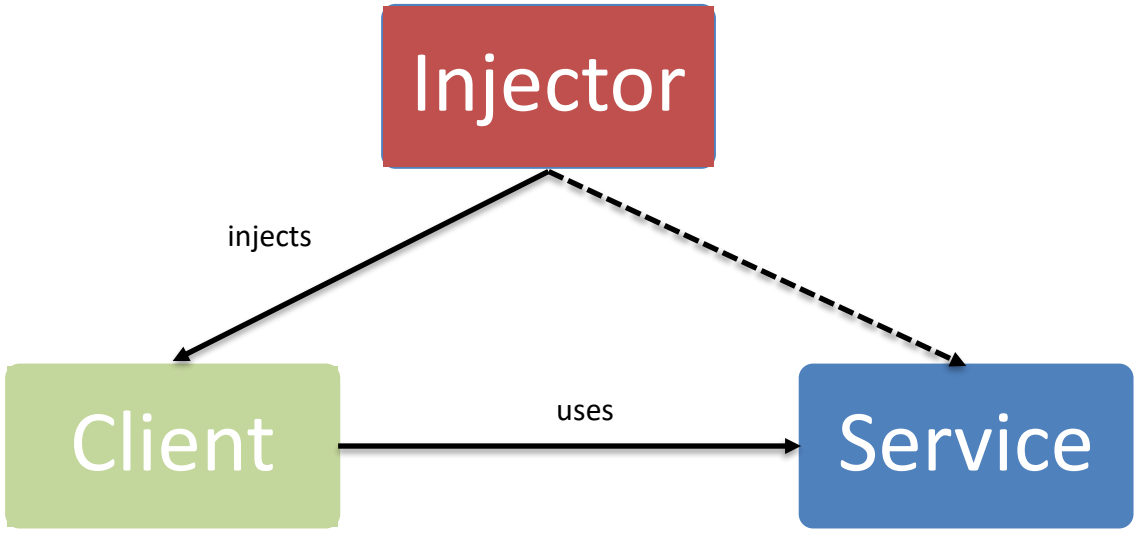
Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them

Dependency Injection (DI)

The Dependency Injection pattern involves 3 types of classes

1. Client Class: The client class (dependent class) is a class which depends on the service class
2. Service Class: The service class (dependency) is a class that provides service to the client class.
3. Injector Class: The injector class injects the service class object into the client class.

Dependency Injection (DI)



Dependency Injection (DI)

Constructor Injection

- the injector supplies the service (dependency) through the client class constructor.

Property Injection

- the injector supplies the dependency through a public property of the client class.

Method Injection

- the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class

Dependency Injection (DI)

Constructor Injection

```
public class BusinessLogic {

    IDataAccess _dataAccess;

    public BusinessLogic(IDataAccess dataAccess)
    {
        _dataAccess = dataAccess;
    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }
}
```

```
public class BusinessService {
    BusinessLogic _customerBL;

    public BusinessService()
    {
        _customerBL = new BusinessLogic(new DataAccess());
    }

    public String GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }

}
```

Dependency Injection (DI)

Property Injection

```
public class BusinessLogic {

    IDataAccess DataAcessObject;

    public BusinessLogic()
    {    }

    public String GetCustomerName(int id)
    {
        return DataAcessObject.GetBusinessData(id);
    }


    public IDataAccess get_dataAccess() {
        return DataAcessObject;
    }

    public void set_dataAccess(IDataAccess _dataAccess) {
        this.DataAcessObject = _dataAccess;
    }
}
```

```
public class BusinessService {
    BusinessLogic _customerBL;

    public BusinessService()
    {
            _customerBL = new BusinessLogic();
            _customerBL.DataAcessObject = new DataAccess();
        }

        public String GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }

}
```

Dependency Injection (DI)

Method Injection

```
public class BusinessLogic implements IDataAccessDependency {

    IDataAccess _dataAccess;

    public BusinessLogic()
    {    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }


    @Override
    public void SetDependency(IDataAccess _dataAccess) {
        this._dataAccess = _dataAccess;
    }
}
```

```
public class BusinessService {
    BusinessLogic _customerBL;

    public BusinessService()
    {
        _customerBL = new BusinessLogic();
        ((IDataAccessDependency) _customerBL).SetDependency(new DataAccess());
    }

    public String GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }

}
```

```
public interface IDataAccessDependency {

    void SetDependency(IDataAccess _dataAccess);
}
```

IoC Container

The IoC container is a framework used to manage automatic dependency injection throughout an application

The IoC container creates an object of the specified class and also injects all the dependency objects through a constructor, a property or a method at run time

In this way we do not need to create objects manually

IoC Container

IoC containers must provide support for the following DI lifecycle.

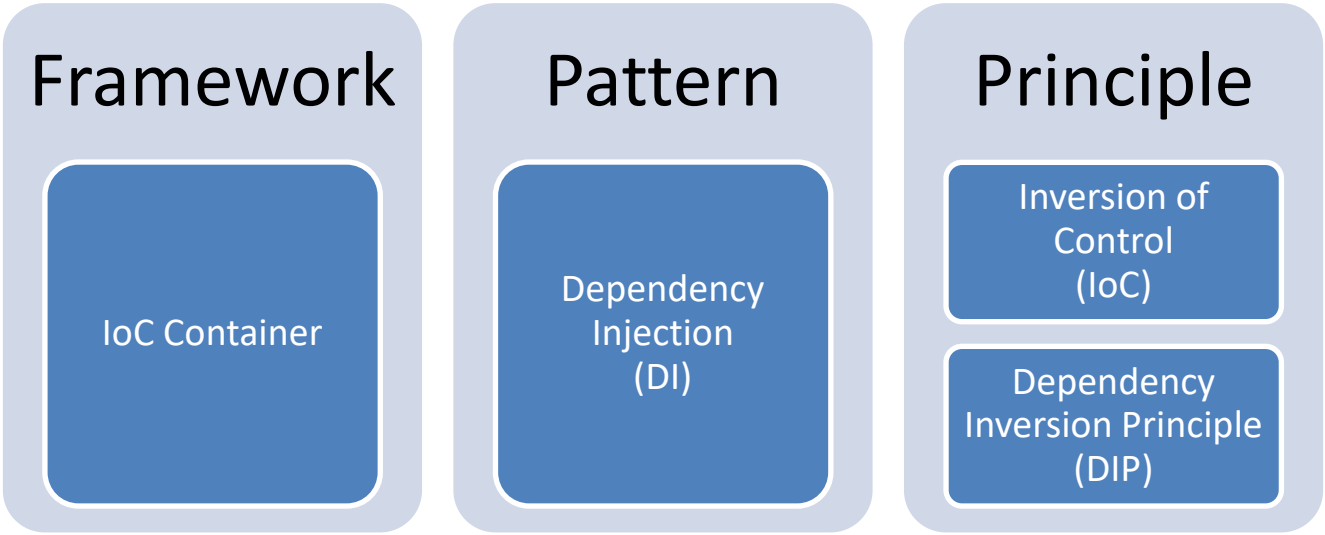
- Register:
  - The container must know which dependency to instantiate when it encounters a particular type.
- Resolve:
  - When using the IoC container, we don't need to create objects manually. The container does it for us. This is called resolution
- Dispose:
  - The container must manage the lifetime of the dependent objects

IoC Container

Examples of IoC Container Frameworks:

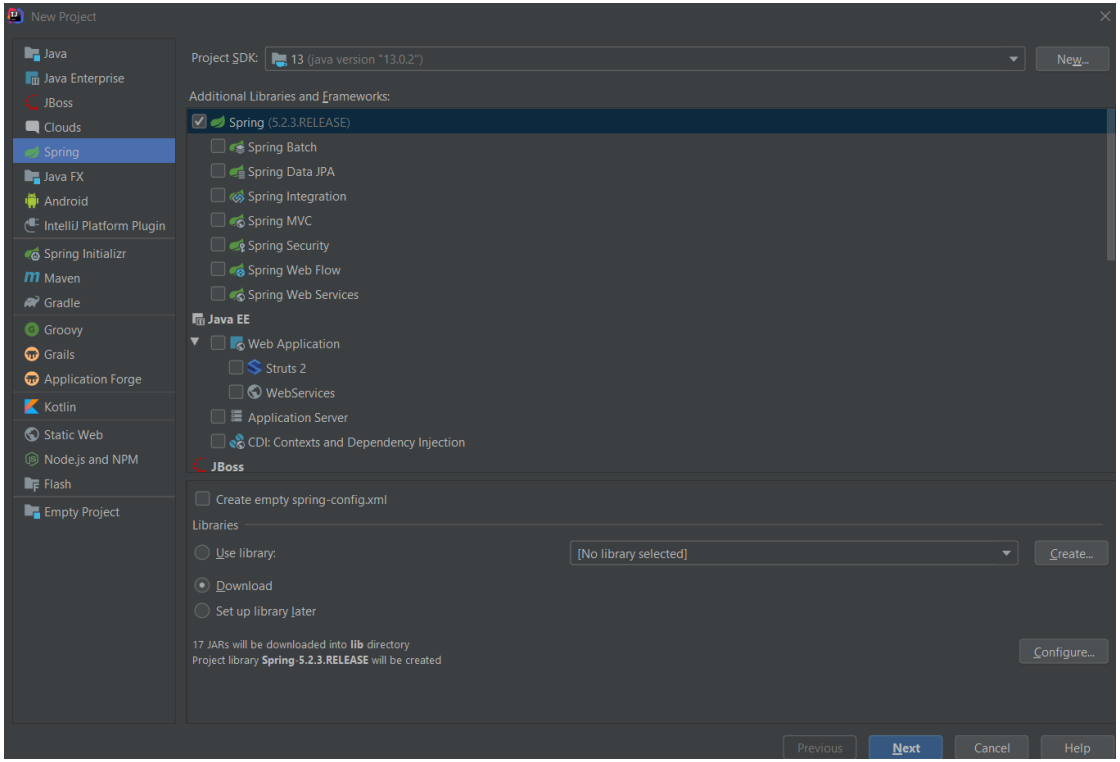
- Java
  - Spring Core
  - JavaServerFaces
  - Java CDI
  - …
- .Net
  - Unity
  - Ninject
  - …

IoC Container



IoC Container

Exemplo



IoC Container

Exemplo

```
public class BusinessLogic implements IDataAccessDependency {

    IDataAccess _dataAccess;

    public BusinessLogic()
    {    }

    public String GetCustomerName(int id)
    {
        return _dataAccess.GetBusinessData(id);
    }


    @Override
    public void SetDependency(IDataAccess _dataAccess) {
        this._dataAccess = _dataAccess;
    }
}
```

```
public interface IDataAccessDependency {

    void SetDependency(IDataAccess _dataAccess);
}
```

```
public class DataAccess implements IDataAccess {

    public String GetBusinessData(int id) {
        return "Dummy Business Name";
    }

}
```

```
public interface IDataAccess {

        String GetBusinessData(int id);

}
```

IoC Container

Exemplo

```
public class BusinessService {
    BusinessLogic _customerBL;

    public BusinessService()
    {
        _customerBL = new BusinessLogic();
        ((IDataAccessDependency) _customerBL).SetDependency(new DataAccess());
    }

    public String GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }

}
```

IoC Container

Exemplo

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context" xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="businessService"    class="demo_5.BusinessService"></bean>

</beans>
```

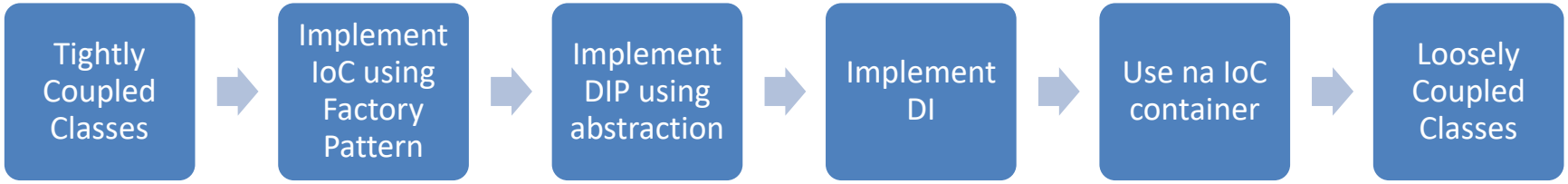
IoC Container

Exemplo

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context" xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="businessService"    class="demo_5.BusinessService"></bean>

</beans>
```

File beans.xml where we specify which objects (beans) should be initialized automatically by the Spring Framework

Design Patterns for IoC Containers



More references:

- https://martinfowler.com/articles/injection.html

SOLID Principles

Single Responsibility Principle

Open Closed Principle

Liskov's Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

SOLID Principles

Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change
- When requirements change, this implies that the code has to undergo some reconstruction
- The more responsibilities a class has, changes will be harder to implement
- The responsibilities of a class are coupled to each-other, as changes in one may result in additional changes in others

SOLID Principles

Open Closed Principle

- You should be able to extend a class's behavior, without modifying it.
- This principle is the foundation for building code that is maintainable and reusable
- Open for extension
- Closed for modification

SOLID Principles

Open Closed Principle

- In frameworks like spring, we can not change their core logic or request processing
- But we modify the desired application flow just by extending some classes and plugin them in configuration files

SOLID Principles

Liskov's Substitution Principle

- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
- That requires the objects of your subclasses to behave in the same way as the objects of your superclass
- In other words, as simple as that, a subclass should override the parent class methods in a way that does **not break functionality from a client's point of view**

SOLID Principles

Interface Segregation Principle

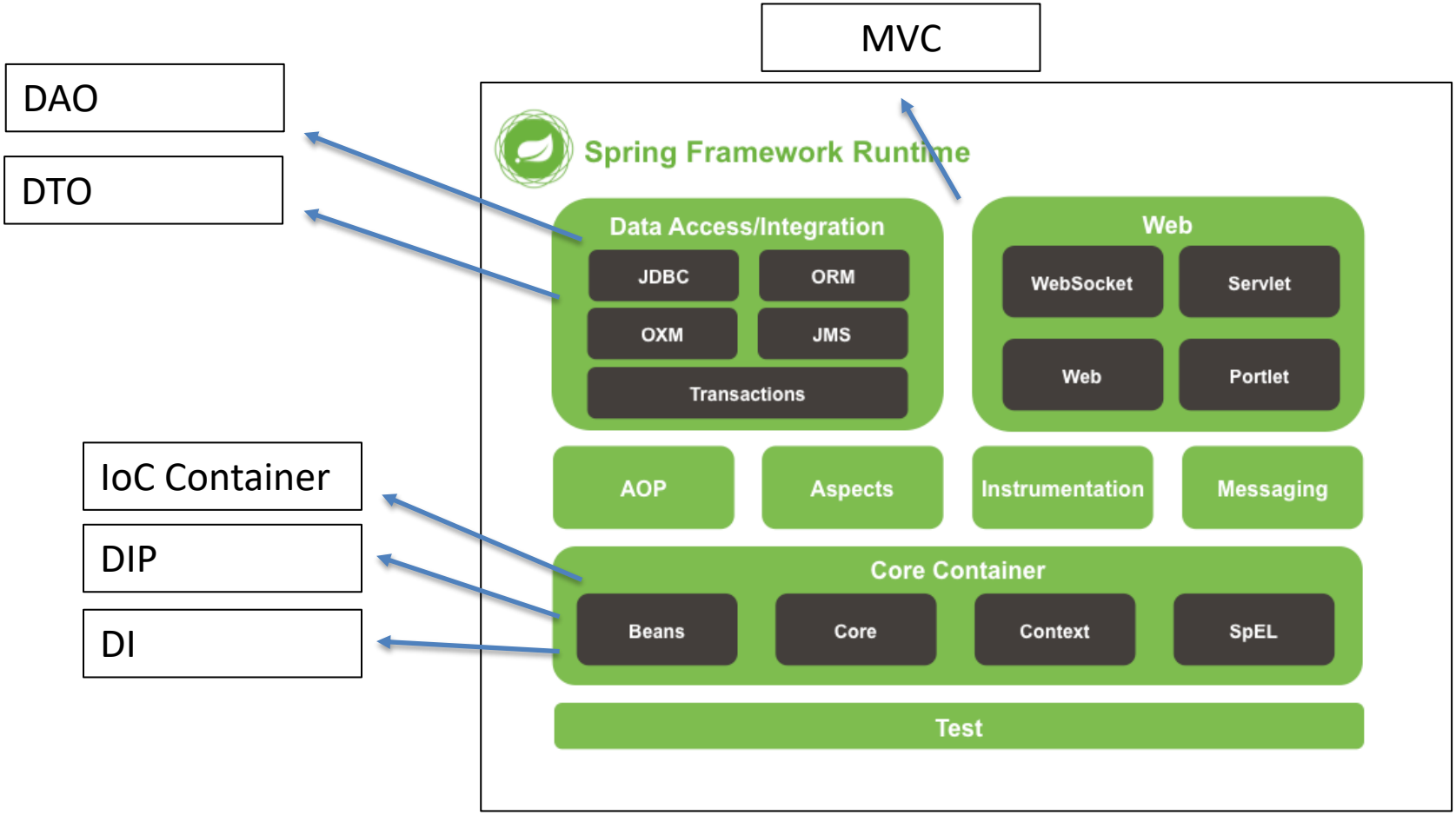
- Clients should not be forced to implement unnecessary methods which they will not use
- Na example is an interface Report with methods for diferente reports (html, json, ..) . This violates the interface segregation principles because a cliente will always be forced to implemente all reports even if does not need them

SOLID Principles

Dependency Inversion Principle

- We should depend on abstrations, not on concretions
- We should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together
- In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module

Spring Framework and Design Patterns



Reference: https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html

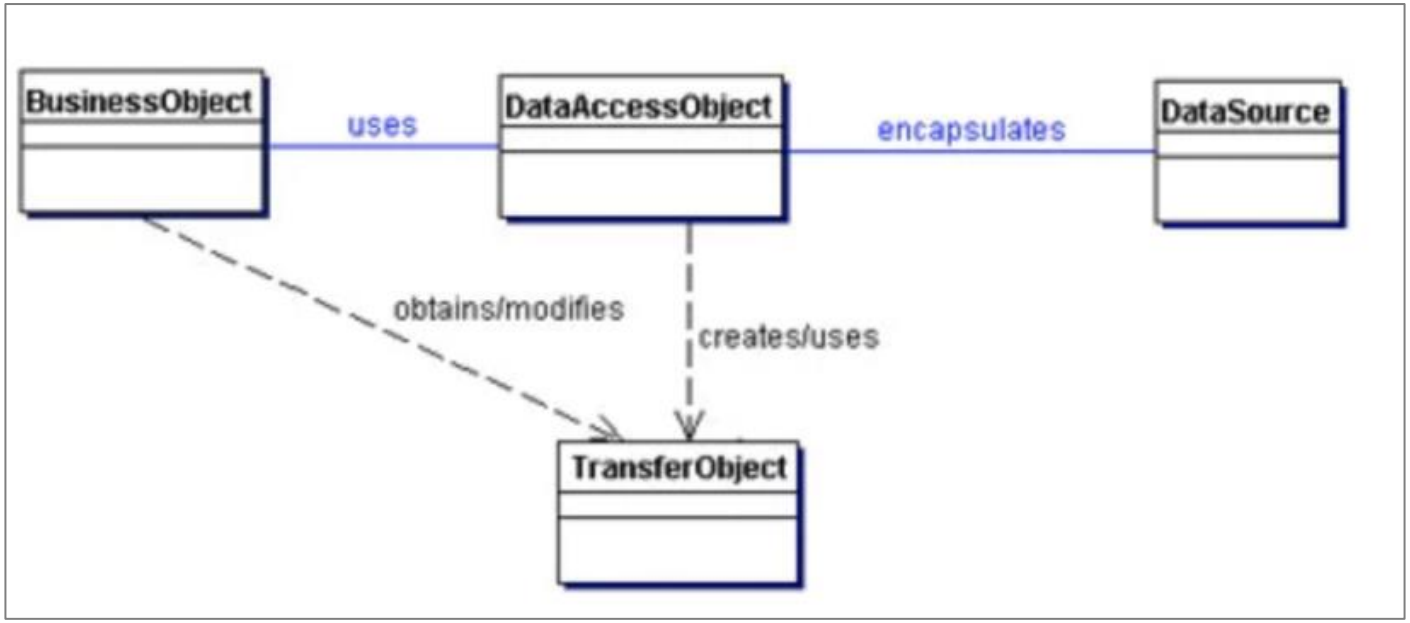
Spring Framework and Design Patterns

Bean in a IoC container

```
@RestController
public class RestController {

    private final ArticleRepositoy repo;

    public RestController(ArticleRepositoy repo) {
        this.repo  = repo;
    }

    @GetMapping("/article/{id}")
    public EntityModel<Article> getArticle(@PathVariable Long id) {
        Article a = this.repo.findById(id).get();
        long numArticles = this.repo.count();
        EntityModel<Article> resp = new EntityModel<>(a, linkTo(methodOn(RestController.class).getArticle(id)).withSelfRel(),
                linkTo(methodOn(RestController.class).allArticles()).withRel("articles"));
        if (numArticles>id){
            resp.add(linkTo(methodOn(RestController.class).getArticle( id: id+1)).withRel("next"));
        }
        if (id>0){
            resp.add(linkTo(methodOn(RestController.class).getArticle( id: id+1)).withRel("previous"));
        }
        return resp;
    }
...
```

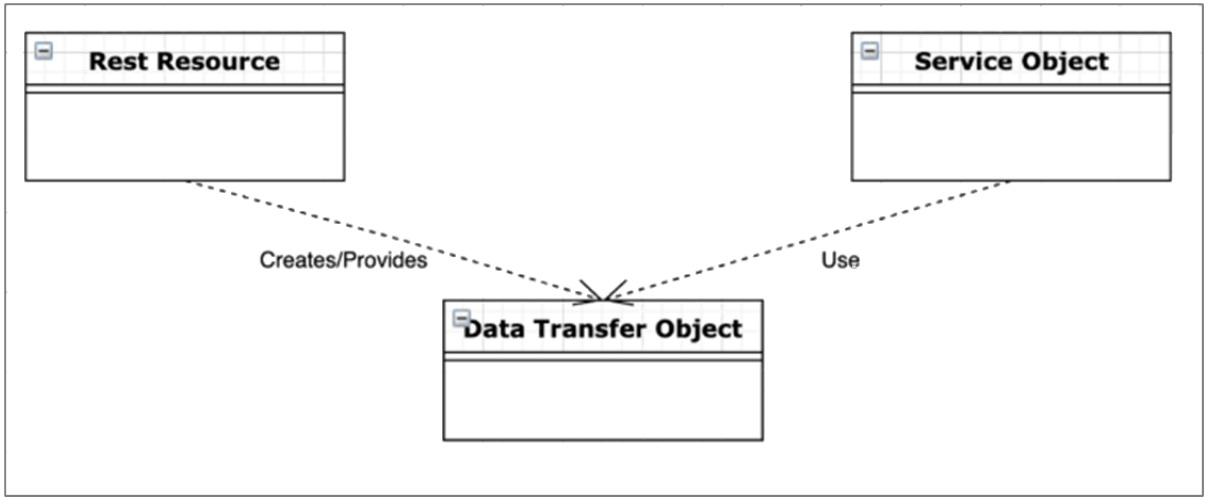
Spring Framework and Design Patterns

Data Transfer Object (DTO)



Spring Framework and Design Patterns

Data Transfer Object (DTO)



Spring Framework and Design Patterns

Data Access Object Pattern (DAO)

```
public class User {

    private String name;
    private String email;

    // constructors / standard setters / getters
}
```

```
public interface Dao<T> {

    Optional<T> get(long id);

    List<T> getAll();

    void save(T t);

    void update(T t, String[] params);

    void delete(T t);
}
```

Spring Framework and Design Patterns

Data Access Object Pattern (DAO)

```
public class UserDao implements Dao<User> {

    private List<User> users = new ArrayList<>();

    public UserDao() {
        users.add(new User("John", "john@domain.com"));
        users.add(new User("Susan", "susan@domain.com"));
    }

    @Override
    public Optional<User> get(long id) {
        return Optional.ofNullable(users.get((int) id));
    }

    @Override
    public List<User> getAll() {
        return users;
```

Spring REST Application

Can you identify other Design Patterns used in our sample application?

Spring REST Application

Lest's start improving our REST API with some design patterns

To bootstrap our work you can use a sample project from this respository

- https://github.com/fasIPP/SpringRest

Spring REST Application

Lets add hypermedia to our REST API following the HATEOAS principle from last class

```
@GetMapping("/article/{id}")
public EntityModel<Article> getArticle(@PathVariable Long id) {
    Article a = this.repo.findById(id).get();
    long numArticles = this.repo.count();

    EntityModel<Article> resp = new EntityModel<>(a,
            linkTo(methodOn(RestController.class).getArticle(id)).withSelfRel(),
            linkTo(methodOn(RestController.class).allArticles()).withRel("articles")
    );

    return resp;
}
```

We should add the EntityModel class to add the _link section in our JSON Response

Spring REST Application

We can do better and add links for news navigation

```
@GetMapping("/article/{id}")
public EntityModel<Article> getArticle(@PathVariable Long id) {
    Article a = this.repo.findById(id).get();
    long numArticles = this.repo.count();

    EntityModel<Article> resp = new EntityModel<>(a,
            linkTo(methodOn(RestController.class).getArticle(id)).withSelfRel(),
            linkTo(methodOn(RestController.class).allArticles()).withRel("articles")
    );
    if (numArticles>id){
        resp.add(linkTo(methodOn(RestController.class).getArticle( id: id+1)).withRel("next"));
    }
    if (id>0){
        resp.add(linkTo(methodOn(RestController.class).getArticle( id: id-1)).withRel("previous"));
    }
    return resp;
}
```

Spring REST Application

We can also improve the readability of our code with functional primitives

```
@GetMapping("/article/{id}")
EntityModel<Article> getArticle(@PathVariable Long id) {

    Article article = repository.findById(id)
            .orElseThrow(() -> new ArticleNotFoundException(id));

    return new EntityModel<>(article,
            linkTo(methodOn(ArticleController.class).getArticle(id)).withSelfRel(),
            linkTo(methodOn(ArticleController.class).all()).withRel("articles"));
}
```

Spring REST Application

We can also improve the readability of our code with functional primitives

```
@GetMapping("/articles2")
CollectionModel<EntityModel<Article>> getAllArticles() {

    List<EntityModel<Article>> articles = repository.findAll().stream()
            .map(article -> new EntityModel<>(article,
                    linkTo(methodOn(ArticleController.class).one(article.getId())).withSelfRel(),
                    linkTo(methodOn(ArticleController.class).getAllArticles()).withRel("articles")))
            .collect(Collectors.toList());

    return new CollectionModel<>(articles,
            linkTo(methodOn(ArticleController.class).all()).withSelfRel());
}
```

Spring REST Application

Lets Abstract our HATEOAS with design patterns

```
@Component
public class ArticleModelAssembler implements
        RepresentationModelAssembler<Article, EntityModel<Article>> {

@Override
public EntityModel<Article> toModel(Article article) {
        return new EntityModel<>(article,
        linkTo(methodOn(ArticleController.class).one(article.getId())).withSelfRel(),
        linkTo(methodOn(ArticleController.class).all()).withRel("articles"));
    }
}
```

Spring REST Application

Lets Abstract our HATEOAS with design patterns

```
public class ArticleController {

    private final ArticleRepository repository;
    private final ArticleModelAssembler assembler;

    public ArticleController(ArticleRepository repository,
                             ArticleModelAssembler assembler) {
        this.repository = repository;
        this.assembler = assembler;
    }

    @GetMapping("/articles")
    CollectionModel<EntityModel<Article>> getAllArticles() {

        List<EntityModel<Article>> articles = repository.findAll().stream()
                .map(assembler::toModel)
                .collect(Collectors.toList());

        return new CollectionModel<>(articles,
                linkTo(methodOn(ArticleController.class).all()).withSelfRel());
```

Spring REST Application

Lets Abstract our HATEOAS with design patterns

```
@GetMapping("/article/{id}")
EntityModel<Article> getArticle(@PathVariable Long id) {

    Article article = repository.findById(id)
            .orElseThrow(() -> new ArticleNotFoundException(id));

    return assembler.toModel(article);
}
```

Spring REST Application

Lets add Pagination to our REST API

Remeber it is critical to control the amount of data delivered on a synchrounous comunication, otherwise we may have performance issues or worse Denial of Service problems

Lets add Pagination to our REST API

Spring REST Application

We have improved our REST application, however, is it enough for modern real-time applications?

Supose we need to be notified from the server about live news or news alerts. Is a REST API enougth?

Spring REST Application

We have improved our REST application, however, is enough?

Supose we need to use the observer pattern in our PWA, can we do it with a REST API?

References

Design Patterns

- https://www.tutorialspoint.com/design_pattern/index.htm

Spring Framework Design Patterns

- https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html
- https://www.baeldung.com/spring-framework-design-patterns

Spring IoC Containers

- https://howtodoinjava.com/java-spring-framework-tutorials/

Spring Rest

- https://spring.io/guides/gs/rest-hateoas/
- https://spring.io/guides/gs/securing-web/
- https://docs.spring.io/spring-data/rest/docs/2.0.0.M1/reference/html/paging-chapter.html
- https://www.baeldung.com/rest-api-pagination-in-spring

P.PORTO

Design Pattern and Spring Framework

Paradigmas Emergentes para Desenvolvimento Web e Mobile