Tema 3

Arquitecturas centradas en el dominio

Parte II

Arquitectura de **puertos y adaptadores**

Contenidos

1. Arquitecturas de software.
2. Arquitecturas centradas en el dominio.
3. Inyección de depedencias.
4. Arquitectura puertos y adaptadores (aka *Arquitectura Hexagonal*)
5. Spring Boot.

1. Arquitectura de Software

¿Qué son las arquitecturas de software y para qué sirven?

1. Arquitectura de software – ¿Qué son?

La arquitectura es **diseño de alto nivel**.

- No miramos unidades pequeñas (ej., clases)
- Miramos grandes estructuras.

1. Arquitectura de software – Definición

> "**Architecture** is about the important stuff. Whatever that is."
>
> – *Ralph Johnson*

¿Qué es lo importante?

- La estructura y organización del software, no su implementación.
- Decisiones que queremos que sean correctas desde el principio.
- Elementos que son costosos de cambiar después de que se implementen.
- Las decisiones esenciales que van a permitir que el software tenga los atributos de calidad requeridos.

2. Arquitecturas centradas en el dominio.

Desacoplando la tecnología del dominio

2. Arquitecturas de dominio – Motivación

Existen arquitecturas cuyo objetivo principal es **proteger el dominio frente a decisiones técnicas**.

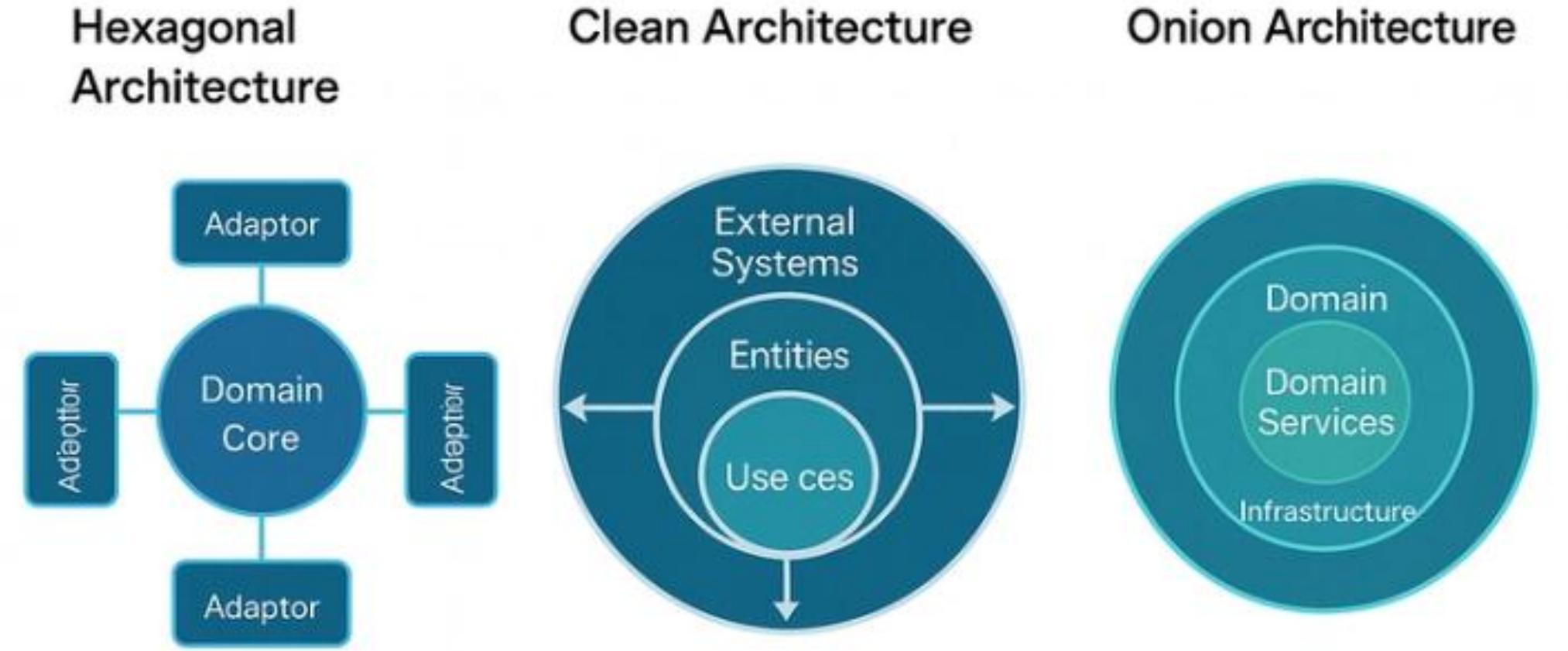
¿Qué caracteriza a estas arquitecturas?

- Consideran la lógica de negocio como el núcleo del sistema.
- Aíslan el dominio de frameworks, BBDDs, interfaces de usuario, …
- Cambio de paradigma:
  El dominio no depende de la tecnología => La tecnología se adapta al dominio.

¿Qué problema resuelven?

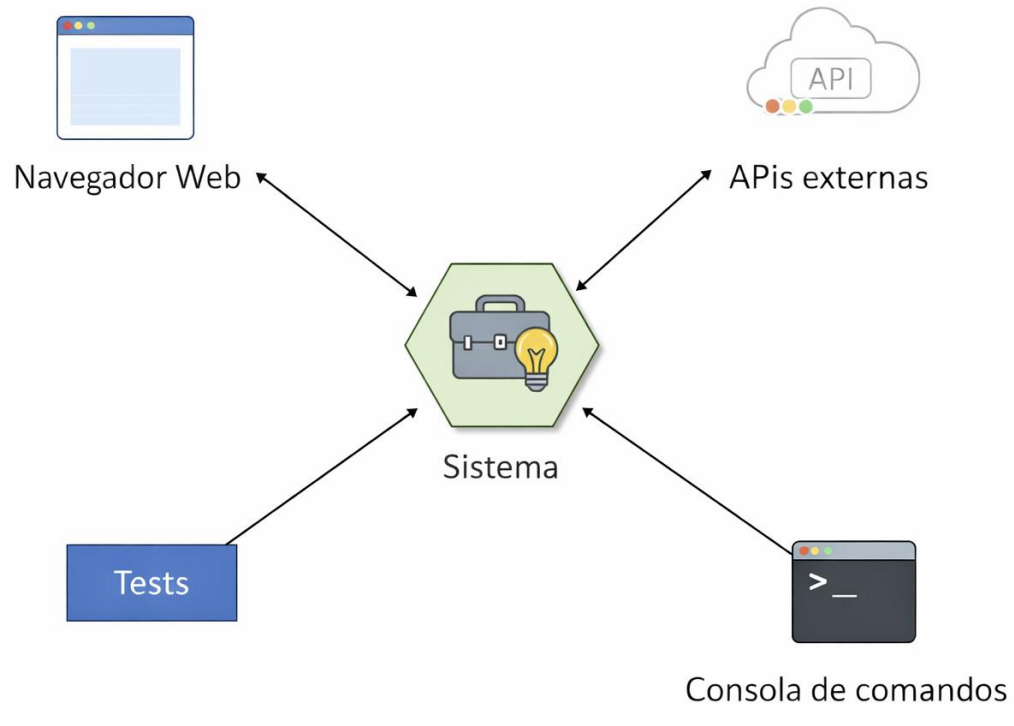
- Evitan que los cambios tecnológicos afecten a la lógica de negocio.
- Facilitan el desarrollo y las pruebas en aislamiento.
- Permiten que el dominio evolucione de forma independiente.

2. Arquitecturas de dominio – Ejemplos



2. Arquitecturas de dominio – Objetivos

- Ejecutar sin interfaz de usuario ni BBDD.
- Independizarse del entorno y tecnologías.
- Permitir múltiples formas de uso
  usuarios, otras aplicaciones, tests automáticos, procesos batch o scripts.
- Desarrollo y pruebas en aislamiento.
- Permitir el cambio de tecnologías sin afectar a la lógica de negocio.
- Reutilizar la lógica en distintos contextos.
- Integración con otros sistemas sin intervención humana.



2. Arquitecturas de dominio – Motivación

En el lado de **uso de la aplicación.**

Problema:
la lógica de negocio se mezcla con el código de la interfaz de usuario.

¿Por qué es un problema?

- No es sencillo probar el sistema con *tests automáticos.*
  Hay que arrancar la UI, escribir en los campos, pulsar un botón, observar el resultado.
- No es posible hacer que el sistema sea manejado por otras entidades:
  - Un sistema de procesamiento *batch.*
  - Una IA que utiliza nuestro programa.
- Se pretende poder ejecutar la aplicación en modo ***headless*.**

2. Arquitecturas de dominio – Motivación

En el lado de **qué necesita la aplicación.**

Problema:

Una aplicación depende de diversas tecnologías:
Servidores de base de datos, servicios externos o dispositivos.

El sistema esté ligado *fuertemente* a estas tecnologías.

¿Por qué es un problema?

- Impide el desarrollo incremental.
  Para construir el sistema, hay que configurar primero todas las dependencias.
- Complica las pruebas automáticas.
- Complica el mantenimiento si las tecnologías cambian o se desean reemplazar.

2. Arquitecturas de dominio – Ejemplo

Requisitos funcionales

Dado un producto que tiene un precio sin IVA y una categoría asociada, la aplicación calculará el IVA que le corresponde.

Requisitos no funcionales

El IVA se obtiene a partir de un servicio proporcionado el gobierno.
Por desgracia, el gobierno suele cambiar la API del servicio con frecuencia y hay que adaptarse.

La aplicación debe tener una interfaz de usuario (UI).
La aplicación debe poder ser usada por otras aplicaciones como una librería.

¿Cuál es la implementación más simple que cumple esto?

2. Arquitecturas de dominio – Ejemplo

Una primera versión de la aplicación podría ser esta. ¿Qué problemas ves?

```
public class CalculadorImpuestos {
  public void main(String[] args) {
    Producto producto = leerProductoUI();
    double impuestos = calculadoraIVA(producto.getPrecio(), producto.getTipoProducto());
    mostrarIVA(impuestos);
  }
  // Dado el importe y el tipo de producto calculo el IVA asociado
  public static double calculadoraIVA(double precioSinIVA, TipoProducto tipoProducto) {
    return switch(tipoProducto) {
      case BASICO -> precioSinIVA * 0.04;
      case ALIMENTACION -> precioSinIVA * 0.10;
      case LUJO -> precioSinIVA * 0.21;
    }
  }
  // Enumerado para determinar los tipos de producto
  public static enum TipoProducto { BASICO, ALIMENTACION, LUJO }
}
```

2. Arquitecturas de dominio – Ejemplo

Identificamos dos problemas principales.

(1) Fuerte acoplamiento con la interfaz
Difícil de usar con otras entradas.

```
public class CalculadorImpuestos {
  public void main(String[] args) {
    Producto producto = leerProductoUI();
    double impuestos = calculadoraIVA(producto.getPrecio(), producto.getTipoProducto());
    mostrarIVA(impuestos);
  }
  // Dado el importe y el tipo de producto calculo el IVA asociado
  public static double calculadoraIVA(double precioSinIVA, TipoProducto tipoProducto) {
    return switch(tipoProducto) {
      case BASICO -> precioSinIVA * 0.04;
      case ALIMENTACION -> precioSinIVA * 0.10;
      case LUJO -> precioSinIVA * 0.21;
    }
  }
  // Enumerado para determinar los tipos de producto
  public static enum TipoProducto { BASICO, ALIMENTACION, LUJO }
}
```

2. Arquitecturas de dominio – Ejemplo

Identificamos dos problemas principales.

```
public class CalculadorImpuestos {
  public void main(String[] args) {
    Producto producto = leerProductoUI();
    double impuestos = calculadoraIVA(producto.getPrecio(), producto.getTipoProducto());
    mostrarIVA(impuestos);
  }
  // Dado el importe y el tipo de producto calculo el IVA asociado
  public static double calculadoraIVA(double precioSinIVA, TipoProducto tipoProducto) {
    return switch(tipoProducto) {
      case BASICO -> precioSinIVA * 0.04;
      case ALIMENTACION -> precioSinIVA * 0.10;
      case LUJO -> precioSinIVA * 0.21;
    }
  }
  // Enumerado para determinar los tipos de producto
  public static enum TipoProducto { BASICO, ALIMENTACION, LUJO }
}
```

(1) Fuerte acoplamiento con la interfaz
Difícil de usar con otras entradas.

(2) Extensibilidad limitada.
Pero si se producen cambios en los tipos de IVA tengo que modificar el código fuente

2. Arquitecturas de dominio – Ejemplo

Como solución al primer problema, creamos la clase `ObtencionIVA` que proveerá los importes adecuados.

```
public class CalculadorImpuestos {
  private ObtencionIVA iva;
  public CalculadorImpuestos(ObtencionIVA iva) {
    this.iva = iva;
  }
}
```

¿Qué tipo de clase es `ObtencionIVA`?

- Es una interfaz, su implementación concreta es la que cambiará si los tipos de IVA se ven afectados.

```
public interface ObtencionIVA {
   double obtenerPara(TipoIVA tipo);
}
```

2. Arquitecturas de dominio – Ejemplo

Como solución al segundo problema, la misma clase **ObtencionIVA** también contiene los valores del IVA a aplicar.

```
public double calcular(double precioSinIVA, TipoProducto tipoProducto) {
   ObtencionIVA.TipoIVA ivaAAplicar = switch (tipoProducto) {
      case BASICO -> ObtencionIVA.TipoIVA.REDUCIDO;
      case ALIMENTACION -> ObtencionIVA.TipoIVA.BASICO;
      case LUJO -> ObtencionIVA.TipoIVA.GENERAL;
    };
    return iva.obtenerPara(ivaAAplicar) * precioSinIVA;
}
```

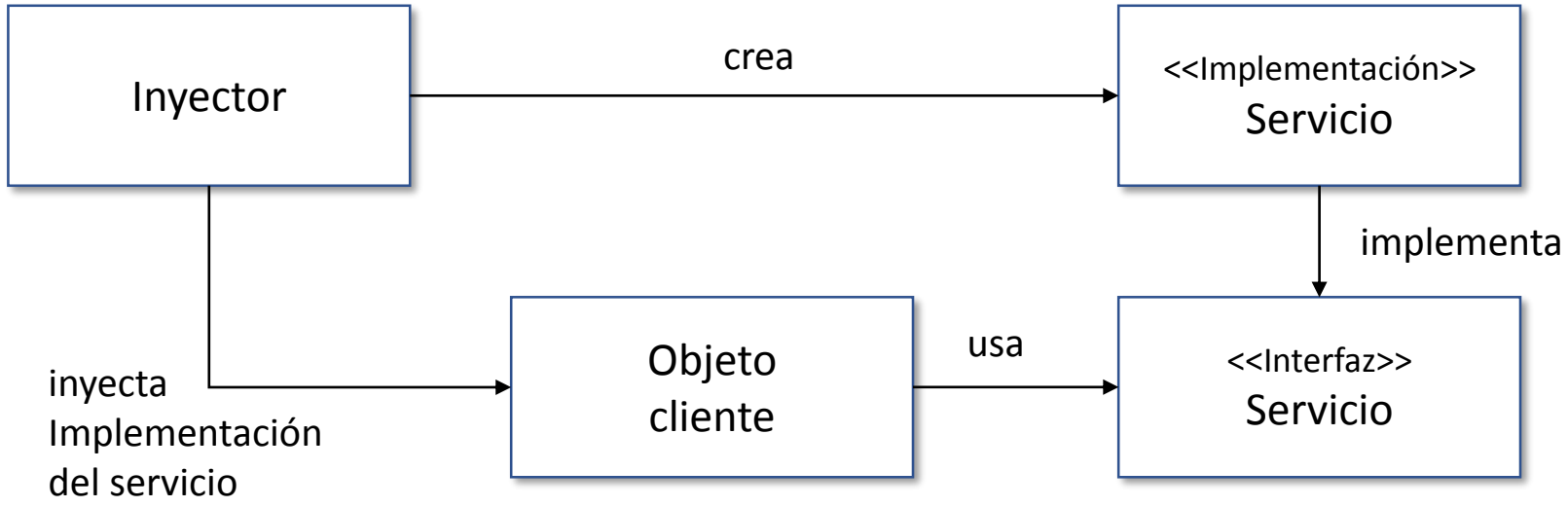
Mi código no implementa el cálculo, lo delega en **ObtenciónIVA.**

3. Inyección de dependencias

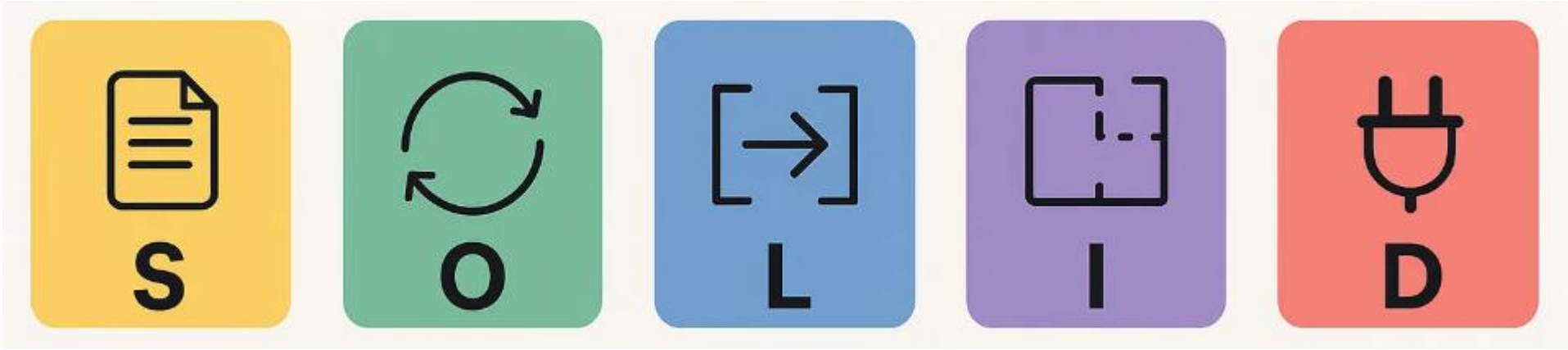
Desacoplar implementaciones

3. Inyección de dependencias – Definición

La inyección de dependencias (DI) es un patrón de diseño en el que un objeto recibe los servicios de los que depende (dependencias) de una fuente externa en lugar de crearlas por sí mismo.



3. Inyección de dependencias – Principios SOLID



Principio de Responsabilidad Única

Una clase debe tener una, y solo una, razón para cambiar.

Principio Abierto/Cerrado

Las entidades de software deben estar abiertas a extensión, pero cerradas a modificación.

Principio de Sustitución de Liskov

Los objetos de una superclase deben poder ser sustituidos por objetos de sus subclases sin alterar el comportamiento esperado del sistema.

Principio de Segregación de Interfaces

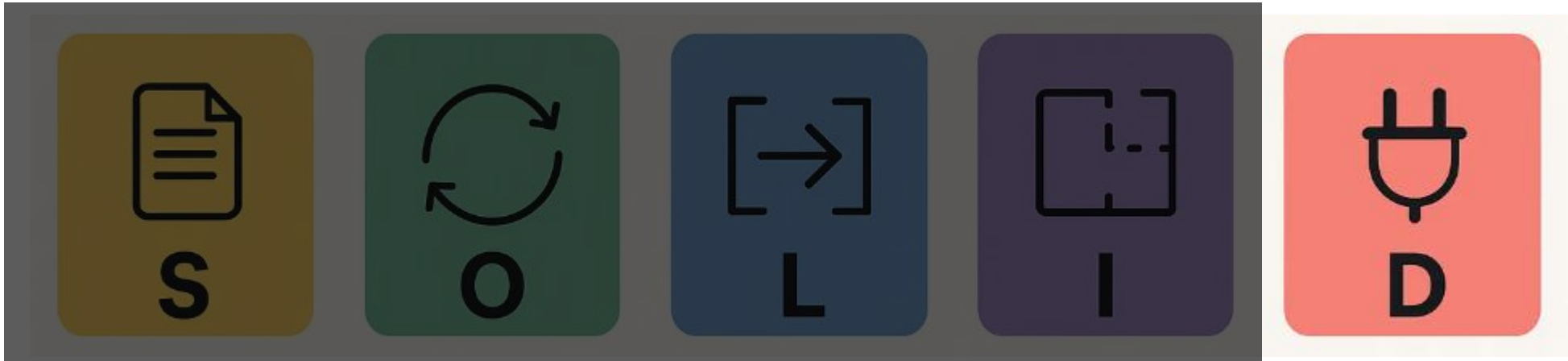
Los clientes no deben depender de interfaces que no utilizan.

Principio de Inversión de Dependencias

Los módulos de alto nivel no deben depender de módulos de bajo nivel, Ambos deben depender de abstracciones.

Las abstracciones no deben depender de detalles ni los detalles deben depender de abstracciones.

3. Inyección de dependencias – Principios SOLID



Principio de Responsabilidad Única

Una clase debe tener una, y solo una, razón para cambiar.

Principio Abierto/Cerrado

Las entidades de software deben estar abiertas a extensión, pero cerradas a modificación.

Principio de Sustitución de Liskov

Los objetos de una superclase deben poder ser sustituidos por objetos de sus subclases sin alterar el comportamiento esperado del sistema.

Principio de Segregación de Interfaces

Los clientes no deben depender de interfaces que no utilizan.

Principio de Inversión de Dependencias

Los módulos de alto nivel no deben depender de módulos de bajo nivel, Ambos deben depender de abstracciones.

Las abstracciones no deben depender de detalles ni los detalles deben depender de abstracciones.

3. Inyección de dependencias – Inversión de control

La Inversión de Control es un principio de diseño que consiste en ceder el control de la creación y gestión de dependencias a un elemento externo.

¿Qué problema resuelve?

El código define qué hace, pero no controla cómo ni cuándo se crean sus dependencias.

- Evita el acoplamiento a implementaciones concretas.
- Facilita el cambio de dependencias (tests, mocks, infraestructura).
- Permite reutilizar la misma lógica en distintos contextos.

Principio de Hollywood

3. Inyección de dependencias – Inversión de control

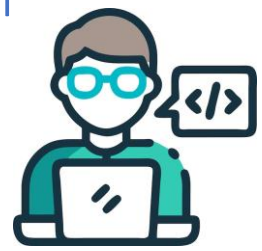
```
public class Aplicacion {
  public static void main (String[] args) {
    Servicio servicio = new Servicio();
    servicio.ejecutar();
  }
}

class Servicio {
  public void ejecutar () {
    System.out.println("Lógica de negocio");
  }
}
```

Problemas

- La función `main` decide qué se crea y cuándo se ejecuta.
- La creación de objetos está acoplada a clases concretas.
- Cambiar una dependencia es modificar el código.
- Dificulta cambiar los objetos en tiempo de ejecución o bien según el entorno.

El flujo de ejecución lo controla el programador



3. Inyección de dependencias – Terminología

Objeto cliente

- El objeto que necesita dependencias.
- Utiliza la **interfaz del servicio** para poder realizar su implementación.

Servicio (*interfaz*)

- El contrato del servicio.

Servicio (*implementación*)

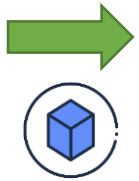
- La implementación concreta del servicio que se quiere inyectar.

Inyector o contenedor de dependencias

- Encargado de crear las implementaciones concretas.
  Usa alguna estrategia para inyectar las dependencias.
- Configurado por la aplicación.

3. Inyección de dependencias – Muy simple

El objeto cliente usa la interfaz de servicio.
Espera recibirlo de alguna forma (ej., en constructor)



```
public class CalculadorImpuestos {
   private final ObtencionIva iva;
   public CalculadorImpuestos(ObtencionIVA iva) {
      this.iva = iva;
   }
}
```

En una capa separada, se crea una implementación de la interfaz de servicio.

```
public class IvaDb implements ObtencionIVA {
   public double obtenerPara(TipoIVA tipo) {
      // conectar a una BD...
   }
}
```

Configuración global, ligando implementaciones de los servicios a los clientes,

```
public static void main(String[] args) {
  ObtencionIVA iva = new IvaDB();
  CalculadorImpuestos calculador =
       new CalculadorImpuestos(iva);
  desplegar(calculador)
}
```

3. Inyección de dependencias – Frameworks

Un **framework de DI** se encarga de:

- Gestionar contenedores de dependencias.
- Permitir el mapeo de interfaces a implementaciones en el contenedor.
- Crear objetos inyectando las dependencias según el contenedor configurado.

Soporte Java a través de anotaciones.

- JSR-330 standard annotations (*Dependency Injection*)
- Implementación concreta por parte de los frameworks.

3. Inyección de dependencias – Frameworks

Ejemplo en varios frameworks

https://github.com/Col-E/Useful-Things/tree/master/tutorials/dependency-injection

google/**guice**

Guice (pronounced 'juice') is a lightweight dependency injection framework for Java 11 and above, brought to you by Google.

77 Contributors · 4k Used by · 10 Discussions · 13k Stars · 2k Forks

vanillasource/**jaywire**

JayWire Dependency Injection

2 Contributors · 7 Used by · 60 Stars · 5 Forks

google/**dagger**

A fast dependency injector for Android and Java.

0 Contributors · 2k Used by · 18k Stars · 2k Forks

3. Inyección de dependencias – Guice

```
// Configuración del contenedor
public class ImpuestosModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(ObtencionIVA.class).
       to(IvaDb.class);
  }
}

// Definición del objeto que tiene
// una dependencia
public class CalculadorImpuestos {
  @Inject
  private final ObtencionIVA iva;

  ...
}
```

Crea el grafo completo de objetos resolviendo dependencias.

```
public static void main(String[] args) {
  Injector inj =
      Guice.createInjector(new ImpuestosModule());

  CalculadorImpuestos calculador =
       inj.getInstance(CalculadorImpuestos.class);

   desplegar(calculador);
}
```

3. Inyección de dependencias – Beneficios

Acoplamiento débil (*loose coupling*)

- Los componentes no dependen de implementaciones concretas.
- Sistema más adaptable puesto que se pueden cambiar las implementaciones
  ej., cambiar el tipo de base de datos.
- Mejor evolución, solo hay que cambiar la configuración del contenedor de DI.

Testability.

- Se pueden inyectar mocks o dependencias “fake”

Reusabilidad.

- Los componentes son realmente independientes.
  Se pueden reutilizar en cualquier contexto que se capaz de proporcionar implementaciones concretas

4. Arquitectura Hexagonal

a.k.a. Arquitectura de Puertos y Adaptadores

4. Arquitectura Hexagonal – ¿Qué es?

- La arquitectura hexagonal es uno de los primeros ejemplos de arquitecturas centradas en el dominio.
- Ideada por Alistair Cockburn.
  - Creada en 2005
  - Originalmente denominada Arquitectura Hexagonal https://wiki.c2.com/?HexagonalArchitecture
  - Nombre actual: **Puertos y adaptadores**

Objetivo básico

Desacoplar la lógica de negocio de la tecnología.

4. Arquitectura Hexagonal – Objetivos

Diseñar la aplicación para:

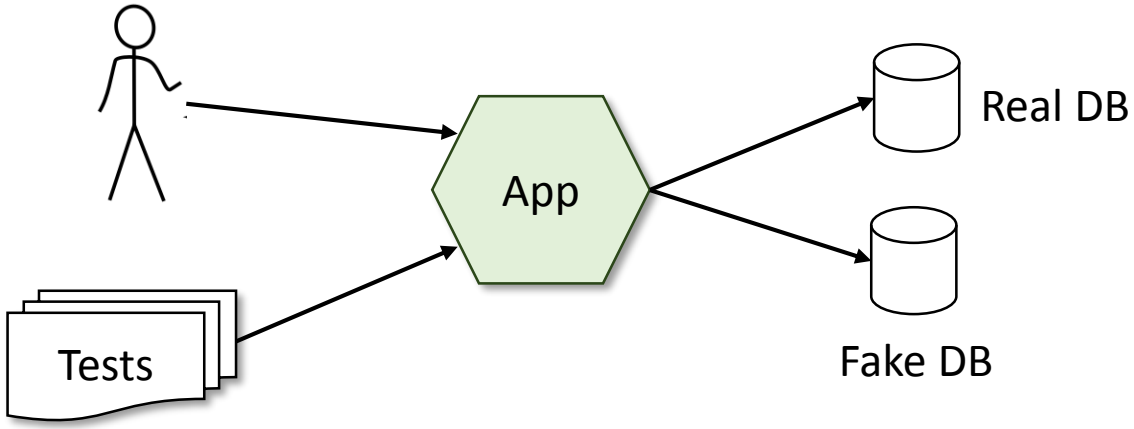
- Ejecutarse sin interfaz de usuario.
- Ejecutarse sin base de datos.
- Ser controlada por tests automáticos o scripts.
- Permitir cambiar tecnologías sin tocar el dominio.
- Integrarse con otros sistemas sin intervención humana.

> *Create your application to work without either a UI or a database so you can run automated regression-tests against it, change connected technologies, protect it from leaks between business logic and technologies, work when the database becomes unavailable, and link applications together without any user involvement.*

4. Arquitectura Hexagonal – Objetivos

Permitir que una aplicación tenga:

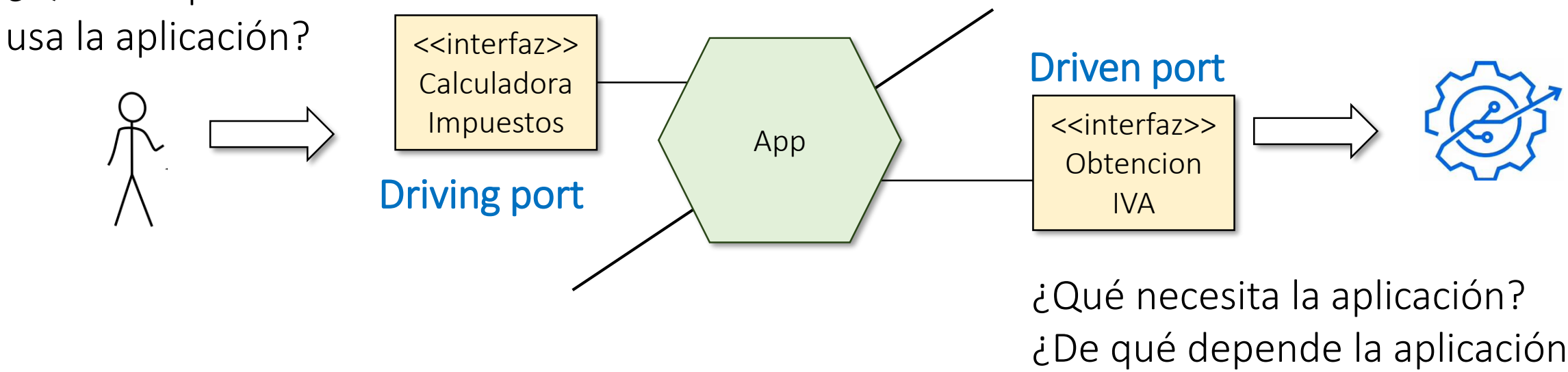
- Múltiples formas de uso:
  usuarios, programas, tests, batch.
- Independencia del entorno.
- Desarrollo y pruebas en aislamiento.
- Misma lógica, distintos contextos.



> *Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.*

4. Arquitectura Hexagonal – Elementos básicos

- Independizar el núcleo de la aplicación del resto.
- Se consideran dos roles.
  - Quién usa la aplicación (*driving*)
  - Qué necesita la aplicación (*driven)*



4. Arquitectura Hexagonal – Conceptos

Una interfaz **(interface)** es un contrato que debe satisfacer aquel que implemente la interfaz.
Normalmente la interfaz se define como un conjunto de métodos.

Una interfaz es una herramienta del lenguaje.

- Nos permite definir un contrato entre dos partes.
- En arquitecturas de dominio, las interfaces no solo son técnicas: asumen un rol arquitectónico.

4. Arquitectura Hexagonal – Conceptos

Una interfaz **(interface)** es un contrato que debe satisfacer aquel que implemente la interfaz. Normalmente la interfaz se define como un conjunto de métodos.

Una "provided interface" (*también API*) es una interfaz que define servicios ofrecidos por la aplicación.

Una "required interface" (*también SPI*) es una interfaz que define servicios que la aplicación necesita para funcionar.

```
public interface CalcularImpuestos {
  public double impuestoDe (Factura f);
}
```

CalcularImpuestos

```
public interface ObtenerIVA {
    public double ivaProducto(TipoProducto p);
}
```

ObtenerIVA

4. Arquitectura Hexagonal – Conceptos

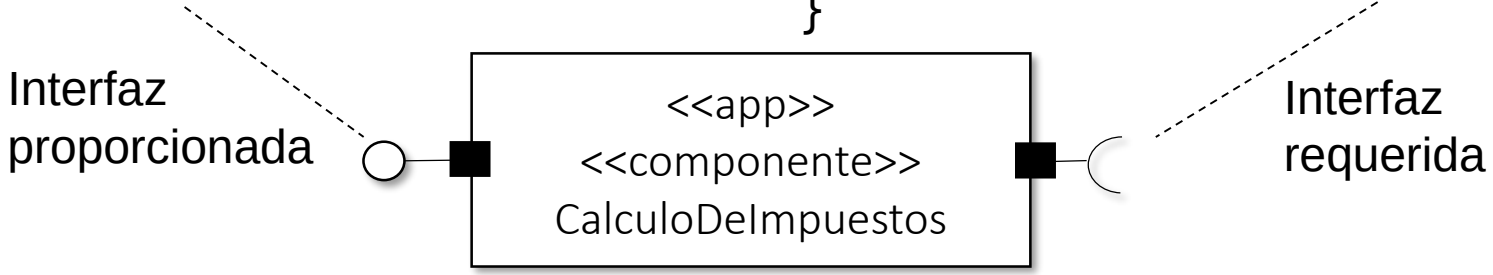
Un puerto **(port)** declara una o más interfaces definidas por la aplicación.
El puerto es donde la aplicación expone la interfaz con la intención de establecer una conversación entre un actor externo y la aplicación.

Un "driving port" es un puerto usado por un actor o un adaptador para controlar la aplicación (ej., una UI)

Un "driven port" es un puerto usado por la aplicación para realizar peticiones a actores externos, a través de un adaptador.

```
public interface CalcularImpuestos {
  public double impuestoDe (Factura f);
}
```

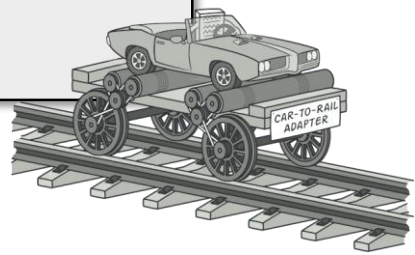
```
public interface ObtenerIVA {
    public double ivaProducto(TipoProducto p);
}
```



4. Arquitectura Hexagonal – Conceptos

Un adaptador **(adapter)** es la implementación necesaria para que las interfaces "encajen" en los actores que dirigen o son dirigidos por la aplicación.

Esencialmente el **patrón adaptador**
se implementa la interfaz para realizar una traducción entre los valores esperados por la interfaz y los usados por los usuarios del adaptador.

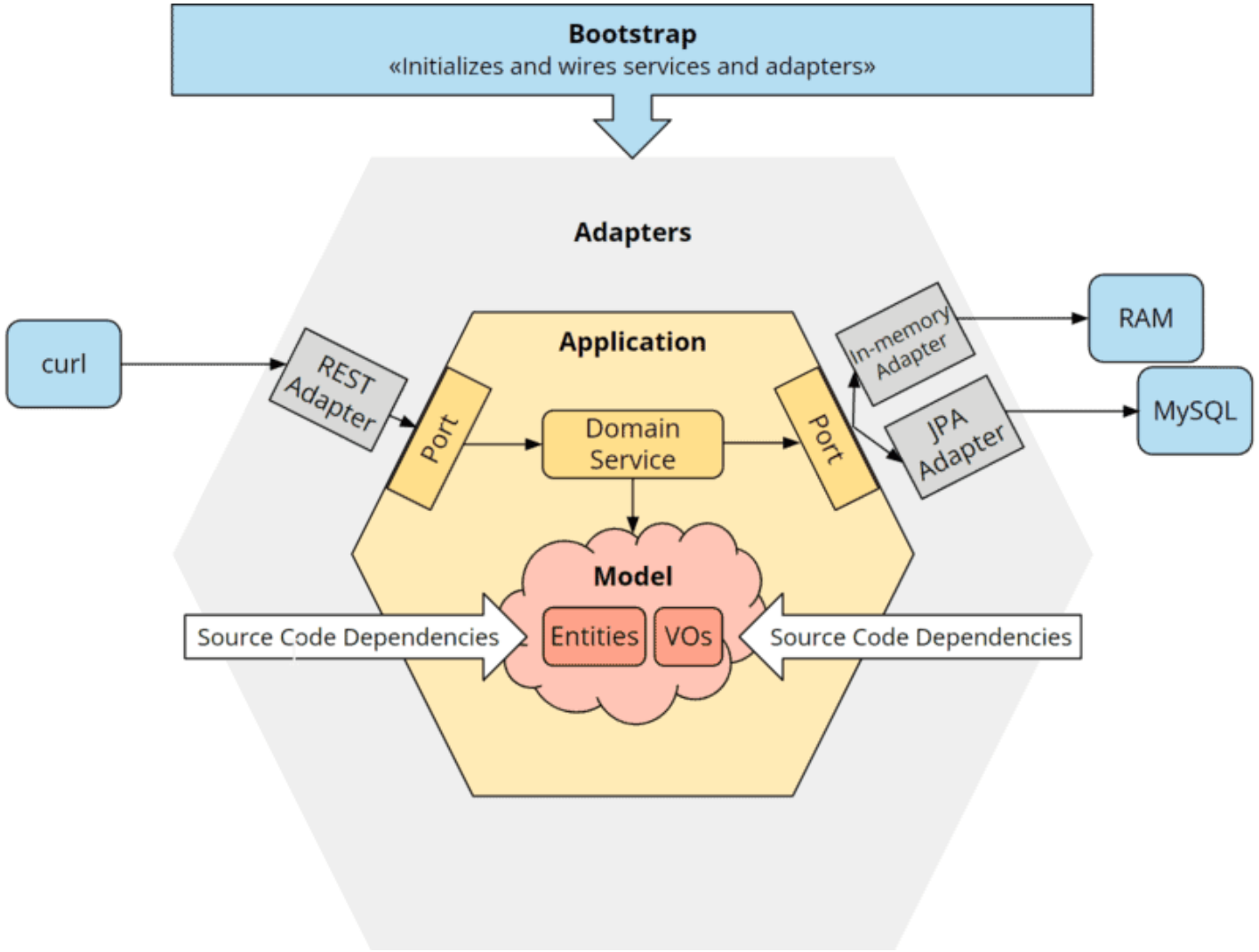



```
@Inject CalcularImpuestos servicio

app = new CalculoDeImpuestos(...);
servicio = app.getServicioCalculo();
...
Factura f = construirFactura(datosEntrada);
double impuestos = servicio.impuestoDe(f);
```

```
public class IvaWEB
  implements ObtenerIVA {
  public double ivaProducto(TipoProducto p) {
     // Traducir p a reducido, normal, etc.
     // Invocar servicio web
  }
}
```

4. Arquitectura Hexagonal – Todo junto



https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture-java/

4. Arquitectura Hexagonal – Todo junto

- Distinción “*lo de dentro*” vs. “*lo de fuera*”.
  La aplicación publica puertos para describir cómo interacciona con el mundo.
- Eventos llegan desde “*el mundo de fuera*”.
  - Los eventos se convierten a mensajes que entiende la aplicación.
  - Un puerto describe cómo es este protocolo de comunicación.
  - Esta conversión la realizan adaptadores.
    Servicios web, test drivers, …
- Cuando la aplicación tiene que enviar algo o guardar datos, lo hace a través de un puerto.
  - El puerto es independiente de la tecnología.

4. Arquitectura Hexagonal – ¿Quién implementa las interfaces?

- Required interfaces (o APIs)
  - Implementadas por la aplicación.
  - La API es una interfaz de cara a aquellos que usan la aplicación.
  - Ojo, no estamos diciendo API REST, esa es una capa adicional que añadimos y que se conecta a la API de nuestra aplicación (está fuera del hexágono)
  - ¡Importante! Los adaptadores son los elementos que usan la aplicación.
- Provided interfaces (o SPIs)
  - Implementadas por adaptadores dedicados.

4. Arquitectura Hexagonal – Todo junto - Ensamblaje

Debe existir un **artefacto que ensamble el sistema**:

1. Crear los adaptadores para puertos “driven”.
2. Instanciar la aplicación y conectarlo los adaptadores “driven”.
3. Crear el adaptador “driving” y pasarle la aplicación configurada.
   Conceptualmente es un adaptador pero no implementa ninguna interfaz, la usa

```
public void main(String[] args) {
  // (1) Construir el adaptador para el puerto ObtenerIVA
  var calculoIva = new IVAWeb();

  // (2) Construir la aplicación y conectarle el adaptador
  var app = new CalculoDeImpuestos(calculoIVA);

  // (3) Construir la GUI que dirige la aplicación
  var gui = new GUI(app);
  gui.start();
}
```

4. Spring Boot

Servicios web con convención sobre configuración

5. Spring Boot – Framework

- Spring
  - Contenedor de inyección de dependencias
  - Framework de AOP
  - Gestión de transacciones
  - Framework web

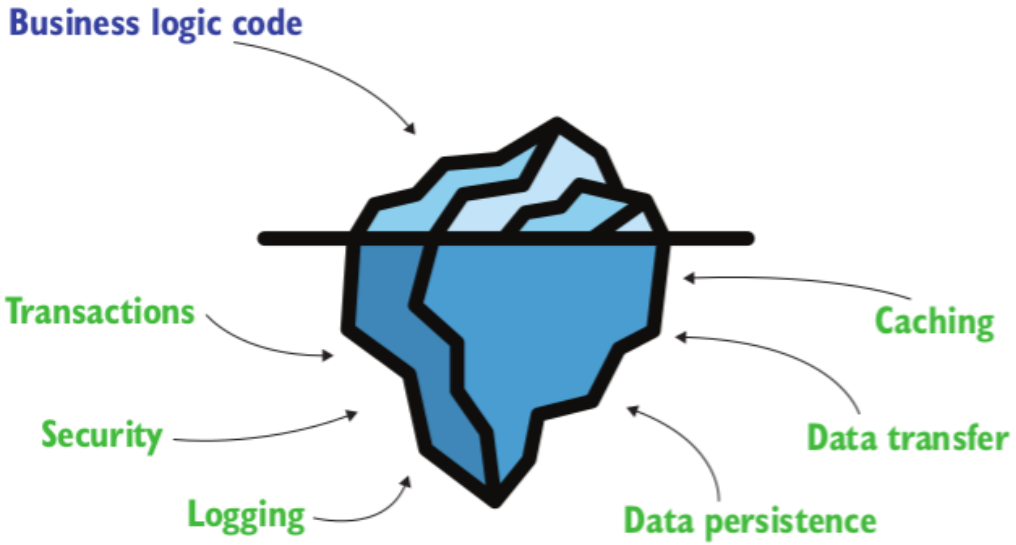
Muy flexible, pero
Configuración tediosa

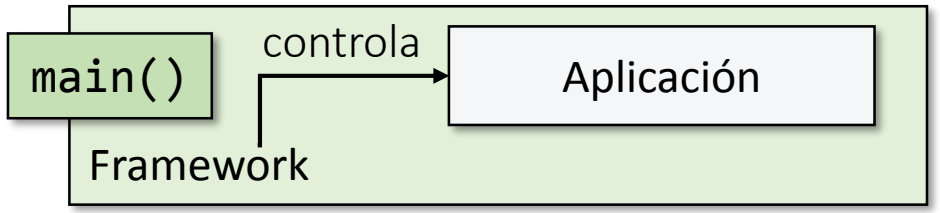
- Spring Boot
  - Framework para la creación de aplicaciones Spring
  - Reduce la configuración manual
  - Selecciona configuraciones básicas a partir de dependencias
  - Servidor web embebido
  - Muy usado para APIs REST y aplicaciones backend

Convención sobre
configuración

5. Spring Boot – Framework

- Framework
  - Proporciona la estructura para crear cierto tipo de aplicaciones.
  - El desarrollador configura y “rellena los huecos” según las reglas del framework.
  - El framework automatiza la ejecución e invoca al código creado por el desarrollador.
    - ¡Inversión de control!

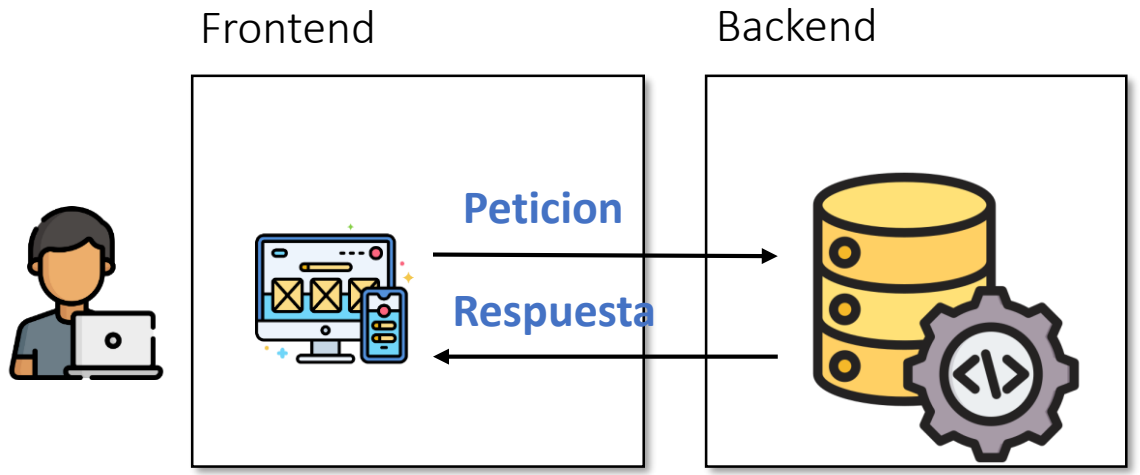




*Spring Starts Here. 2021. Laurenţiu Spilcă

5. Spring Boot – Aplicaciones web

- División *frontend – backend*
- Arquitectura cliente-servidor
- Frontend: parte cliente
- Backend: parte servidor



- Interacción
  - Comunicación iniciada por el frontend
  - Frontend envía peticiones
  - Backend responde

5. Spring Boot – Conceptos básicos

- **Bean** es un objeto gestionado por el contenedor de Spring.
  - Spring se encarga de su creación, configuración y ciclo de vida.
  - Se define mediante anotaciones como @Component, @Service, @Repository o @Bean.
- **@Configuration**
  - Indica que una clase contiene configuración de Spring.
  - Se usa para definir beans manualmente con métodos anotados con @Bean.
  - Spring resuelve las dependencias “mirando” los tipos de los parámetros y tipo de retorno de los métodos anotados con @Bean

5. Spring Boot – Ejemplo

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

@Configuration
public class CalculoIvaConfig {
  @Bean
  ObtencionIVA obtencionIVA(ObjectMapper objectMapper) {
     return new IvaEuRatesAdapter(objectMapper);
  }

  @Bean
  CalculadorImpuestos calculadorImpuestos(ObtencionIVA obtencionIVA) {
     return new CalculadorImpuestos(obtencionIVA);
  }
}
```

Declaración de una dependencia

5. Spring Boot – Conceptos básicos

- @RestController
  - Marca una clase como controlador REST (servicio web)
  - Los métodos definirán *endpoints* de la API REST
  - Los endpoints transforman el valor de retorno en datos en JSON directamente
- @GetMapping
  - Asocia una URL y el método HTTP GET a un método Java.
  - Un método GET en HTTP es un endpoint para devolver información

5. Spring Boot – Ejemplo

```java
@RestController
public class CalculoIvaController {

  private final CalculadorImpuestos calculadorImpuestos;

  public CalculoIvaController(CalculadorImpuestos calculadorImpuestos) {
    this.calculadorImpuestos = calculadorImpuestos;
  }

  @GetMapping("/api/calculo-iva")
  public CalculoIvaResponse calcularIva(@RequestParam double precio,
                                        @RequestParam TipoProducto producto) {
    double importeIVA = calculadorImpuestos.calcular(precio, producto);
    return new CalculoIvaResponse(precio, producto.name(), importeIVA);
  }
}
```

Depende de este puerto
El adaptador se crea en la configuración

Endpoint GET /api/calculo-iva

Conversión automática a JSON

5. Spring Boot – Ejemplo

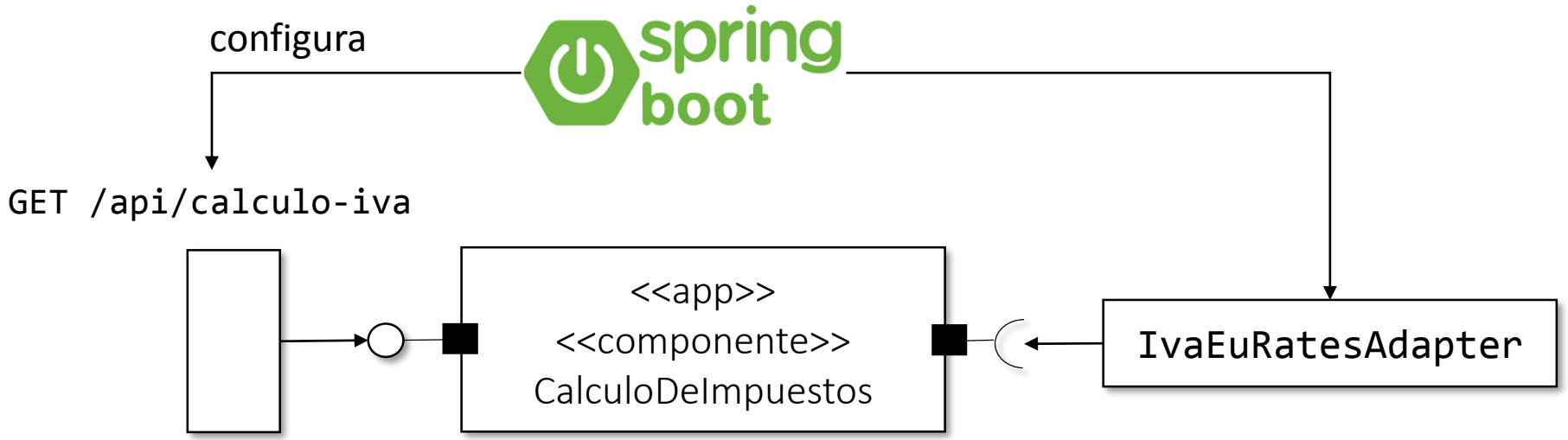
```
curl "http://localhost:8080/api/calculo-iva?precio=10&producto=BASICO"
```

```
@GetMapping("/api/calculo-iva")
public CalculoIvaResponse calcularIva(@RequestParam double precio,
                                      @RequestParam TipoProducto producto)
```

```
{
  precioSinIVA: 100.0,
  tipoProducto: “BASICO”,
  importeIVA: 110.0
}
```

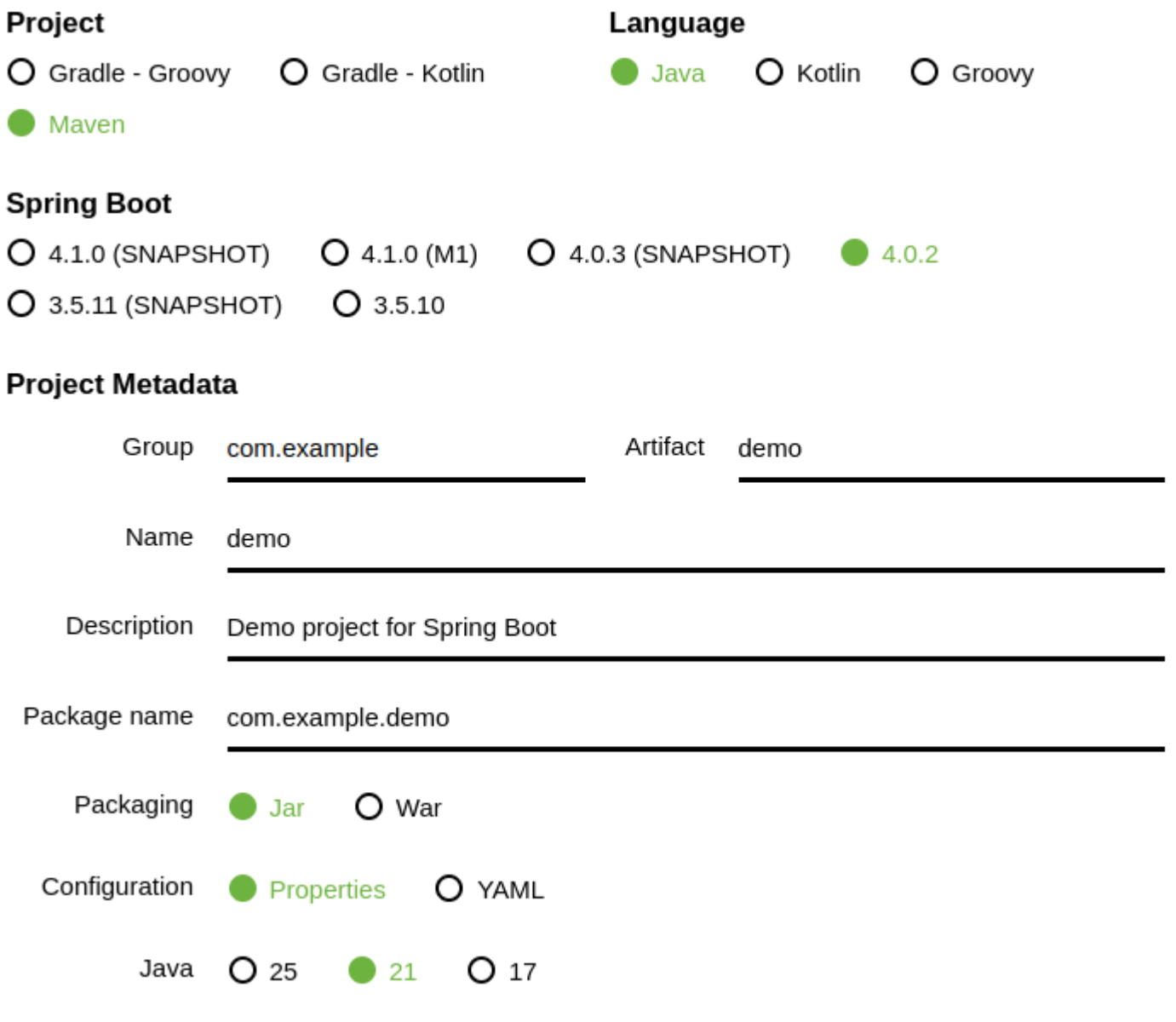
5. Spring Boot – Arquitectura Hexagonal

- Spring Boot actua como *driving adapter*
  - Maneja los puertos *driving* o de entrada utilizando la interfaz proporcionada
  - El controlador REST típicamente hace de adaptador, invocando a los servicios
- Spring Boot configura los *driven ports*
  - Utiliza @Configuration y @Bean para obtener configurar la aplicación con las implementaciones concretas



5. Spring Boot – Inicialización

- https://start.spring.io/



Bibliografía

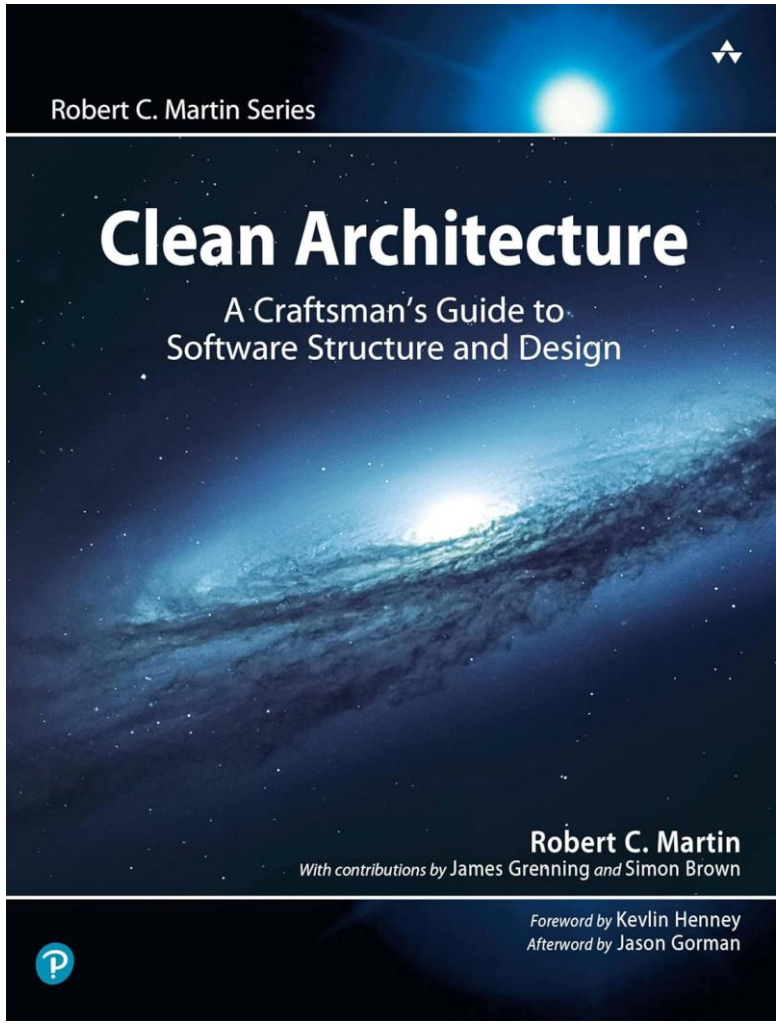
- Hexagonal Architecture Explained. 2025. Alistair Cockburn, José Manuel Garrido de Paz.
- Hexagonal Architecture in Java. https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture-java/
- Hexagonal Architecture. Marco Tulio Valente.
  - https://softengbook.org/articles/hexagonal-architecture

5. Arquitectura limpia

5. Arquitectura limpia – ¿Qué es?

Arquitectura ideada por Robert C. Martin.

- Mismo **objetivo.**
  Proteger el dominio.
- Misma **regla.**
  Dependencias hacia dentro.
- ¿Qué cambia entonces?
  Como organizar el código (*pero no los principios*).



5. Arquitectura limpia – Motivación

En muchos sistemas:

- El dominio depende de frameworks como Spring.
- Los casos de uso dependen de la base de datos.
- Cambiar la UI obliga a tocar lógica de negocio.

Las decisiones importantes del negocio no deben depender de decisiones técnicas.

5. Arquitectura limpia – ¿Qué aporta?

Da una **estructura más explícita en capas**.

Separa **reglas de negocio**:

- Reglas de la empresa (entidades)
- Reglas de la aplicación (casos de uso)
- Facilita entender *quién puede depender de quién*.

> *El código debe organizarse para proteger las reglas de negocio, no para acomodar frameworks, bases de datos o interfaces.*
>
> *Robert C. Martin (Uncle Bob)*

5. Arquitectura limpia – Tipos de reglas

Reglas de negocio de la empresa (estables)

- Entidades
- Conceptos del dominio
- Reglas de la aplicación

Casos de uso (Cambian más, pero siguen siendo negocio)

- Orquestan el dominio

Detalles técnicos (Volátiles)

- UI
- Base de datos
- Frameworks

5. Arquitectura limpia – Arquitectura