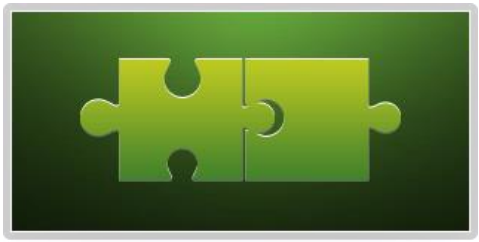
Formación
BDI



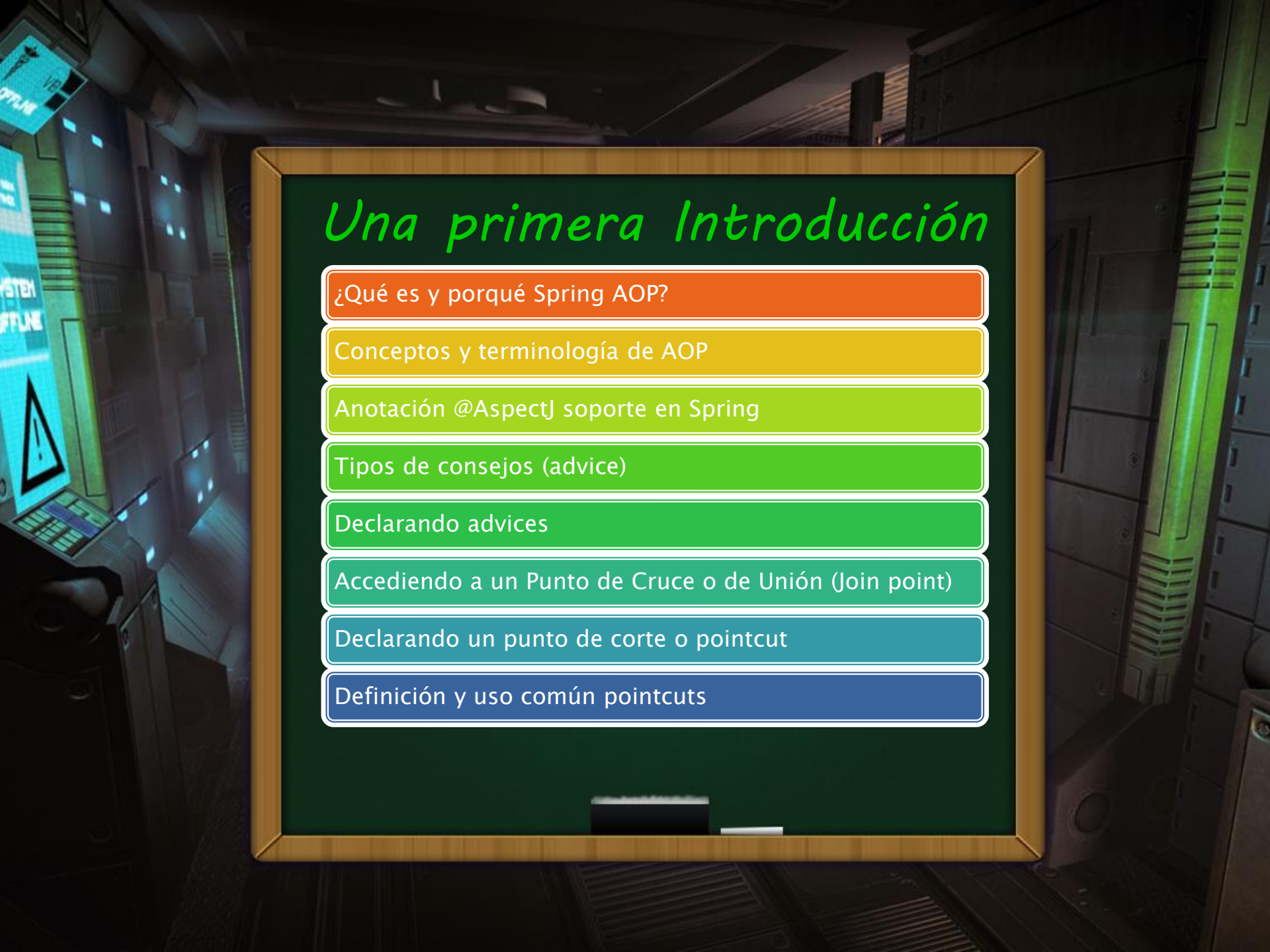
Curso Spring Framework

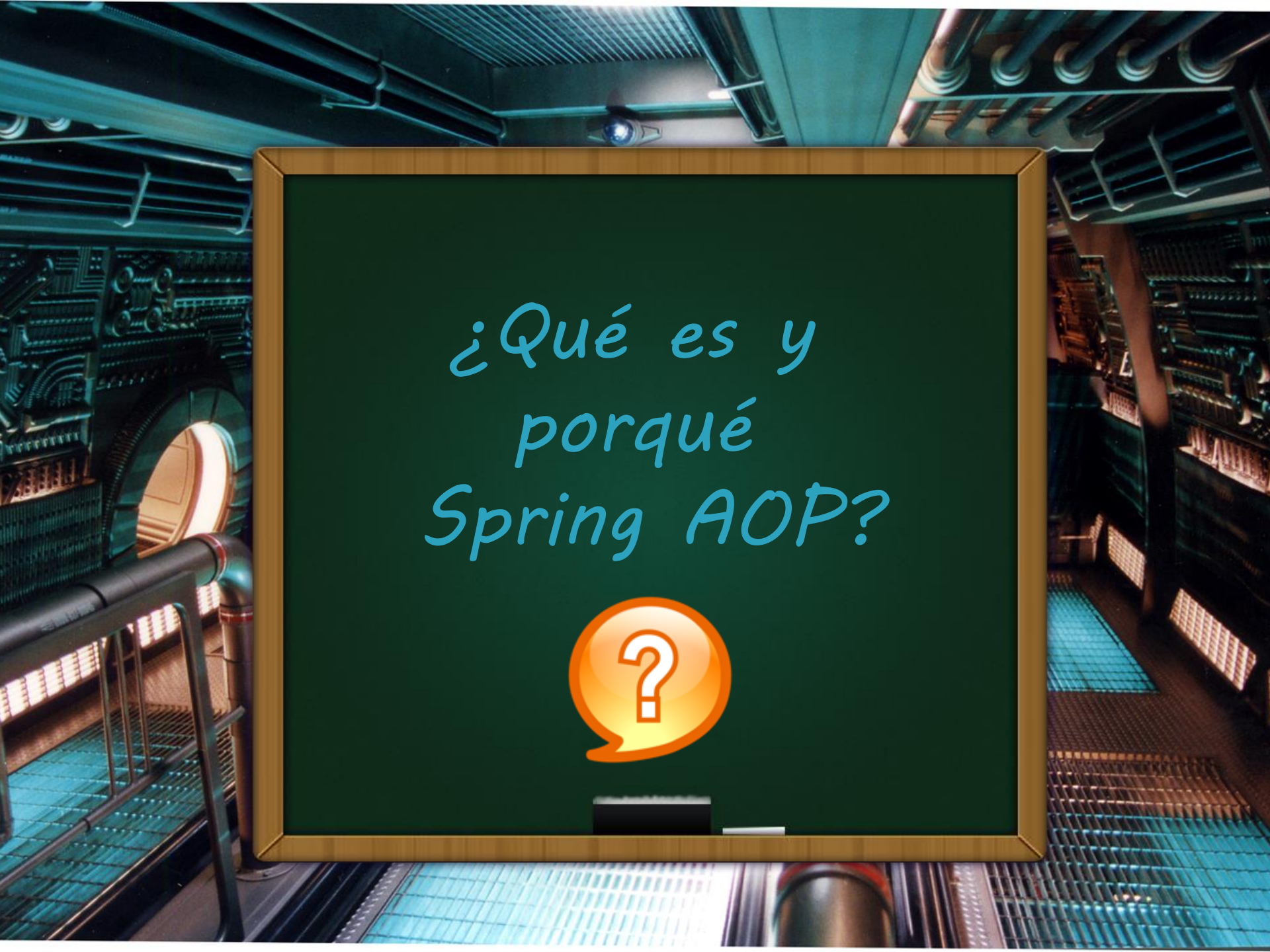
Módulo 7

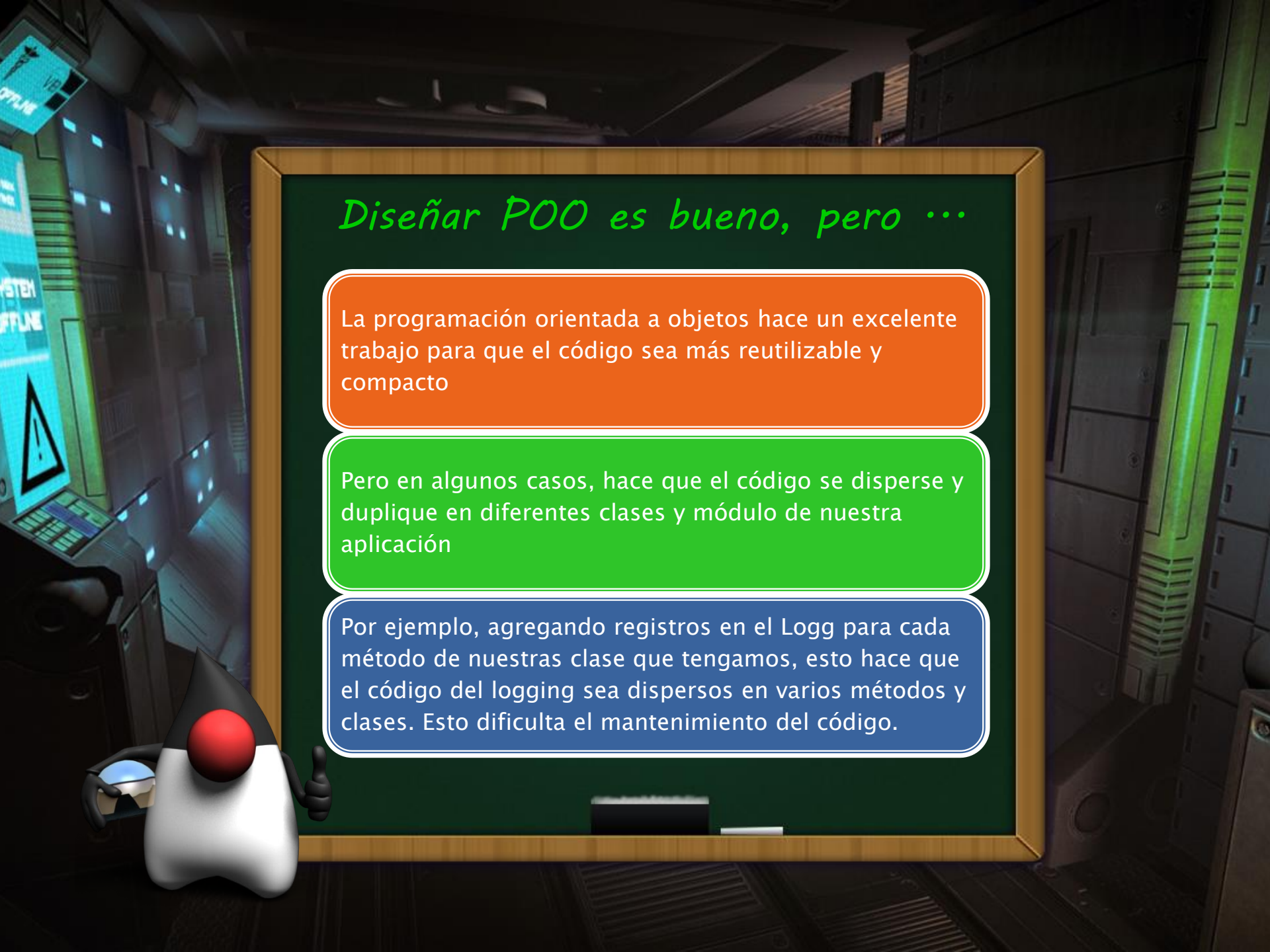
Programación Orientada a Aspectos – AOP

spring

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com


Una primera Introducción
¿Qué es y porqué Spring AOP?
Conceptos y terminología de AOP
Anotación @AspectJ soporte en Spring
Tipos de consejos (advice)
Declarando advices
Accediendo a un Punto de Cruce o de Unión (Join point)
Declarando un punto de corte o pointcut
Definición y uso común pointcuts


¿Qué es y
porqué
Spring AOP?


Diseñar POO es bueno, pero ...
La programación orientada a objetos hace un excelente trabajo para que el código sea más reutilizable y compacto
Pero en algunos casos, hace que el código se disperse y duplique en diferentes clases y módulo de nuestra aplicación
Por ejemplo, agregando registros en el Logg para cada método de nuestras clase que tengamos, esto hace que el código del logging sea dispersos en varios métodos y clases. Esto dificulta el mantenimiento del código.

Problemas que no resuelve la POO

Dispersión de Código

- Mismas tareas o procesos que se propagan y repiten a través de diferentes módulos y/o clases

Enredos de Código o espaguetis

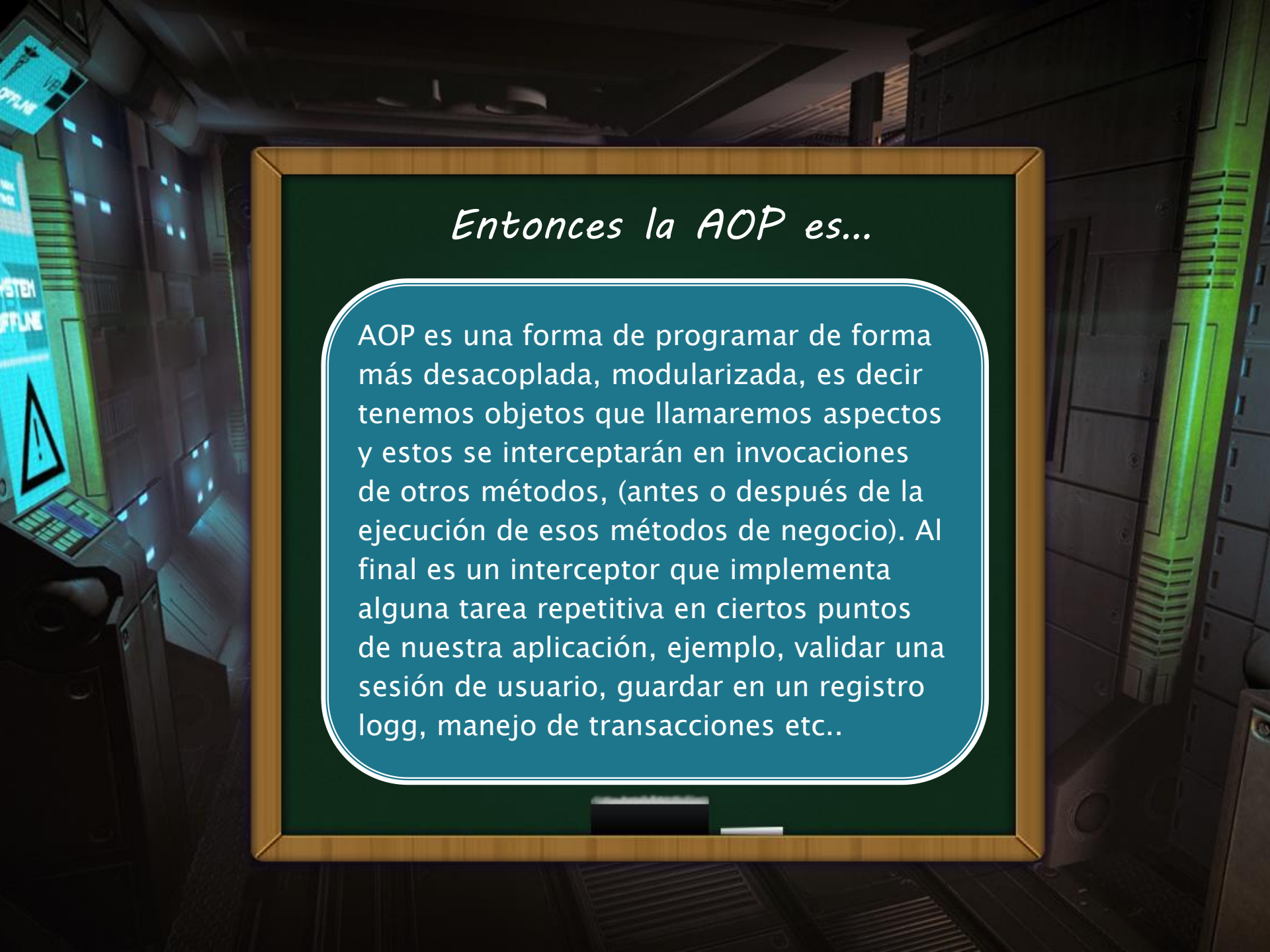
- El acoplamiento de diferentes tareas (por ejemplo, el manejo de logging podría mezclarse con código de nuestra lógica de negocio o seguridad, sumado al manejo de transacción)

AOP es la solución

La Programación Orientada Aspecto (AOP) se ocupa de estas tareas que son transversales a nuestro código y lógica de negocio

En lugar de agregar el código de registro logging en cada uno de los métodos, es mejor delegar esta funcionalidad a un contenedor AOP y definir aquellos métodos que requieren aplicar el logg.

Luego, el contenedor AOP se encarga de invocar el aspecto de logging cuando se llaman los métodos


Entonces la AOP es...
AOP es una forma de programar de forma más desacoplada, modularizada, es decir tenemos objetos que llamaremos aspectos y estos se interceptarán en invocaciones de otros métodos, (antes o después de la ejecución de esos métodos de negocio). Al final es un interceptor que implementa alguna tarea repetitiva en ciertos puntos de nuestra aplicación, ejemplo, validar una sesión de usuario, guardar en un registro logg, manejo de transacciones etc..

AOP y POO

Programación orientada a aspecto (AOP) complementa la Programación orientada a objetos (POO), proporcionando otra manera de pensar acerca de la estructura del programa.

Aspectos permiten la modularización de procesos tales como la gestión de transacciones que abarcan varios tipos y objetos, los mismo sucede con registros logg

Resumen: Beneficios AOP

- Complementa la programación POO.
- Código limpio y modular sin código de dispersión y sin problemas de código espaguetis (mezclas)
- Los Aspectos son configurables, se pueden agregar o quitar según sea necesario, sin necesidad de modificar el código java.
- El código de nuestra lógica de negocio y objetos de dominio no están amarrados a la API o framework, son independientes y desacoplados

Conceptos AOP

Preocupaciones transversales (Cross-cutting)

Funcionalidades presentes y necesarias a traves de toda la aplicación

Ejemplos de preocupaciones transversales:

- Registro Logg
- Gestión de transacciones
- Seguridad
- Bloqueo (Locking)
- Manejo de eventos

Concepto: Aspects (Aspectos)

Un Aspects es una modularización de una preocupación (tarea o proceso) que corta a través de múltiples objetos y métodos, un interceptor.

Concepto: Join point (Punto de Cruce o Unión)

Un Join point es un punto en la ejecución del programa, ejemplo en una invocación de método o manejo de una excepción

En Spring AOP, un join point representa una ejecución de un método.

Concepto: Advice (Consejo o asesoramiento)

Código que se ejecuta en un particular punto de unión (joinpoint)

Tipos de consejos:

1.- antes/before advice, se ejecuta antes del punto de unión (joinpoint)

2.- después/after advice, se ejecuta después del punto de unión (joinpoint)

3.- alrededor/around advice, se ejecuta alrededor del punto de unión (joinpoint)

Concepto: Pointcuts (Puntos de Corte)

Un punto de corte (pointcut) es una expresión que agrupa uno o más puntos de unión (join points)

Un Pointcuts puede estar compuesto en base a relaciones complejas definidas por expreciones regulares (patrón), para restringir y afinar aún más cuando se deba ejecutar un advice o consejo

Concepto: Pointcuts (Puntos de Corte)

Mediante el uso de puntos de corte (pointcut), podemos obtener un control muy preciso sobre la forma de aplicar el consejo (advice) en nuestros componentes o clases, ejemplo

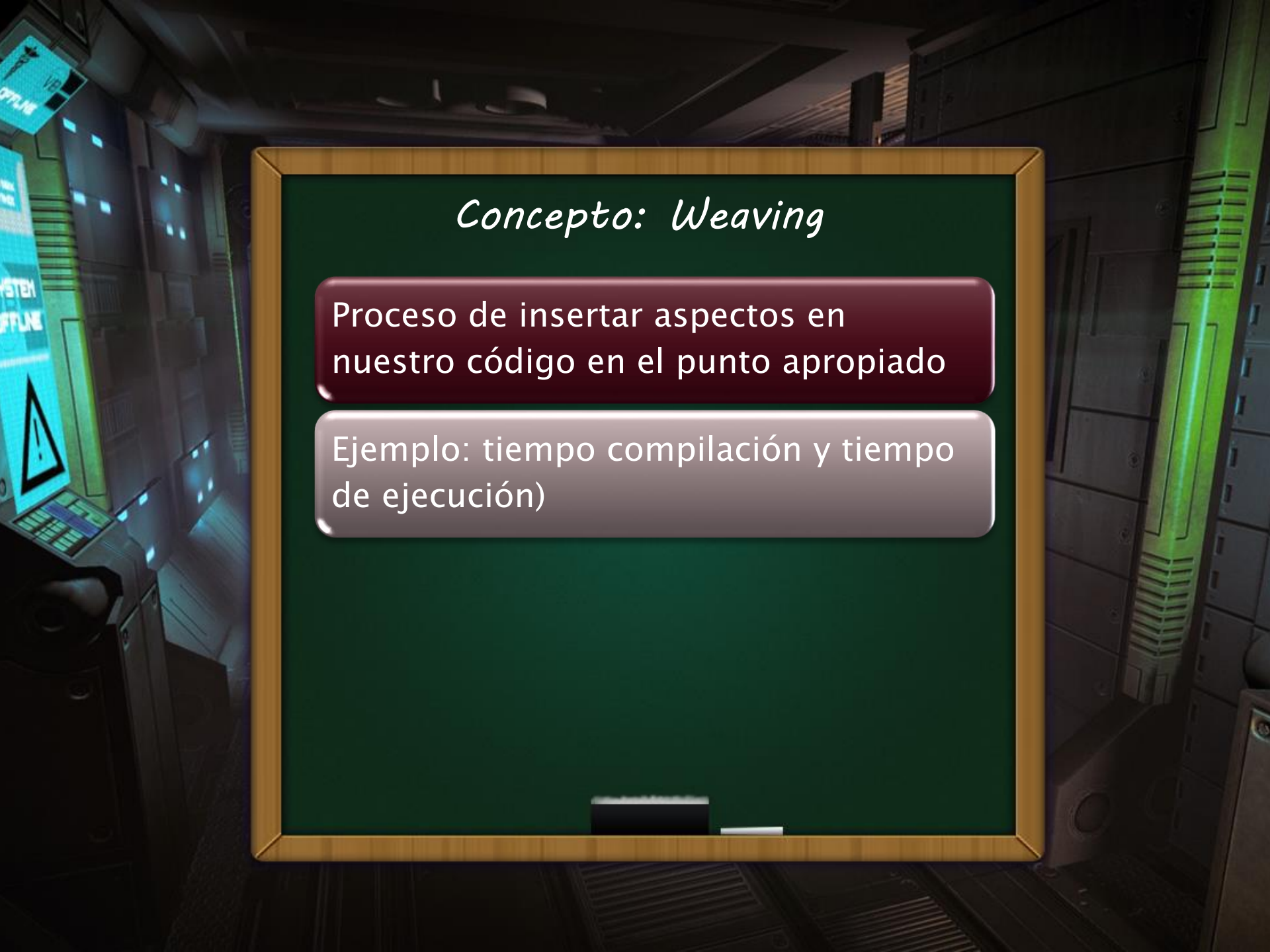
- Un típico joinpoint es una invocación de método.
- Un típico pointcut es una agrupación de todas las invocaciones de métodos de una clase en particular

Concepto: Target (objetivo)

Un objeto cuyo flujo de ejecución es modificado por algún proceso AOP, en otras palabras es interceptado por un aspecto

También se les conoce con el nombre de objetos aconsejados/asesorados (advised object)

Corresponde al objeto o método en cuestión que será intereptado por el aspecto


Concepto: Weaving
Proceso de insertar aspectos en nuestro código en el punto apropiado
Ejemplo: tiempo compilación y tiempo de ejecución)

Concepto: Introduction

Proceso mediante el cual se puede modificar o manipular la estructura de un objeto mediante la introducción de métodos o atributos adicionales, e incluso la definición misma de la clase

Por ejemplo, podemos usar la Introducción para hacer que un determinado objeto implemente una interfaz específica sin necesidad de implementarla explícitamente en la definición de la clase

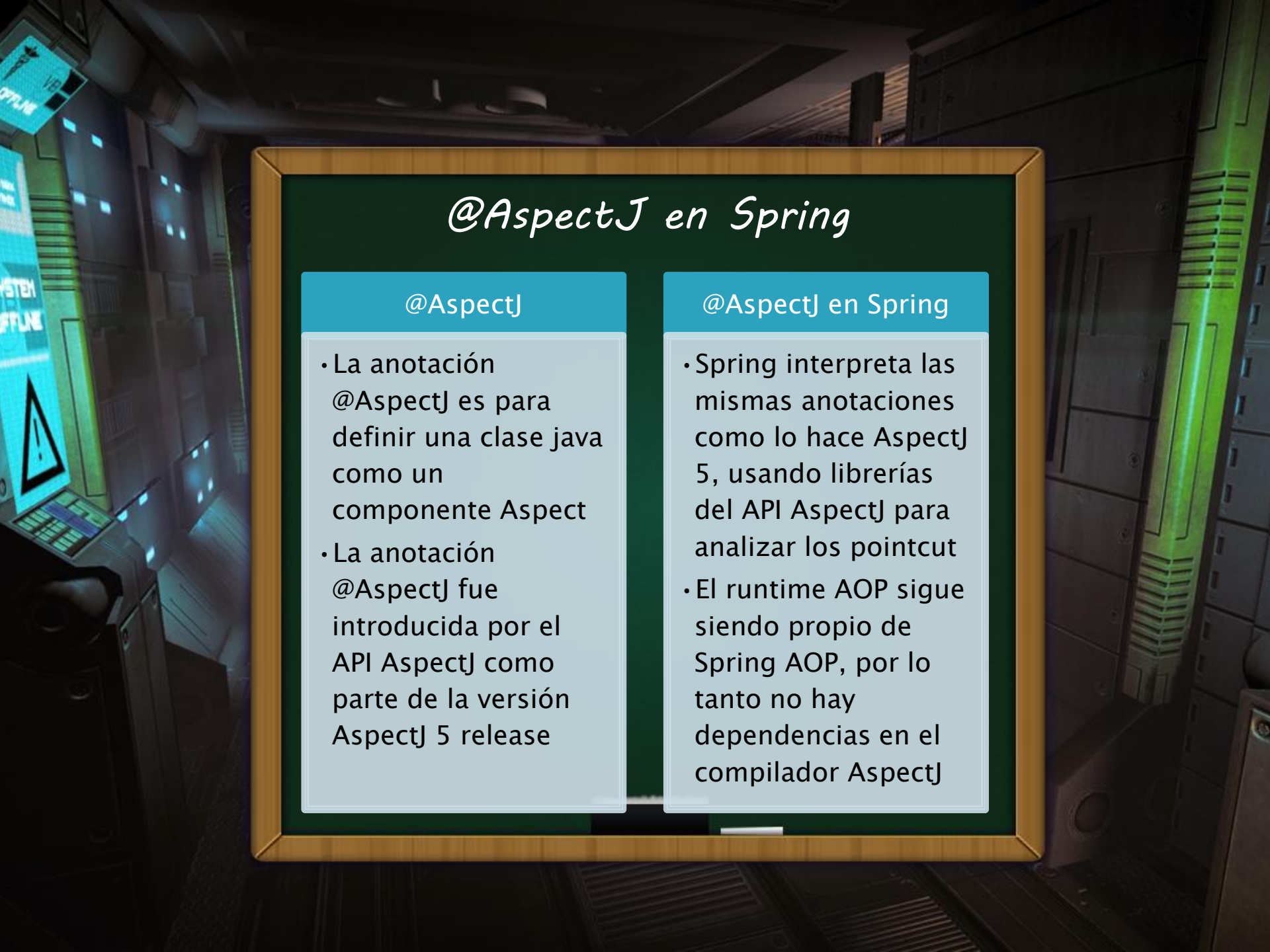

Spring AOP
Capacidades y
Objetivos

Spring AOP

Spring AOP es implementado en puro código Java, no hay necesidad de un proceso de compilación en especial

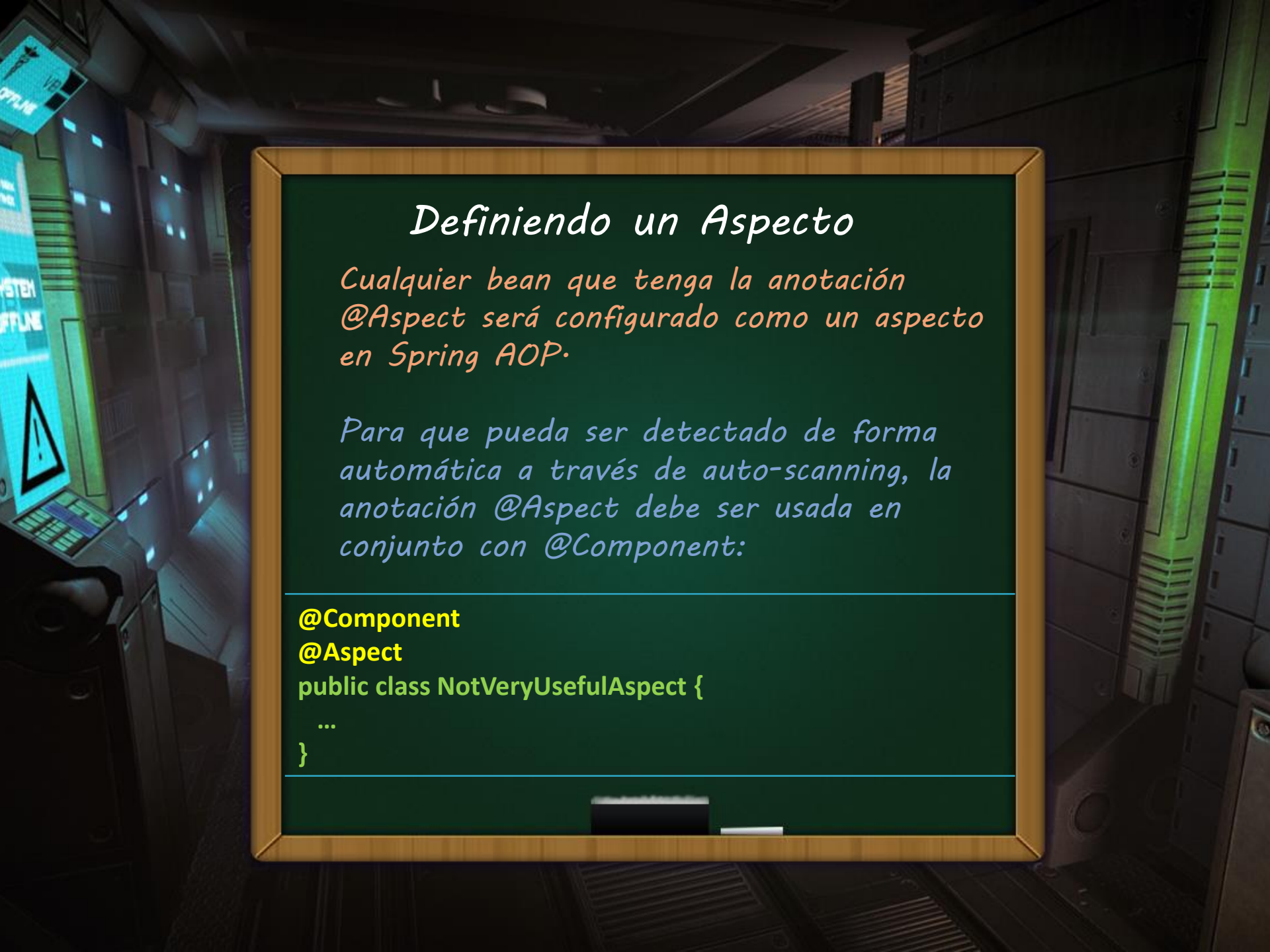
Spring AOP actualmente soporta sólo ejecución de métodos join points (asesoran (advising) la ejecución de métodos sobre los beans de Spring, para agregar funcionalidades transversales a nuestro código)


Soporte
@AspectJ


@AspectJ en Spring
@AspectJ
• La anotación @AspectJ es para definir una clase java como un componente Aspect
• La anotación @AspectJ fue introducida por el API AspectJ como parte de la versión AspectJ 5 release
@AspectJ en Spring
• Spring interpreta las mismas anotaciones como lo hace AspectJ 5, usando librerías del API AspectJ para analizar los pointcut
• El runtime AOP sigue siendo propio de Spring AOP, por lo tanto no hay dependencias en el compilador AspectJ

Configuración @AspectJ en Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <aop:aspectj-autoproxy />

...
</beans>
```


Definiendo un Aspecto
Cualquier bean que tenga la anotación @Aspect será configurado como un aspecto en Spring AOP.
Para que pueda ser detectado de forma automática a través de auto-scanning, la anotación @Aspect debe ser usada en conjunto con @Component:
@Component
@Aspect
public class NotVeryUsefulAspect {
...
}

Tipos de Advice en Spring

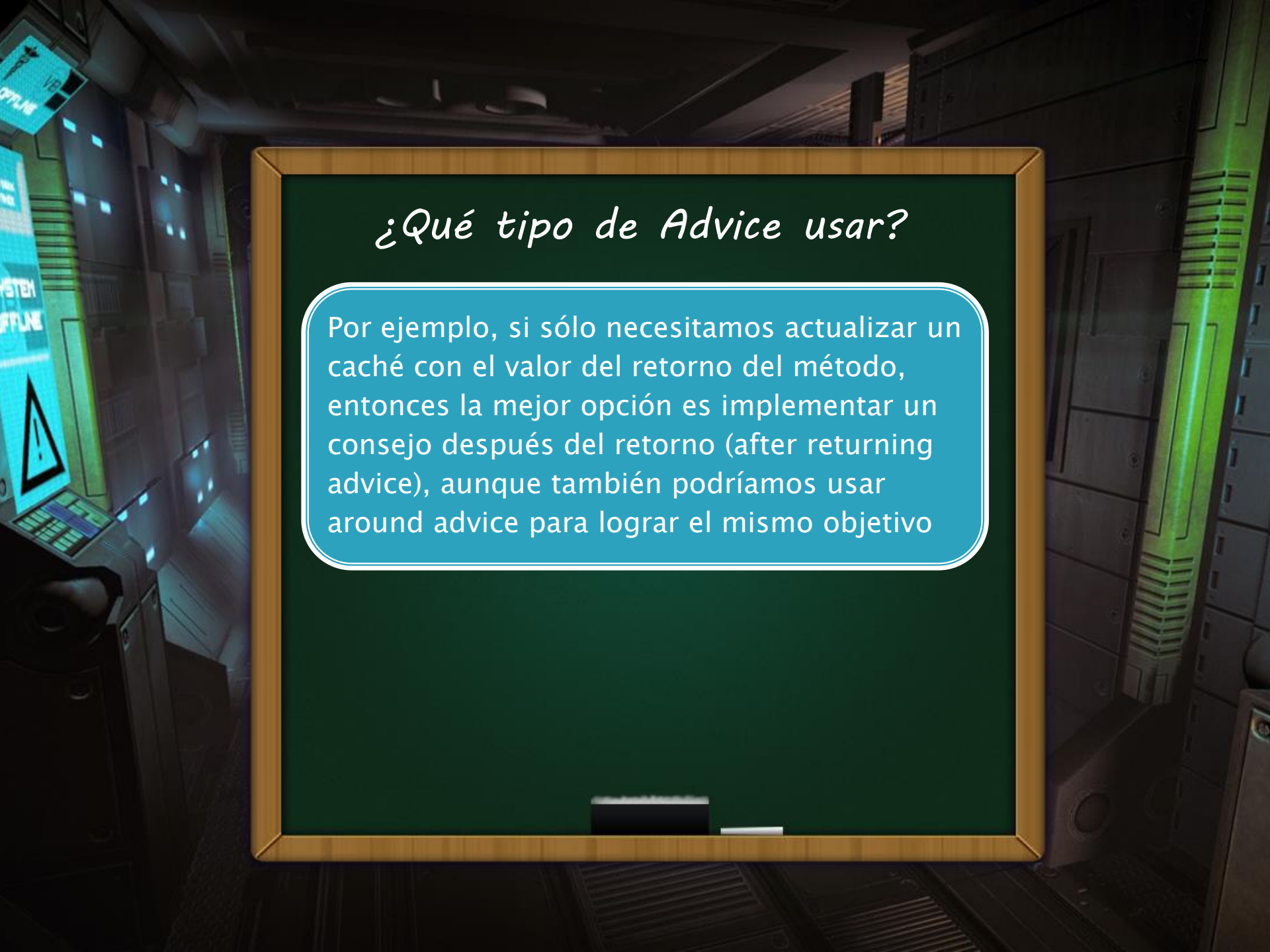
Before advice: Consejo que se ejecuta antes de una ejecución del método

After returning advice: Consejo a ser ejecutado después de que una ejecución de método sea completado con normalidad, es decir haya retornado el valor/objeto

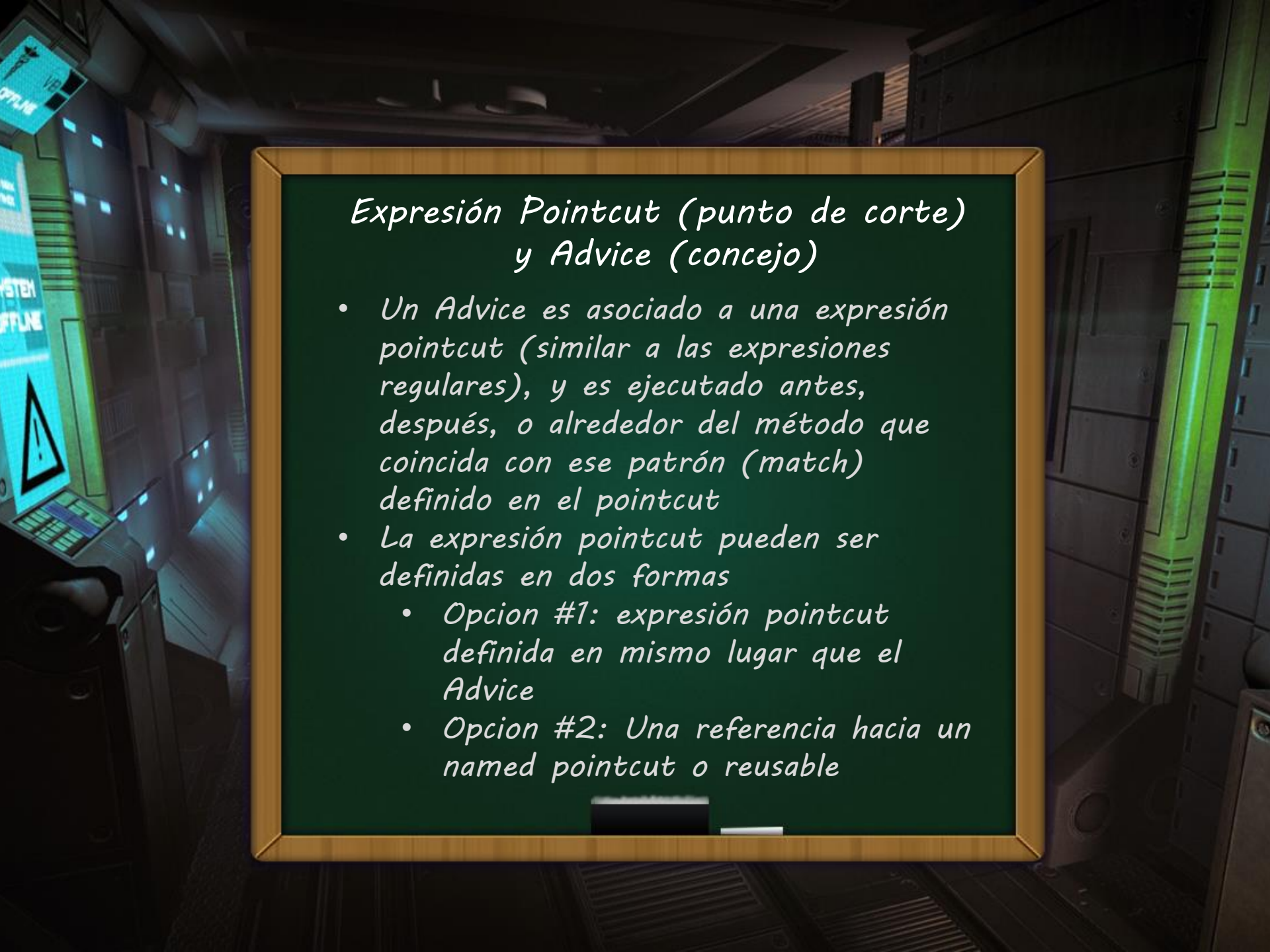
After throwing advice: Consejo a ser ejecutado si un método lanza una excepción

After (finally) advice: Consejos a ser ejecutado independientemente de si un método retorna/finaliza normalmente o si lanza una excepción


Tipos de Advice en Spring
Around advice
• Consejo que envuelven a una invocación de método
• Este es el más poderoso de los tipos de advice
• Around advice puede realizar tareas personalizadas antes y después de la invocación del método


¿Qué tipo de Advice usar?
Por ejemplo, si sólo necesitamos actualizar un caché con el valor del retorno del método, entonces la mejor opción es implementar un consejo después del retorno (after returning advice), aunque también podríamos usar around advice para lograr el mismo objetivo

Declarando un Advice


Expresión Pointcut (punto de corte) y Advice (concejo)
• Un Advice es asociado a una expresión pointcut (similar a las expresiones regulares), y es ejecutado antes, después, o alrededor del método que coincida con ese patrón (match) definido en el pointcut
• La expresión pointcut pueden ser definidas en dos formas
• Opcion #1: expresión pointcut definida en mismo lugar que el Advice
• Opcion #2: Una referencia hacia un named pointcut o reusable

Before Advice (antes del método)

Before advice es definido en una clase aspect usando la anotación @Before

Ejemplo opción #1 – La expresión Pointcut es definida en el mismo lugar

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EjemploAntes {

   @Before("execution(* com.abc.miapp.dao.*.*(..))")
   public void compruebaAcceso() {
      // ...
   }
}
```

aspecto

Expresión Pointcut

advice

After Returning Advice (después del retorno del método)

El advice After returning se ejecuta cuando el método en cuestión (que coincide con la expreción) retorna normalmente (sin errores)

Ejemplo opción #1 – La expresión Pointcut es definida en el mismo lugar

```
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class EjemploDespuesRetorno {

   @AfterReturning("execution(* com.abc.miapp.dao.*.*(..))")
   public void compruebaAcceso() {
      // ...
   }
}
```

After Throwing Advice
(después de lanzar la excepción)

El advice After throwing se ejecuta cuando el método en cuestión lanza un error de excepción

Ejemplo opción #1 – La expresión Pointcut es definida en el mismo lugar

```
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class EjemploDespuesLanzarExepcion {

   @AfterThrowing("execution(* com.abc.miapp.dao.*.*(..))")
   public void manejarError() {
      // ...
   }
}
```

After (finally) Advice
(después del retorno o lanzar la excepción)

advice After se ejecuta independientemente de la ejecución del método, si finalizó normalmente o si ocurrió un error, típicamente se utiliza para liberar o cerrar recursos/conexiones, etc.

Ejemplo opción #1 – La expresión Pointcut es definida en el mismo lugar

```
import org.aspectj.lang.annotation.After;

@Aspect
public class EjemploDespues {

   @After("execution(* com.abc.miapp.dao.*.*(..))")
   public void hacerAlgoInteresante() {
      // ...
   }
}
```

Around Advice (alrededor del método)

Around es el advice más robusto, se ejecuta alrededor del método en cuestión

- Tiene la oportunidad de hacer un trabajo, tanto antes como después de la invocación del método en cuestión, y determinar cuándo, cómo, e incluso si el método realmente llega a ejecutarse completamente.

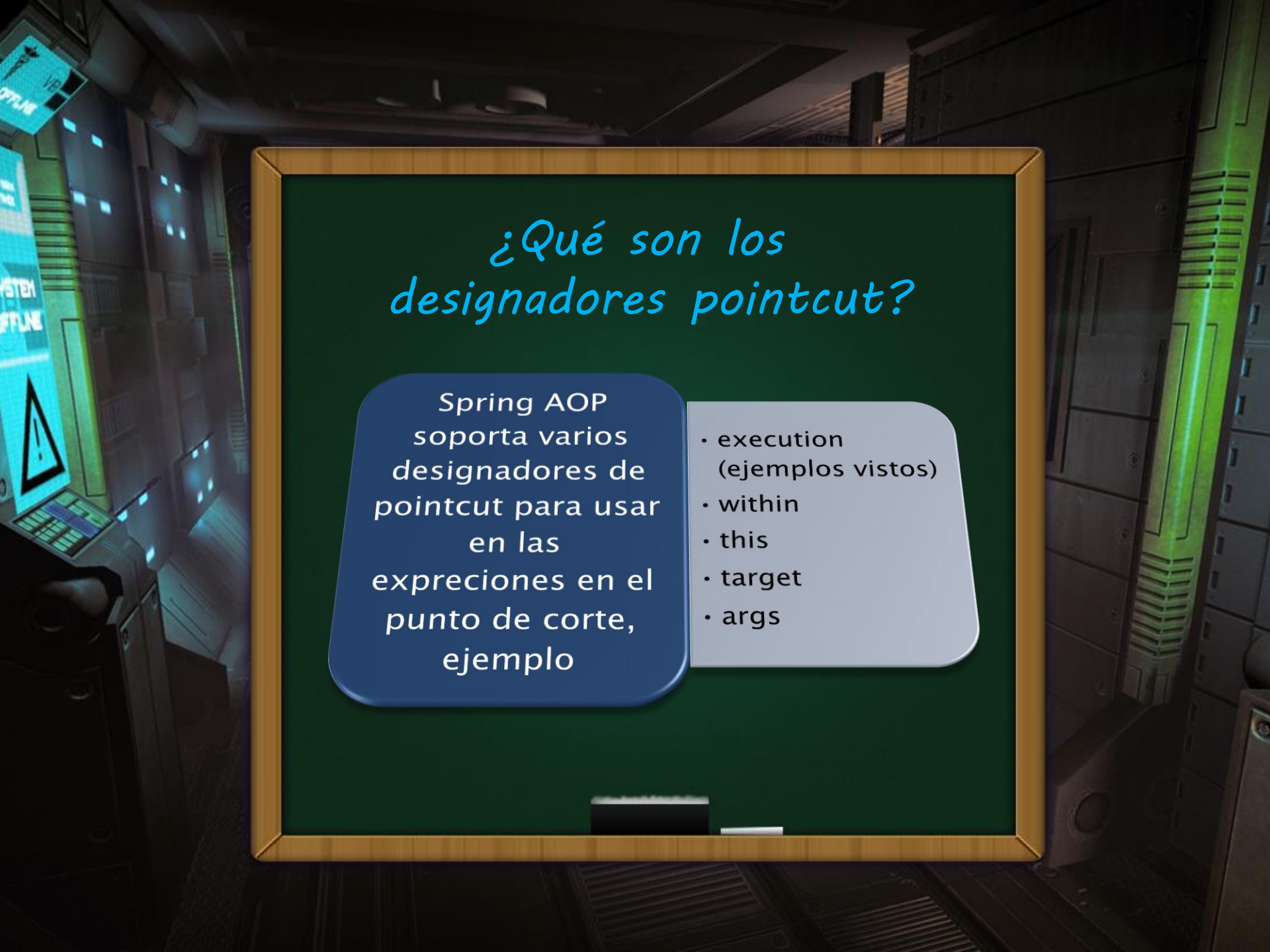
El primer argumento del método advice (o consejo) es del tipo ProceedingJoinPoint

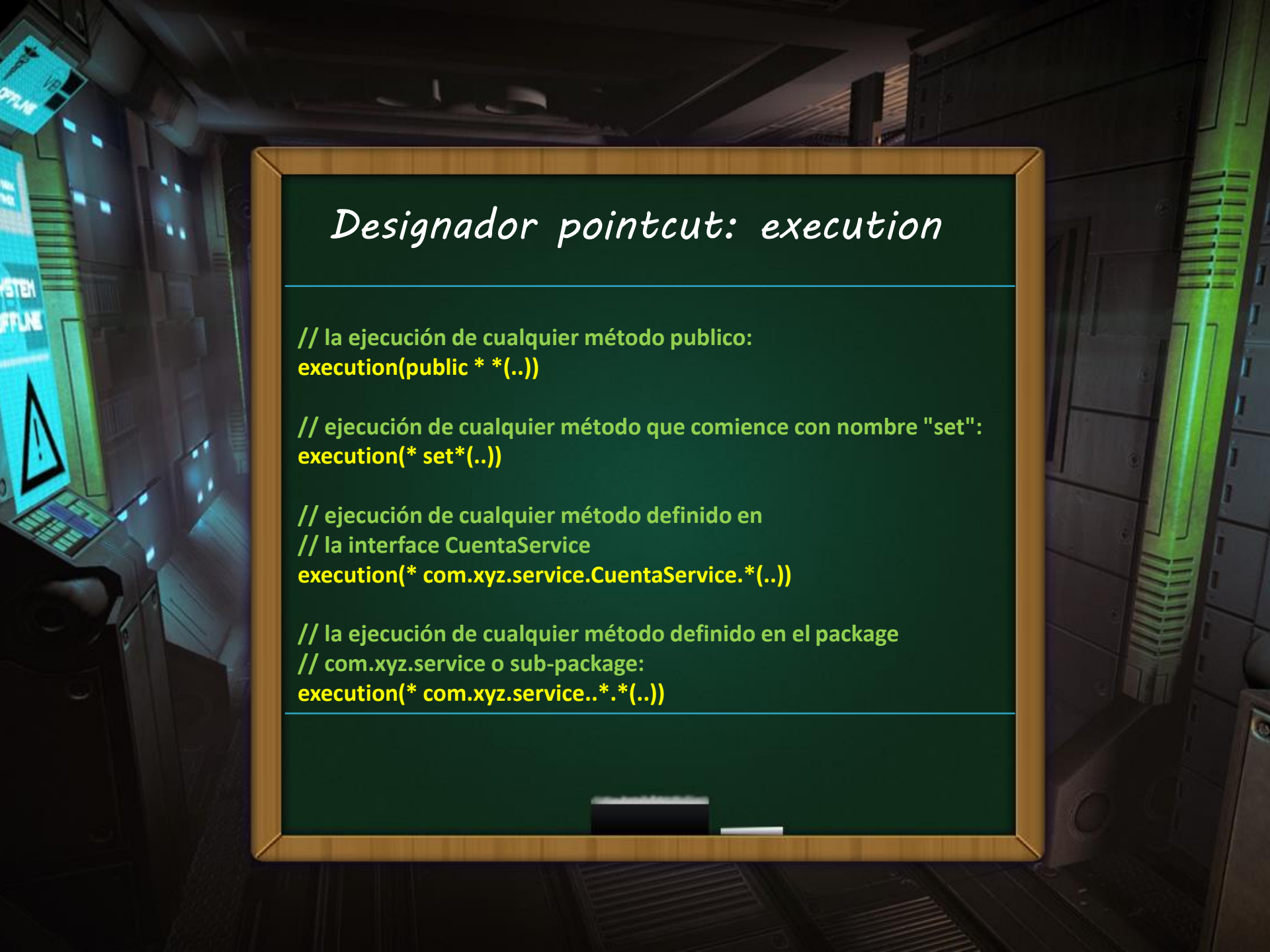
- Dentro del cuerpo del método advice, invocamos proceed() (método del objeto ProceedingJoinPoint), esto hace que la invocación del método en cuestión quede anidado dentro de la ejecución del advice anotado con @Around

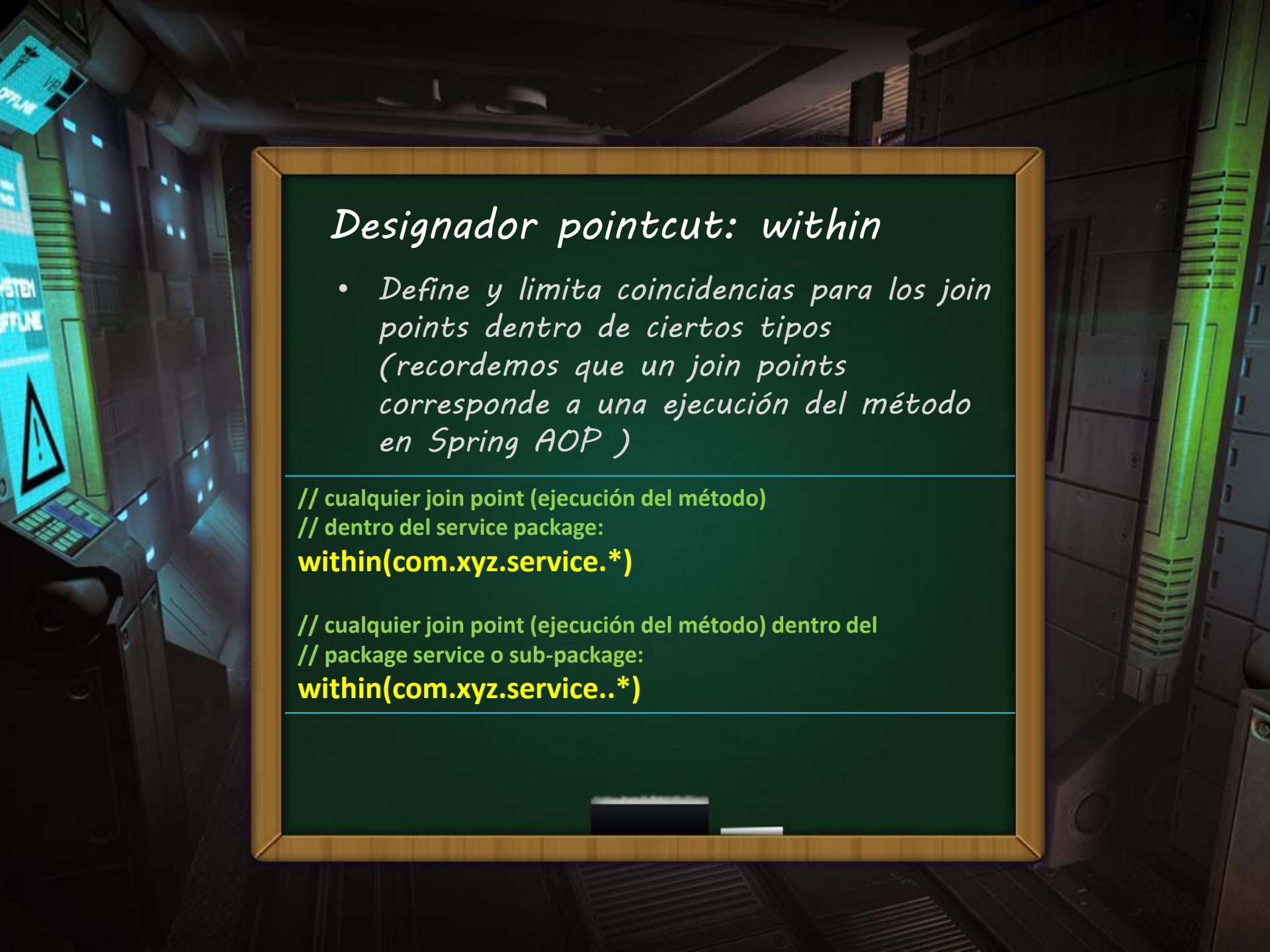
Around Advice
(alrededor del método)

```
import org.aspectj.lang.annotation.Around;

@Aspect
public class EjemploAlrededor {

   @Around("execution(* com.abc.miapp.dao.*.*(..))")
   public void hacerAlgoInteresante(ProceedingJoinPoint pjp)
                                                    throws Throwable {
      // Hacemos algo antes de llamar al método target o en cuestión
      // ...
      // Invocamos el método target o en cuestión
      Object target = pjp.proceed();
      // Hacemos  algo después de llamar al método target
      // ...
      // Retornamos el valor de la invocación del método target
      return target;
   }
}
```

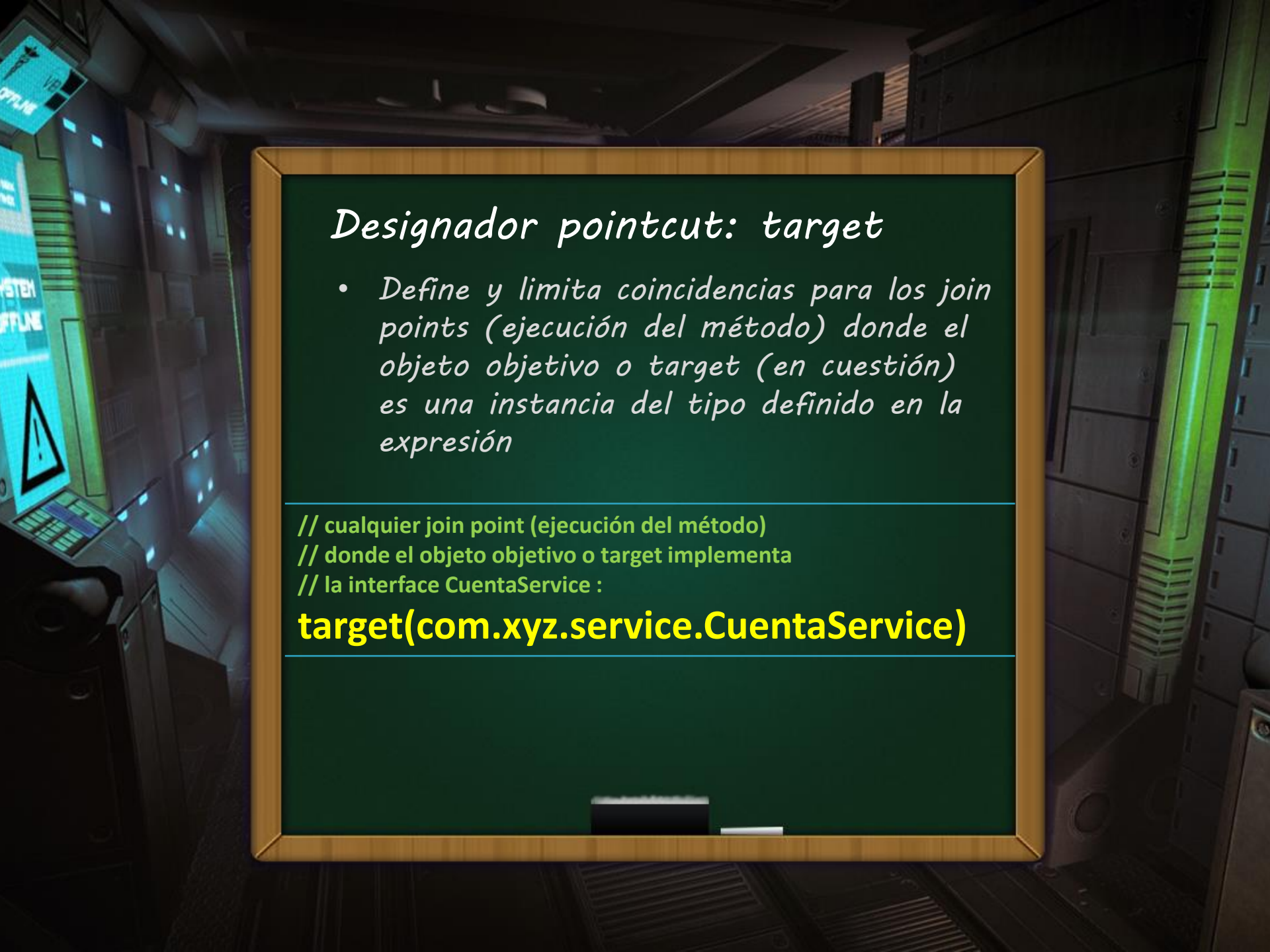
Designadores Pointcut (usado en las expresiones)

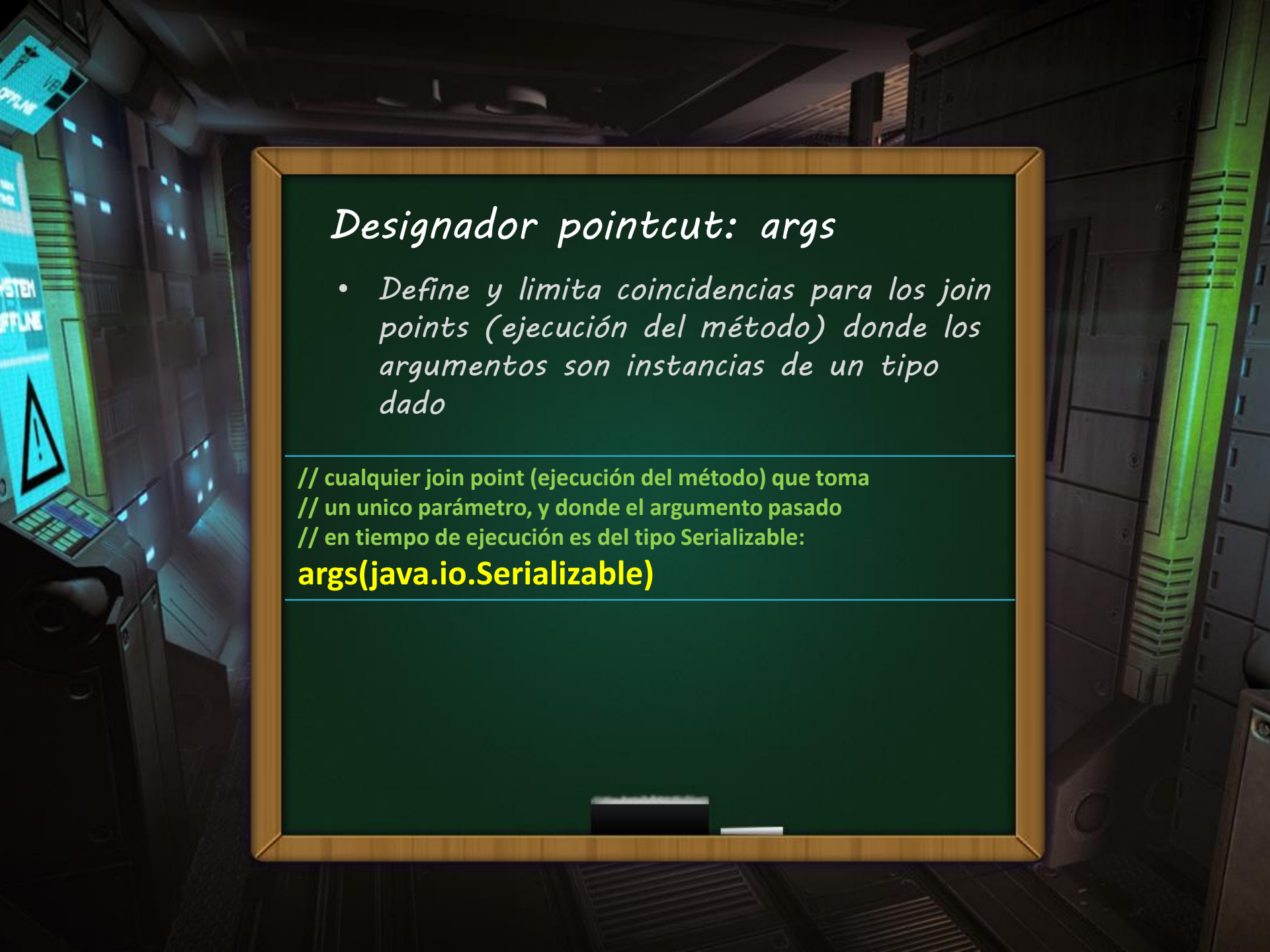

¿Qué son los designadores pointcut?
Spring AOP soporta varios designadores de pointcut para usar en las expreciones en el punto de corte, ejemplo
• execution (ejemplos vistos)
• within
• this
• target
• args


Designador pointcut: execution
// la ejecución de cualquier método publico:
execution(public * *(..))
// ejecución de cualquier método que comience con nombre "set":
execution(* set*(..))
// ejecución de cualquier método definido en
// la interface CuentaService
execution(* com.xyz.service.CuentaService.*(..))
// la ejecución de cualquier método definido en el package
// com.xyz.service o sub-package:
execution(* com.xyz.service..*.*(..))


Designador pointcut: within
• Define y limita coincidencias para los join points dentro de ciertos tipos (recordemos que un join points corresponde a una ejecución del método en Spring AOP)
// cualquier join point (ejecución del método)
// dentro del service package:
within(com.xyz.service.*)
// cualquier join point (ejecución del método) dentro del
// package service o sub-package:
within(com.xyz.service..*)


Designador pointcut: this
• Define y limita coincidencias para los join points donde la referencia del bean (un proxy Spring AOP) es una instancia de un determinado tipo
// cualquier join point donde el proxy implementa
// la interface CuentaService
this(com.xyz.service.CuentaService)


Designador pointcut: target
• Define y limita coincidencias para los join points (ejecución del método) donde el objeto objetivo o target (en cuestión) es una instancia del tipo definido en la expresión
// cualquier join point (ejecución del método)
// donde el objeto objetivo o target implementa
// la interface CuentaService :
target(com.xyz.service.CuentaService)



Designador pointcut: args

- Define y limita coincidencias para los join points (ejecución del método) donde los argumentos son instancias de un tipo dado

```
// cualquier join point (ejecución del método) que toma
// un unico parámetro, y donde el argumento pasado
// en tiempo de ejecución es del tipo Serializable:
args(java.io.Serializable)
```


Declarar un
Reusable Pointcut
(Opción # 2)

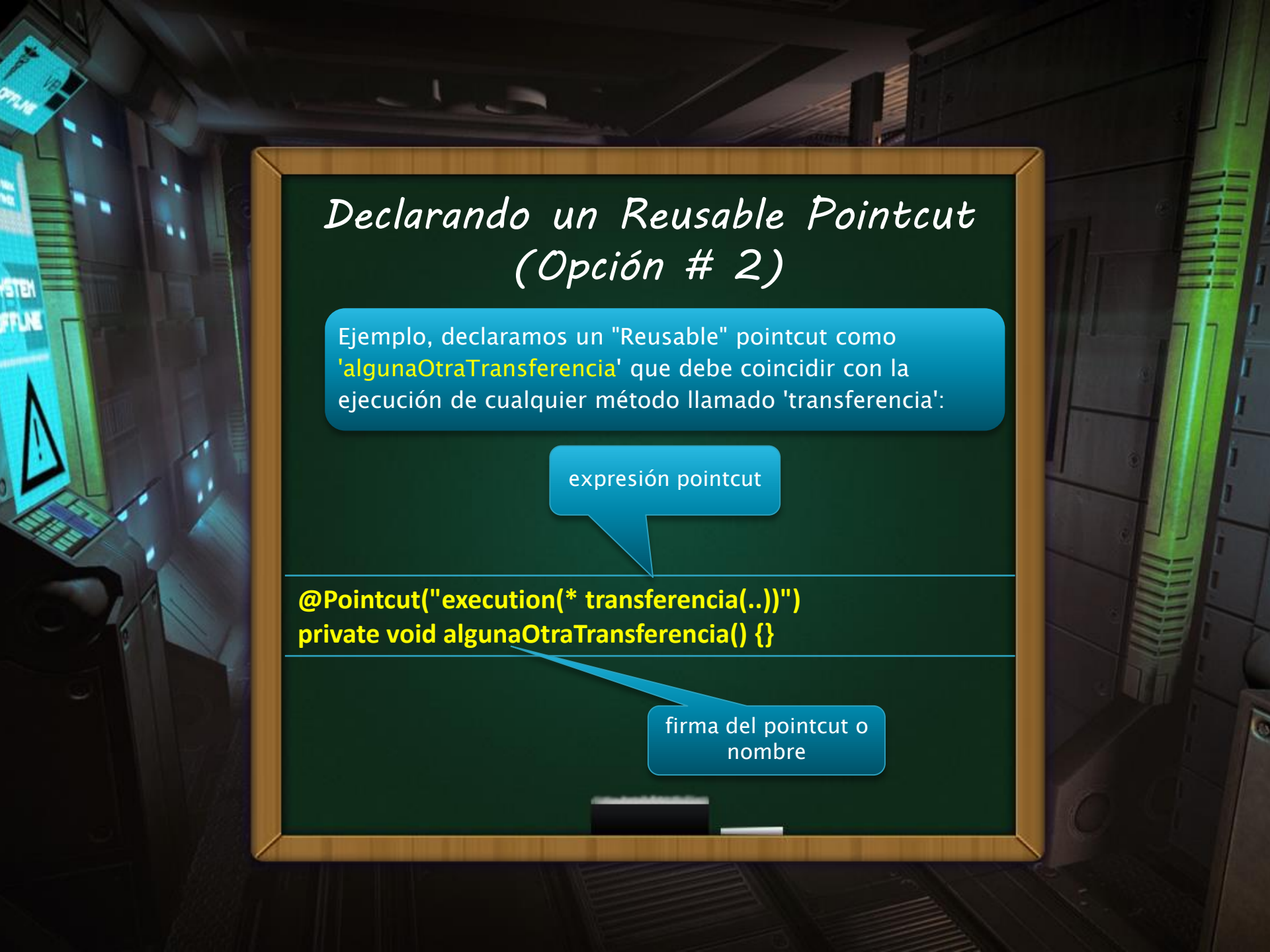
Declarando un Reusable Pointcut (Opción # 2)

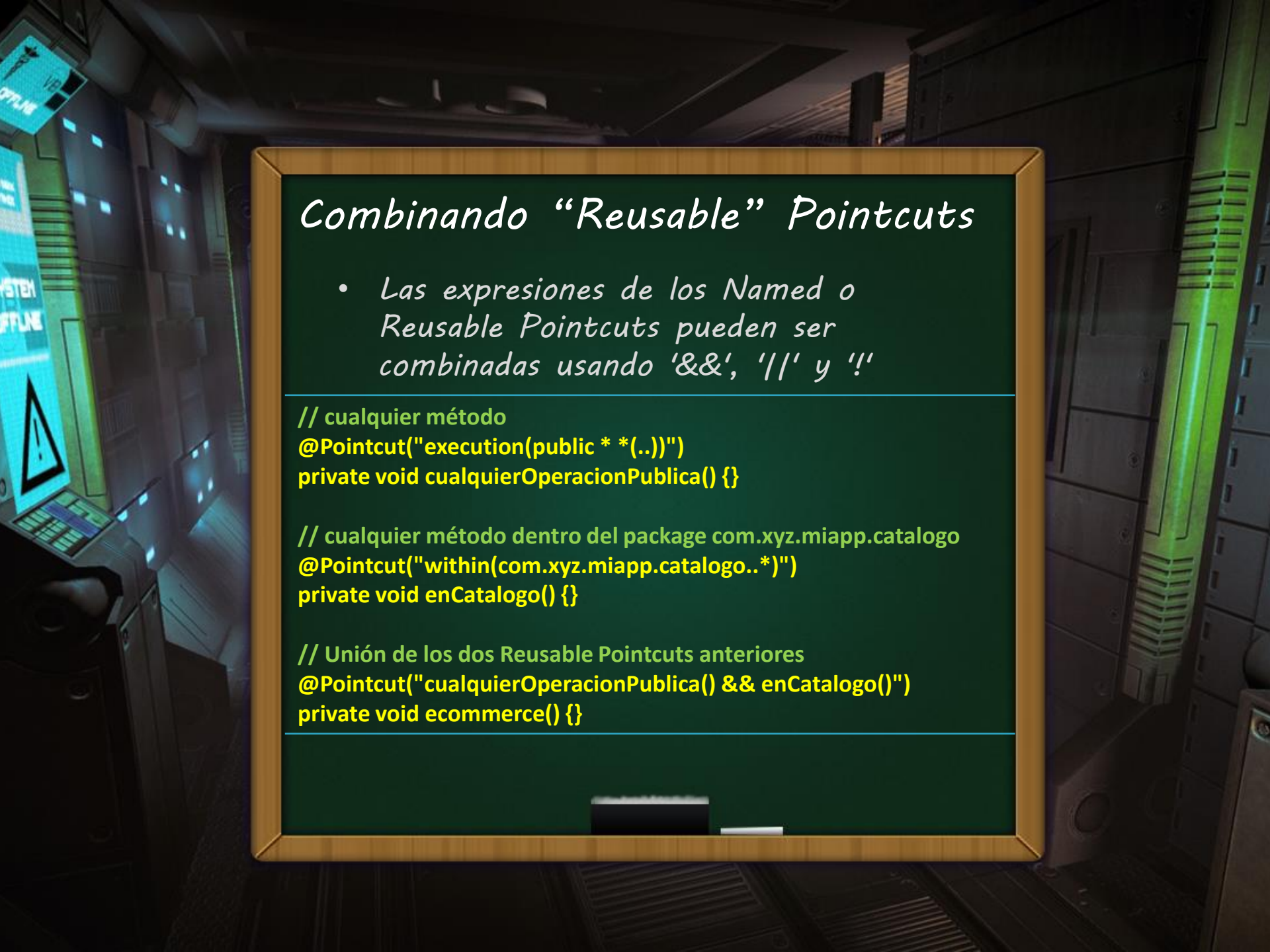
Útil cuando necesitamos reutilizar una expresión pointcut

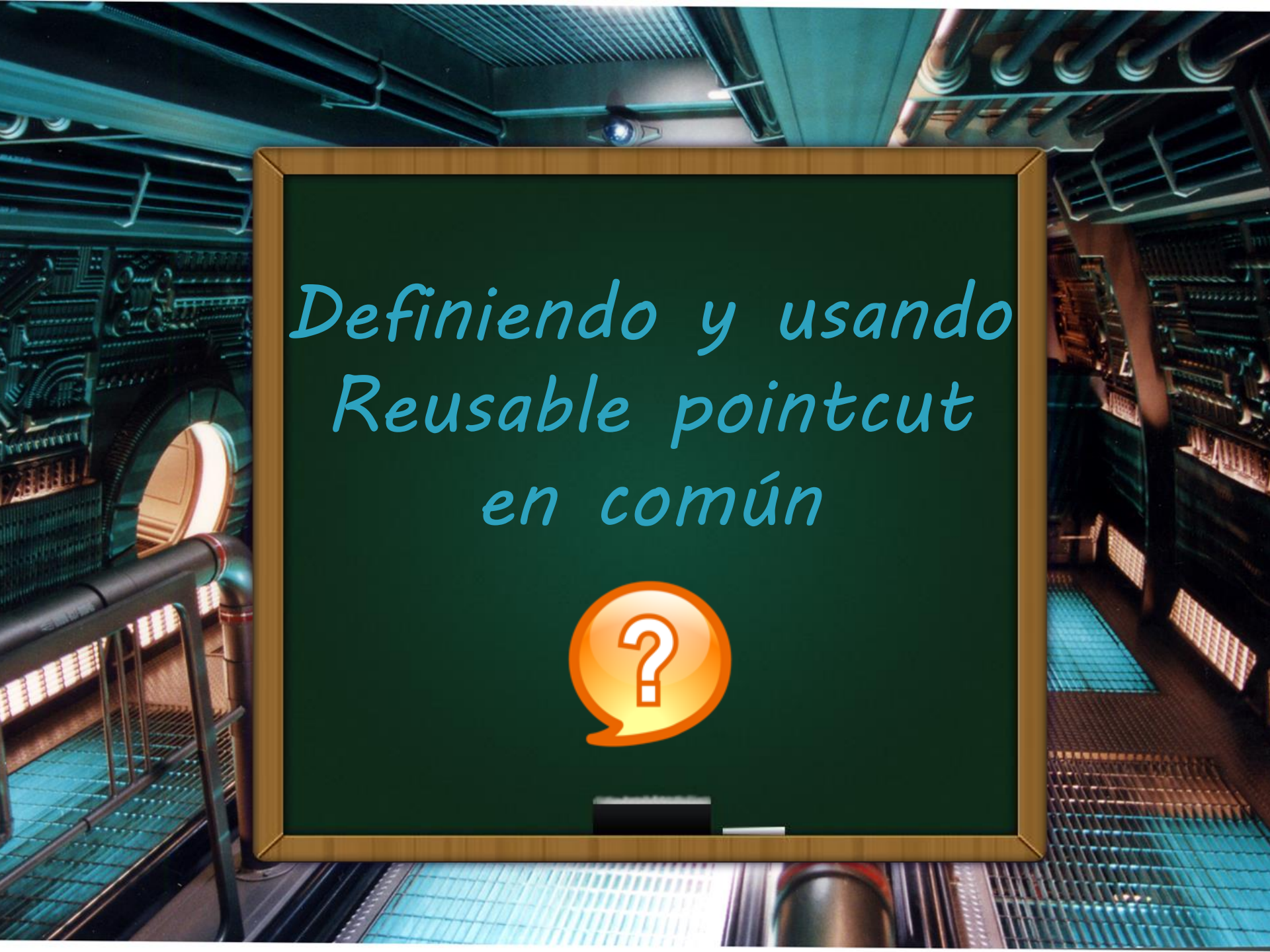
- Podemos crear un “Reusable" (o Named) Pointcut y usar este en varios lugares

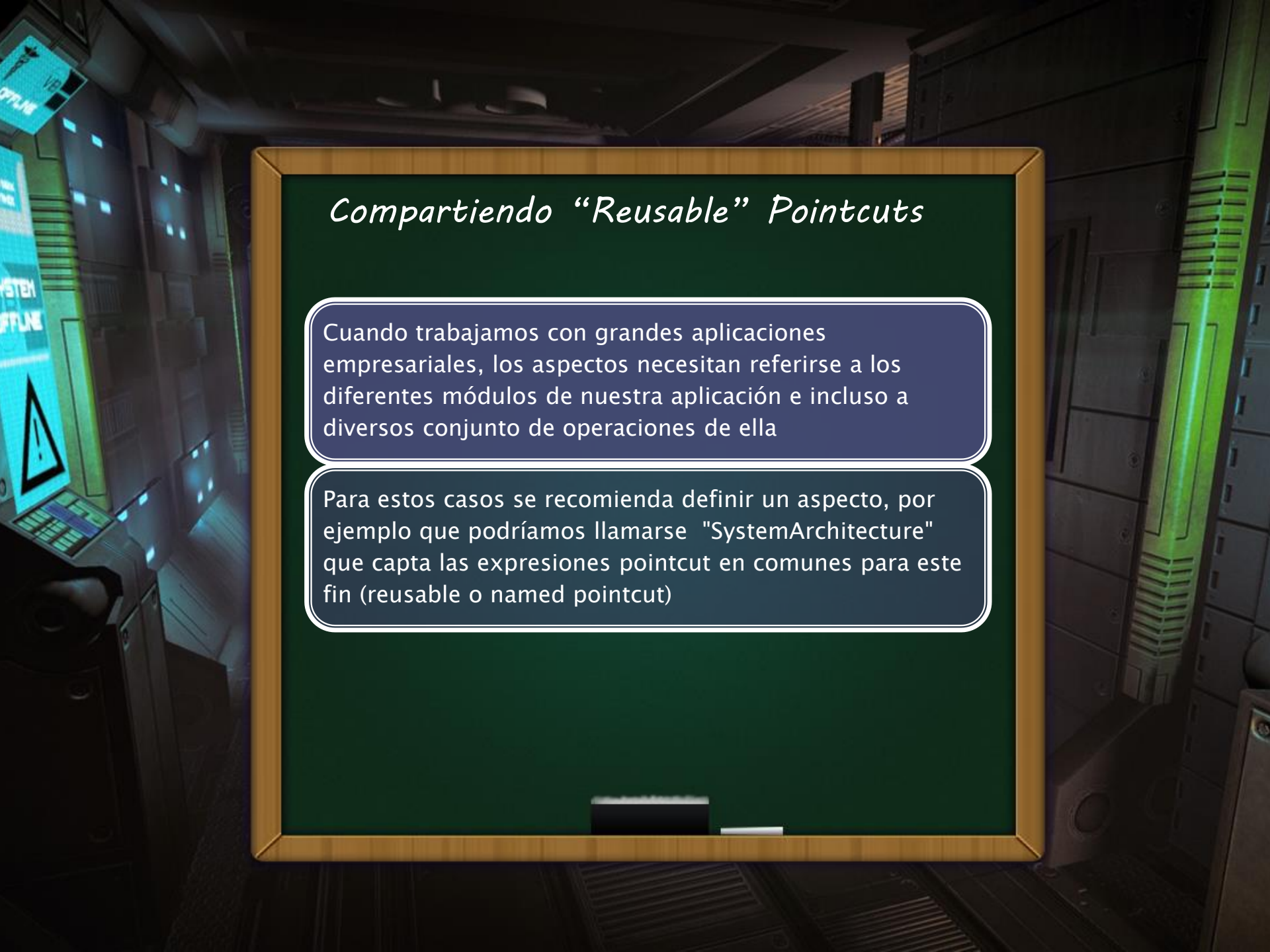
Una declaración de "Reusable" pointcut tiene 2 partes:

- Una expresión pointcut que determina exactamente que ejecuciones de métodos (join point) nos interesa interceptar, usamos la anotación @Pointcut
- Y una firma de método que comprende un nombre y parámetros (sin implementar)


Declarando un Reusable Pointcut (Opción # 2)
Ejemplo, declaramos un "Reusable" pointcut como 'algunaOtraTransferencia' que debe coincidir con la ejecución de cualquier método llamado 'transferencia':
expresión pointcut
@Pointcut("execution(* transferencia(..))")
private void algunaOtraTransferencia() {}
firma del pointcut o nombre


Combinando "Reusable" Pointcuts
• Las expresiones de los Named o Reusable Pointcuts pueden ser combinadas usando '&&', '||' y '!'
// cualquier método
@Pointcut("execution(public * *(..))")
private void cualquierOperacionPublica() {}
// cualquier método dentro del package com.xyz.miapp.catalogo
@Pointcut("within(com.xyz.miapp.catalogo..*)")
private void enCatalogo() {}
// Unión de los dos Reusable Pointcuts anteriores
@Pointcut("cualquierOperacionPublica() && enCatalogo()")
private void ecommerce() {}


Definiendo y usando
Reusable pointcut
en común


Compartiendo “Reusable” Pointcuts
Cuando trabajamos con grandes aplicaciones empresariales, los aspectos necesitan referirse a los diferentes módulos de nuestra aplicación e incluso a diversos conjunto de operaciones de ella
Para estos casos se recomienda definir un aspecto, por ejemplo que podríamos llamarse "SystemArchitecture" que capta las expresiones pointcut en comunes para este fin (reusable o named pointcut)

Ejemplo del aspecto: System Architecture

```
package com.xyz.algunapp;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

/**
* Join point aplicado a la capa web, si el método que se ejecuta
* está definido dentro del package com.xyz.algunapp.web
* o cualquier sub-package de web.
*/
@Pointcut("within(com.xyz.algunapp.web..*)")
public void inWebLayer() {}

/**
* Join point aplicado a la capa service, si el método que se ejecuta
* está definido dentro del package com.xyz.algunapp.service
* o cualquier sub-package de service.
*/
@Pointcut("within(com.xyz.algunapp.service..*)")
public void inServiceLayer() {}

//Continúa en la siguiente dispositiva
```

Ejemplo del aspecto: System Architecture

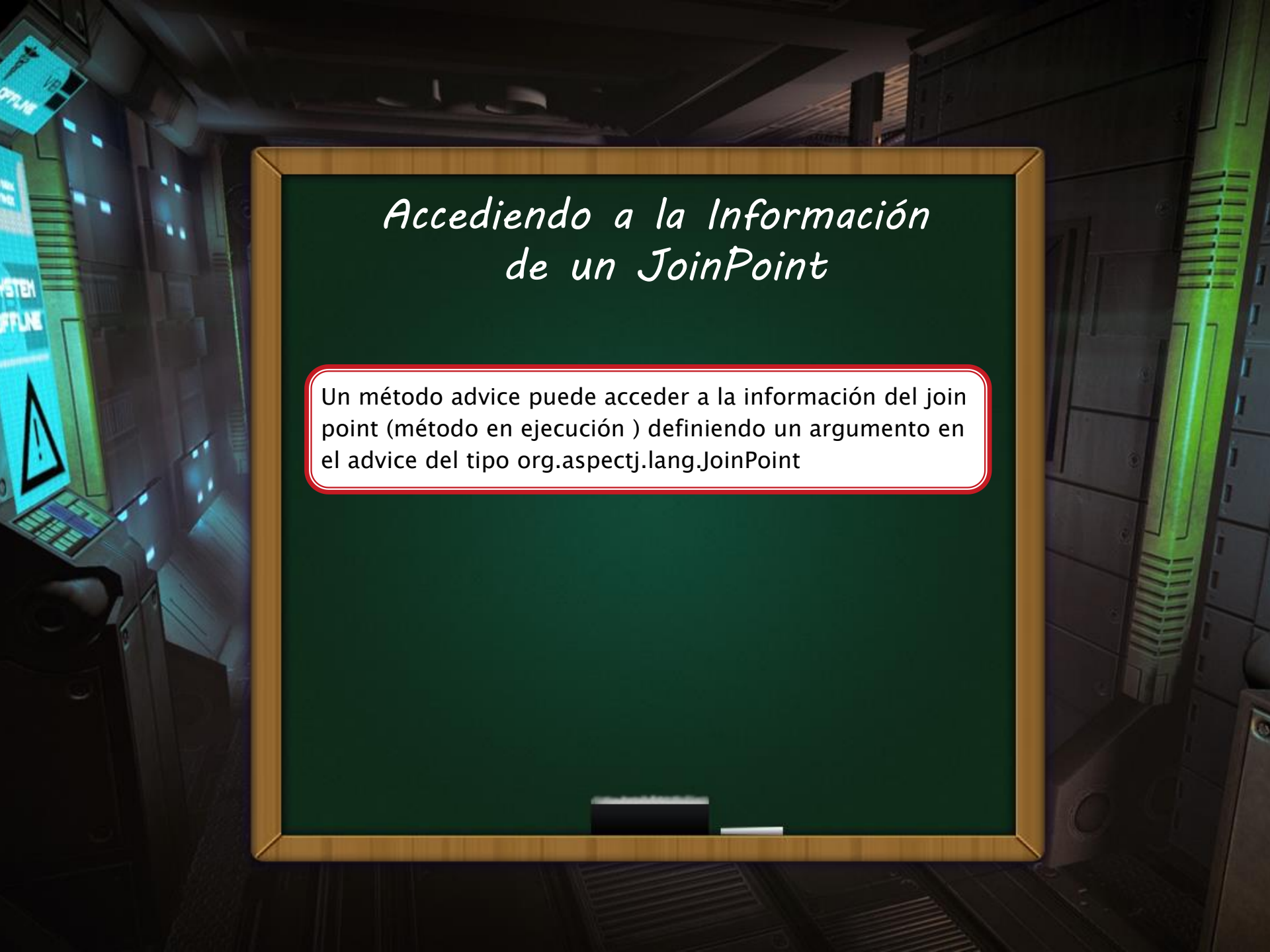
```
/**
* Join point aplicado a la capa data access, si el método
* es definido dentro del package com.xyz.algunapp.dao
* o cualquier sub-package de dao.
*/
@Pointcut("within(com.xyz.algunapp.dao..*)")
public void inDataAccessLayer() {}

/**
* Join point aplicado a la ejecución de cualquier método definido en la
* interface service. Esta definición asume que la interfaz está ubicada en el package
* "service", y que las implementaciones están en los sub-packages.
*
* Si tenemos diferentes módulos con interfaces service (por ejemplo,
* en los packages com.xyz.algunapp.abc.service y com.xyz.def.service) entonces
* la expresión pointcut podría ser "execution(* com.xyz.algunapp..service.*.*(..))"
*
@Pointcut("execution(* com.xyz.algunapp.service.*.*(..))")
public void businessService() {}

/**
* Join point aplicado a la ejecución de cualquier método definido en la
* interface dao. Esta definición asume que la interfaz está ubicada en el package
* "dao", y que las implementaciones están en los sub-package.
*/
@Pointcut("execution(* com.xyz.algunapp.dao.*.*(..))")
public void dataAccessOperation() {}

}
```

Usando un Named Pointcut

Ejemplo opción #2 (Named Pointcut o Reusable), utiliza una referencia hacia un named pointcut (que se define en la clase SystemArchitecture de la diapositiva anterior)

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EjemploUsandoNamedPointcut {

   @Before("com.xyz.algunapp.SystemArchitecture.dataAccessOperation()")
   public void compruebaAcceso() {
      // ...
   }
}
```


Accediendo a la Información de un JoinPoint


Accediendo a la Información
de un JoinPoint
Un método advice puede acceder a la información del join point (método en ejecución) definiendo un argumento en el advice del tipo org.aspectj.lang.JoinPoint

Accediendo a la Información de un JoinPoint

```
@Aspect
@Order(1)
public class AspectoSaludoLogger {

   private Log log = LogFactory.getLog(this.getClass());

   @Pointcut("execution(* SaludoService.decirHola(..))")
   private void holaLogger(){}

   @Before("holaLogger()")
   public void logAntes(JoinPoint joinPoint) {
      String nombreMetodo = joinPoint.getSignature().getName();
      String argumentos = Arrays.toString(joinPoint.getArgs());
      log.info("Before1: se invoca " + nombreMetodo + " con " + argumentos);
   }

   @AfterReturning("holaLogger()")
   public void logDespuesRetorno(JoinPoint joinPoint) {
      String nombreMetodo = joinPoint.getSignature().getName();
      String argumentos = Arrays.toString(joinPoint.getArgs());
      log.info("AfterReturning1: se invoca " + nombreMetodo + " con " +
argumentos);
   }
}
```

Múltiples Aspectos y Orden

Ordenamiento de Aspectos

Para un target u objetivo, puede tener asociado varios aspectos, con diferentes tipos de tareas o procesos

El ordenamiento de los aspectos puede ser muy importante, para ello se especifica mediante la anotación @Order(<número>)

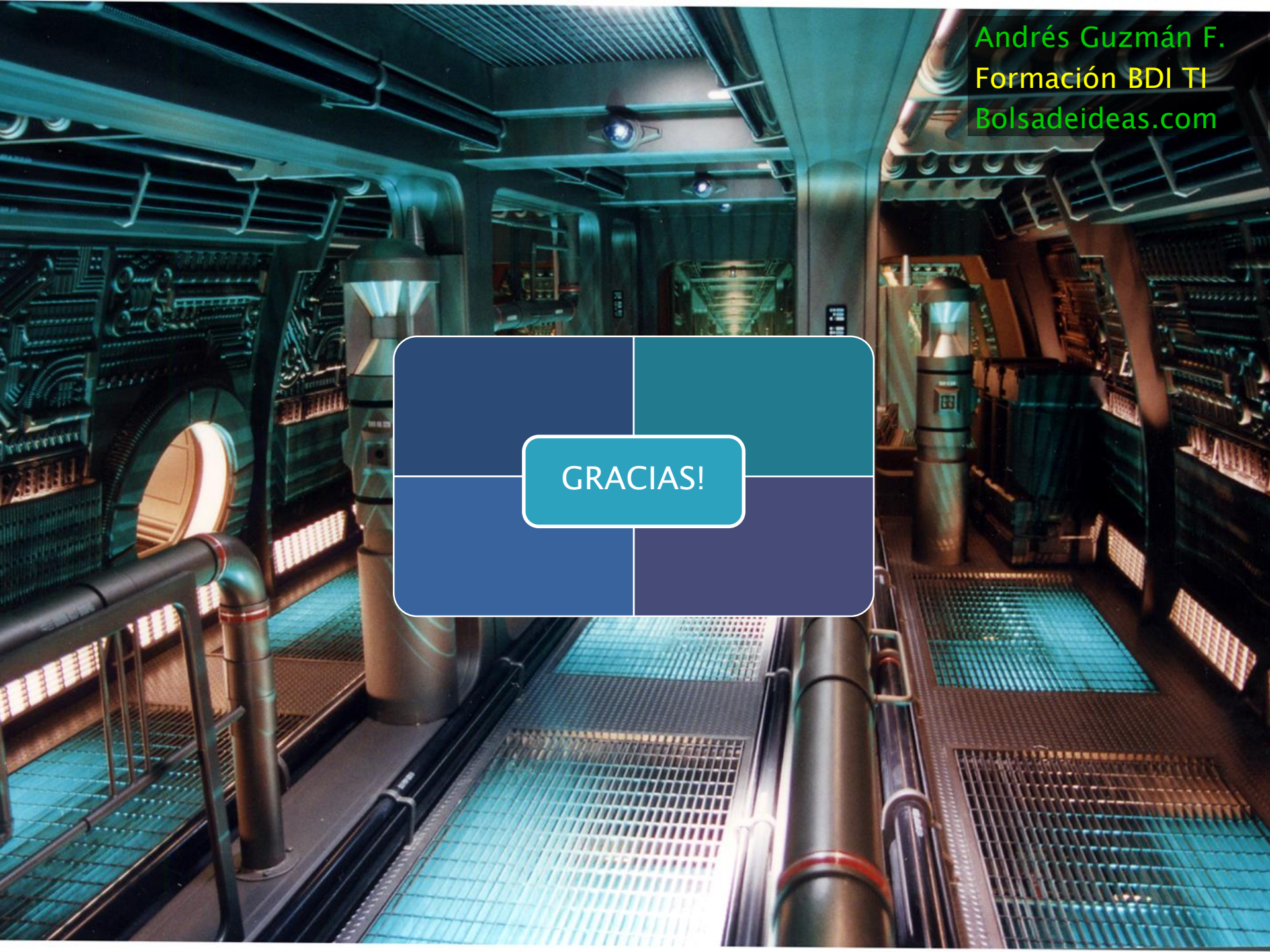
Ordenamiento de Aspectos

```
@Aspect
@Order(1) // especificamos el orden
public class AspectoSaludoLogger {

   private Log log = LogFactory.getLog(this.getClass());

   @Pointcut("execution(* SaludoService.decirHola(..))")
   private void holaLogger(){}

   @Before("holaLogger()")
   public void logAntes(JoinPoint joinPoint) {
      String nombreMetodo = joinPoint.getSignature().getName();
      String argumentos = Arrays.toString(joinPoint.getArgs());
      log.info("Before1: se invoca " + nombreMetodo + " con " + argumentos);
   }
}
```

Dependencias Maven

Dependencia Maven

```
<!-- Aspectj -->
<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjrt</artifactId>
   <version>1.8.6</version>
</dependency>

<dependency>
   <groupId>org.aspectj</groupId>
   <artifactId>aspectjweaver</artifactId>
   <version>1.8.6</version>
</dependency>
```


Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com
GRACIAS!