



Guía general: cómo desarrollar un CRUD orientado a objetos en Java

1. Analiza el dominio: qué vas a gestionar

Objetivo: identificar el tipo de objeto que quieras guardar.

Por ejemplo: *estudiantes, cursos, animales, productos*, etc.

Pregúntate: ¿qué datos necesito guardar de cada elemento? → Esos serán los **atributos** del modelo.

Ejemplo:

Para gestionar cursos: id, title, hours, modality.

2. Crea la clase base (modelo o entidad)

Nombre habitual: Student, Course, Animal, Product, etc.

Qué debe incluir:

- Atributos **privados** (encapsulación).
- **Constructores** (con y sin id).
- **Getters y setters** para acceder/modificar los datos.
- Método `toString()` para mostrar la información por consola.

 Este paso representa el “molde” del tipo de dato que se va a gestionar.

3. Diseña la interfaz del repositorio

Nombre habitual: CourseRepository, StudentRepository, etc.

Función:

Define el **contrato** que cualquier clase encargada de almacenar los objetos debe cumplir. Gracias a ella, el resto del programa podrá usar el repositorio sin saber **cómo** guarda los datos.

Métodos típicos:

- `create(T elemento)`
- `findById(int id)`
- `findAll()`
- `update(T elemento)`
- `deleteById(int id)`
- `count()`

 Aquí aplicamos el principio de **abstracción**: sabemos qué hace, no cómo lo hace.



4. Crea una clase que implemente la interfaz (repositorio concreto)

Nombre habitual:

InMemoryCourseRepository, FileStudentRepository, DatabaseProductRepository, etc.

Función:

Guardar realmente los objetos, normalmente usando una **estructura de datos** (lista o mapa).

Esta clase se encuentra “íntimamente” relacionado con la manera en la que se van a guardar los datos. Es la más específica de todas, y la que más nivel de detalle debe contener.

 *Aquí aplicamos polimorfismo: diferentes repositorios podrán comportarse igual desde fuera.*

5. Crea la capa de servicio

Nombre habitual: CourseService, StudentService, etc.

Función:

- Aplicar **reglas de negocio y validaciones** antes de guardar o modificar.
- Delegar la gestión de los datos al repositorio.
- Servir de puente entre el programa principal y el repositorio.

 *Aquí se aplican abstracción y polimorfismo: el servicio trabaja con la interfaz, no con una clase concreta.*

6. Implementa la clase principal (Main)

Función:

- Crear los objetos principales del programa.
- Probar las operaciones CRUD.
- Mostrar resultados por pantalla.

 Aquí se ve el uso real del polimorfismo: el servicio no sabe cómo se guardan los datos.



A continuación mostramos qué **principios SOLID** se aplican **directamente** en el desarrollo de este tipo de ejercicios en Java, y cómo se ven reflejados en el ejemplo del CRUD (por ejemplo, el de *Courses* o *Students*).

Principios SOLID aplicados a un CRUD en Java

Recordatorio:

Los principios SOLID son un conjunto de buenas prácticas de programación orientada a objetos que ayudan a diseñar software **mantenible, flexible y fácil de ampliar**.

S — Single Responsibility Principle (SRP)

Principio de responsabilidad única

Definición sencilla:

Cada clase debe tener una sola razón para cambiar. Es decir, **una única responsabilidad clara**.

Cómo se aplica en nuestro CRUD:

Clase	Responsabilidad única
Student / Course	Representar los datos del modelo (atributos y estado).
StudentRepository	Definir las operaciones CRUD.
InMemoryStudentRepository	Implementar el almacenamiento.
StudentService	Aplicar las validaciones y reglas de negocio.
Main	Ejecutar y mostrar resultados.

Beneficio:

Si mañana cambian las reglas de validación, solo modificas StudentService.

Si decides guardar en una base de datos, solo cambias InMemoryStudentRepository.

 Cada clase tiene un propósito claro y no mezcla responsabilidades.



O — Open/Closed Principle (OCP)

Principio abierto/cerrado

Definición sencilla:

El código debe estar **abierto a la extensión**, pero **cerrado a la modificación**.

Cómo se aplica:

- El sistema permite **añadir nuevas clases** sin tener que **cambiar las ya existentes**.
- Por ejemplo, puedes crear un nuevo repositorio (DatabaseStudentRepository) sin modificar el servicio ni la interfaz.

```
StudentRepository repo = new DatabaseStudentRepository();
```

```
StudentService service = new StudentService(repo);
```

Beneficio:

Se pueden añadir nuevas funcionalidades **sin romper nada**.

El programa **crece sin reescribirse**.

👉 *La interfaz StudentRepository actúa como una frontera estable entre el código antiguo y el nuevo.*

L — Liskov Substitution Principle (LSP)

Principio de sustitución de Liskov

Definición sencilla:

Las clases hijas o implementaciones deben poder sustituir a sus padres sin que el programa deje de funcionar.

Cómo se aplica:

- InMemoryStudentRepository **puede sustituir** a StudentRepository sin romper el código.
- Si mañana creas FileStudentRepository o ApiStudentRepository, el código seguirá funcionando mientras respeten los mismos métodos de la interfaz.

Beneficio:

Permite **cambiar implementaciones libremente**, manteniendo un comportamiento coherente.

👉 *Cualquier clase que implemente la interfaz puede usarse sin que el resto del sistema lo note.*



I — Interface Segregation Principle (ISP)

Principio de segregación de interfaces

Definición sencilla:

Es mejor tener **interfaces pequeñas y específicas** que una interfaz grande que obligue a implementar métodos que no necesitas.

Cómo se aplica:

- Nuestra interfaz StudentRepository solo define los métodos esenciales del CRUD: create, findById, findAll, update, deleteById, count.

Si más adelante se necesitara separar tipos de acceso, se podrían crear interfaces más específicas:

- ReadableRepository → solo lectura.
- WritableRepository → solo escritura.

Beneficio:

Las clases **implementan solo lo que necesitan**, no métodos vacíos o inútiles.

 *Las interfaces pequeñas son más fáciles de mantener y reutilizar.*

D — Dependency Inversion Principle (DIP)

Principio de inversión de dependencias

Definición sencilla:

Las clases deben depender de **abstracciones (interfaces)**, no de **implementaciones concretas**.

Cómo se aplica:

```
public class StudentService {  
    private StudentRepository repo; // depende de una interfaz, no de una clase concreta  
}
```

Y en el Main:

```
StudentRepository repo = new InMemoryStudentRepository();  
StudentService service = new StudentService(repo);
```

Beneficio:

El servicio no necesita saber cómo se guardan los datos.

Podemos reemplazar el repositorio por cualquier otro sin modificar StudentService.

 *Esta es la base del polimorfismo y la inyección de dependencias.*

“Programa contra una interfaz, no contra una implementación concreta.”



Resumen

Principio Nombre	Qué significa	Cómo se aplica en el CRUD
S	<i>Single Responsibility</i>	<i>Una clase, una función clara</i> Cada clase cumple un rol distinto (modelo, repositorio, servicio, main).
O	<i>Open/Closed</i>	<i>Extiende, no modifiques</i> Puedes añadir otro tipo de repositorio sin cambiar el resto.
L	<i>Liskov Substitution</i>	<i>Sustituye sin romper</i> Cualquier repositorio que implemente la interfaz funciona igual.
I	<i>Interface Segregation</i>	<i>Interfaces pequeñas y útiles</i> Solo métodos CRUD necesarios.
D	<i>Dependency Inversion</i>	<i>Depender de abstracciones, no implementaciones</i> StudentService usa StudentRepository, no su clase concreta.

Conclusión.

Al diseñar un CRUD con modelo, interfaz, repositorio, servicio y main, estamos aplicando de forma natural varios principios SOLID:

- Separar las responsabilidades.
- Trabajar contra interfaces, no implementaciones.
- Permitir crecimiento sin romper el código existente.

 Esto hace que nuestro software sea más fácil de mantener, probar y reutilizar.



Castilla-La Mancha

Consejería de Educación, Cultura y Deportes

IES VIRREY MORCILLO

AVDA. MENÉNDEZ PELAYO S/N

967 140 881 02003120.ies@edu.jccm.es

