DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

- Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- Abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

- Puede implementarse con:
  - Inyección de dependencias
  - IoC (Inversión del control)

¿Cuales son los módulos de alto nivel y los de bajo nivel?

Pensando en módulos como librerías o paquetes, módulo de alto nivel serían aquellos que tradicionalmente tienen dependencias y los de bajo nivel de los que dependen.

Módulo de alto nivel -> donde se invoca la acción del **Módulo de bajo nivel** ->donde se realiza la acción.

ES razonable que no existan una dependencia entre concreciones sino que se debe depender de una abstracción. Pero según el enfoque que le demos podemos estar aplicando mal la inversión de dependencia aunque dependamos de una abstracción.

Qué tal si, dado que nos movemos dentro del ámbito de la orientación de objetos, sustituimos “módulos de alto nivel” por “clases importantes”. Y desglosamos el término “abstracciones” por “interfaces o clases abstractas”. Con esto podemos llegar a lo siguiente:

Las clases importantes no deben depender de las clases menos importantes. Ambas deben depender de interfaces o clases abstractas.

Si í lo importante es, por ejemplo, la estabilidad del código, es decir que tu aplicación sufra lo menos posible ante los cambios, podemos también sustituir “importante” por “estable” y la definición quedaría así:

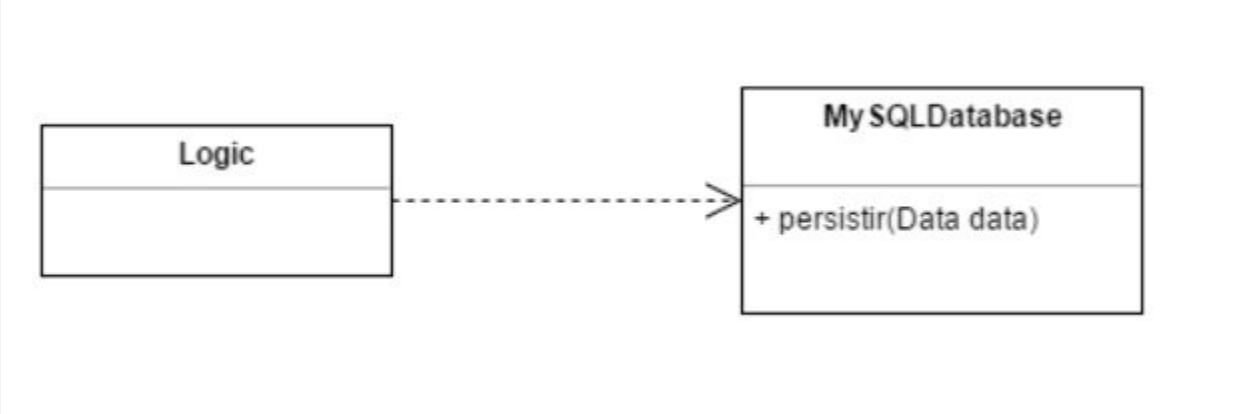
Las clases más estables no deben depender de las clases más inestables. Ambas deben depender de interfaces o clases abstractas.

¿qué hay de la segunda premisa? Pues básicamente nos viene a decir que lo primero en lo que hay que pensar es en definir bien las interfaces y a partir de esa especificación construir las clases que lo implementan, y no al contrario.

Las clases importantes no deben depender de las clases menos importantes. Ambas deben depender de interfaces o clases abstractas.

Las interfaces y clases abstractas no deben depender de las implementaciones. Las implementaciones deben depender de las interfaces y clases abstractas.

En una implementación tradicional por un lado tendremos una clase MySqlDatabase para realizar operaciones sobre la base de datos MySql. Esta clase dará servicio a numerosas operaciones de nuestra aplicación y será más o menos compleja. Cuando queramos implementar la clase Logic para realizar una nueva operación será ésta la que se adapte al comportamiento de MySqlDatabase, es decir Logic usará y dependerá de MySqlDatabase:



Este podría ser el código:

```
public class MySqlDatabase {
    public void persist(Data data){
        // Guarda el dato en una base de datos
MySql
    }
}

public class Logic {
    public void operation(Data input) {
      // Manipular el dato
      Data output = calculateOutput(input);

      // Persistir el resultado
      MySqlDatabase db = new MySqlDatabase();
        db.persist(output);
    }
 }
```

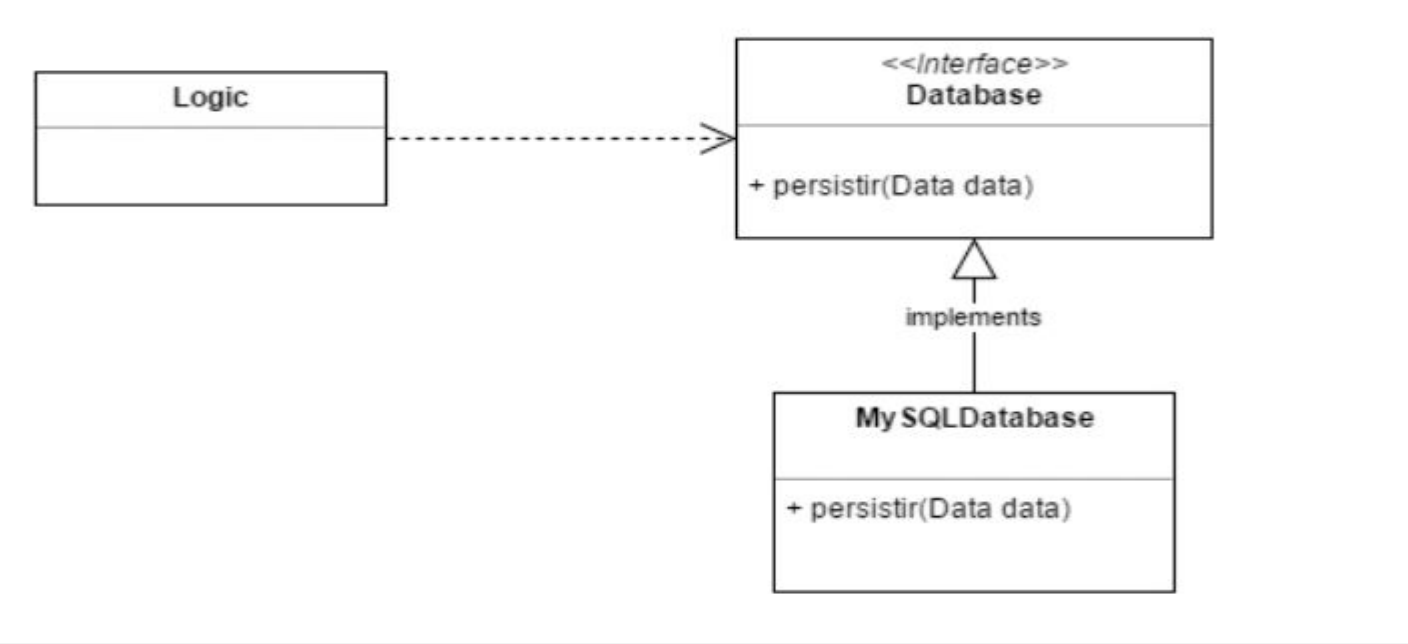
Algunos **problemas** de este planteamiento son:

- *Las clases están **fuertemente acopladas***. Si ahora quiero almacenar el dato en una base de datos PostgreSQL tengo que modificar el código de la clase Logic. Es decir, me veo obligado a hacer cambios en una clase importante y estable, Logic, por culpa de una modificación en una menos estable, el modo de almacenar esos datos.
- *Dificultad para testear*. No puedo probar fácilmente el método 'operation' de la clase 'Logic' sin hacer uso de la base de datos.
- *Incita a la mezcla de responsabilidades*. Cuando un desarrollador, sobretodo si no tiene demasiada experiencia, deba añadir una nueva funcionalidad no tendrá claro dónde hacerlo y porqué: ¿Quién debe tratar las Excepciones lanzadas por MySQL?

Además, si replicamos este planteamiento en todas las clases de lógica y en todas las clases de servicio llegará un momento en que tengamos todo acoplado a todo, resultando en el típico "código cebolla" donde cualquier mínimo cambio puede resultar traumático.

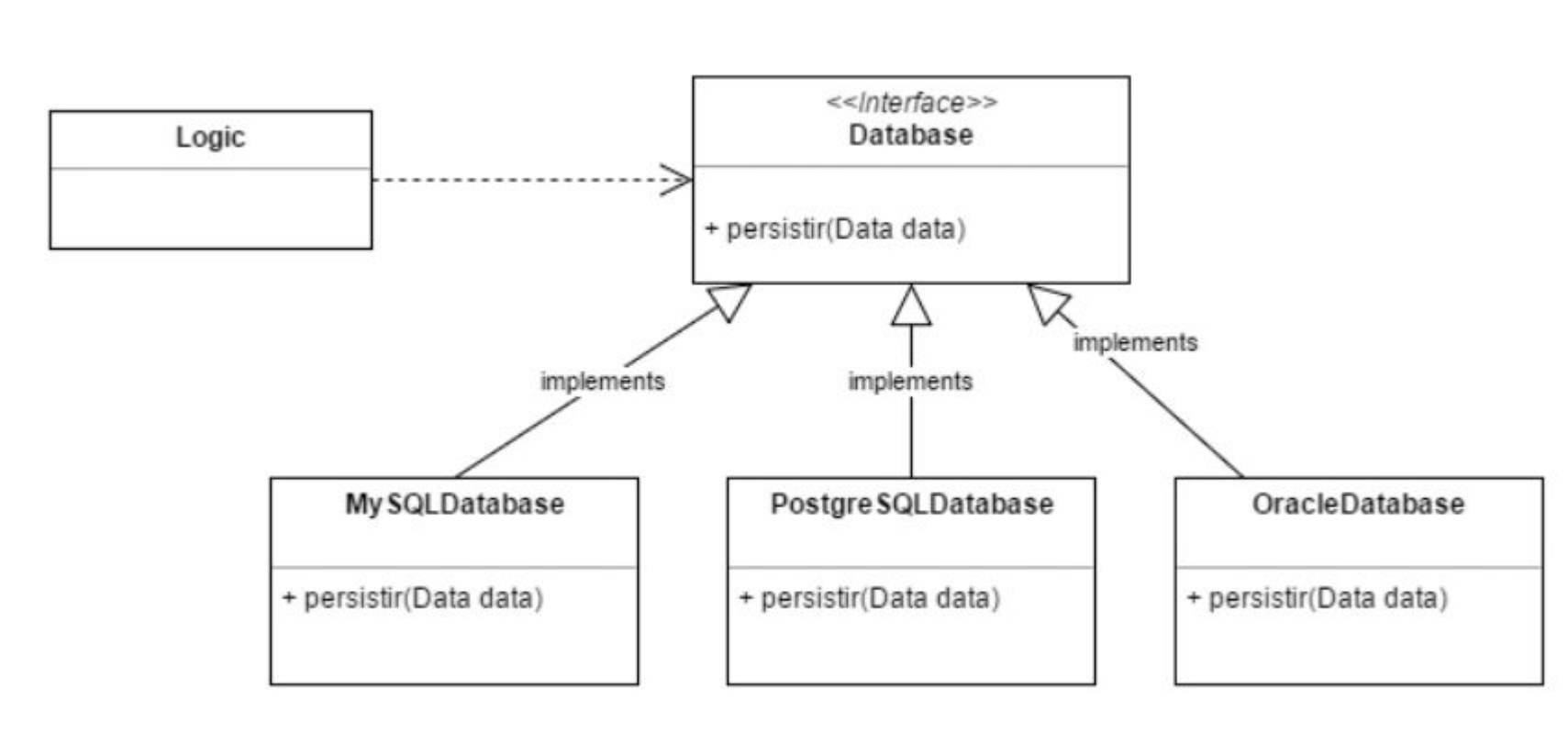
Ejemplo con inversión de dependencia

la clase ‘Logic’, que realiza la lógica de negocio (importante) necesita utilizar los servicios de otra clase para persistir los datos (menos importante), a ‘Logic’ le da igual si estos datos se guardan en MySql, PostgreSql o MongoDB, solo que se guarden.¿ Cómo hacer que sea transparente para la clase ‘Logic’. **Las interfaces** definen qué es lo que puede hacerse con el servicio, **actúan como contratos** o protocolos de comunicación. Logic usará una interfaz y el sistema de persistencia, se implementará de una forma u otra según lo que indique dicha interfaz.

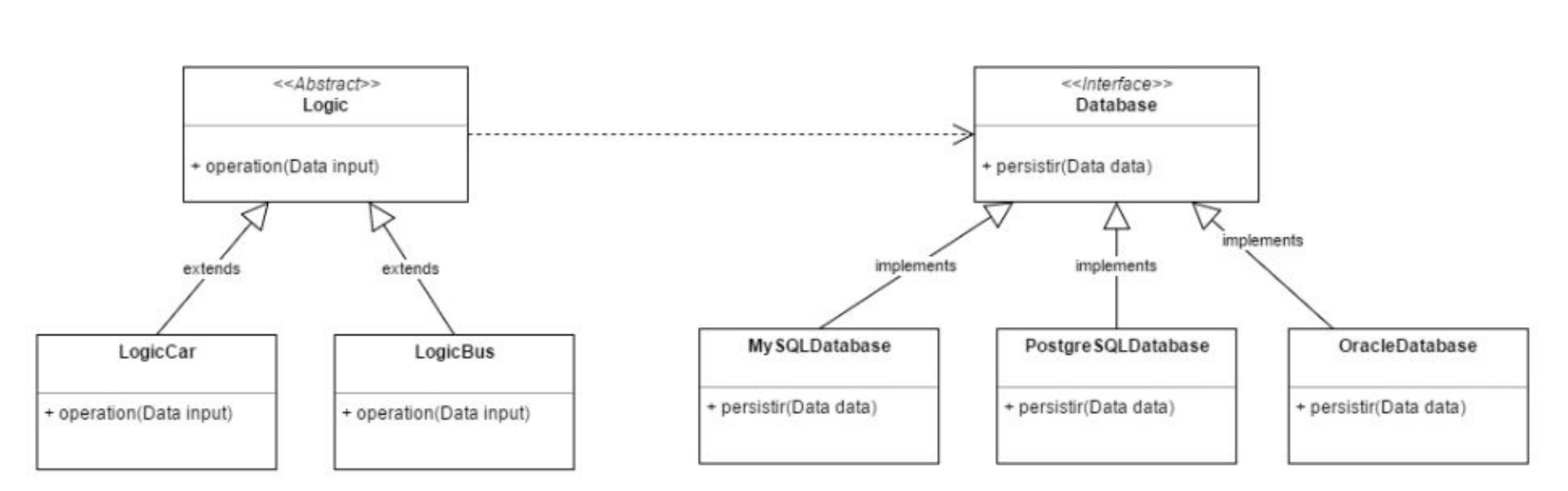


```
public class Logic {
    private final Database database;
    public Logic(Database database) {
        this.database = database;}
    public void operation(Data input) {
        // Manipular el dato
        Data output = calculateOutput(input);

        // Persistir el resultado
        database.persist(output);}}
public interface Database {
    void persist(Data data);}

public class MySqlDatabase implements Database {

    @Override
    public void persist(Data data){
        // Guarda el dato en una base de datos
MySql     }}
```

¿Hemos invertido la dependencia?



Elevar un grado más nuestro diseño



Inyección de Dependencias (DI)

La Inyección de Dependencias (Dependency Injection, DI) es un mecanismo que se encarga de extraer la responsabilidad de creación de instancias de un componente para delegarla en otro. En definitiva no deja de ser una técnica más para separar la construcción de la ejecución.

Inversión de Control (IoC)

La Inversión de Control (Inversion of Control, IoC) se refiere a que **la aplicación delega algún tipo de flujo de control en un tercero**, generalmente un framework. El framework se encargará de gestionar el ciclo de vida de la aplicación e irá notificando eventos a la propia aplicación para que ésta actúe en consecuencia.

DIP – Arquitectura tradicional

UI

Negocio

Acceso a Datos

Componentes

Dependencia

DIP – Arquitectura invertida

Referencias

- **Posters motivacionales**
  http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/
- **PluralSight – SOLID Principles of Object Oriented Design**
  http://www.pluralsight-training.net/microsoft/Courses/TableOfContents?courseName=principles-oo-design
- **Principios de DOO – Bob Martin**
  http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- **Pablo's SOLID Software Development**
  http://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf
- **Principios SOLID con ejemplos reales**
  http://blog.gauffin.org/2012/05/solid-principles-with-real-world-examples/


THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
JUSPAXAR
DANKSCHEEN
BÏYAN
SHUKRIA
TINGKI
YAQHANYELAY
TASHAKKUR ATU
SUKSAMA
EKHMET
MEHRBANI
GRAZIE
KOMAPSUMNIDA
MAAKE
GOZAIMASHITA
EFCHARISTO
BOLZÏN
MERCI
SPASSIBO
SNACHALHUYA
NUHUN
CHALTU
WABEEJA
MAITEKA
YUSPAGABÀTAM
HUI
DHANYABAAD
ANIHA
ATTO
SPASIBO
DENKAUJA
NENACHALHIYA
UNALCHÉESH
MERSI
GUI
HATUR
EKOJU
SIKOMO
MAKETAJ
MINMONCHAR
PALDIES
LAH
SANCO
MERASTAWHY
GAEJTHO
TAVTAPUCH
MEDAWAGSE
AGUYJE
FAKAAUE
BAIKA