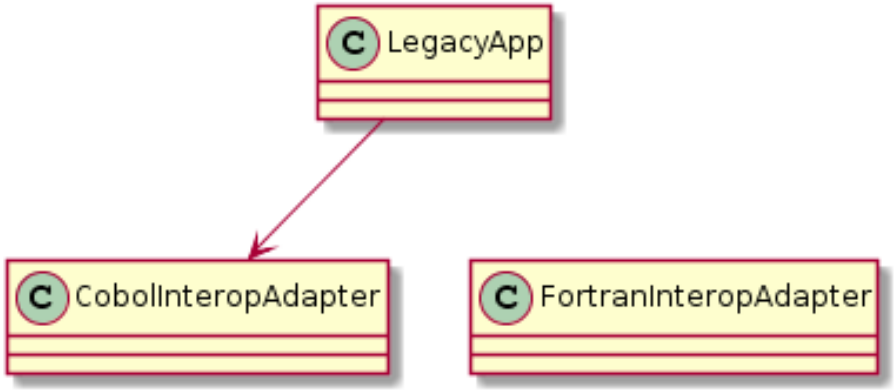
Elementos de diseño

DiP vs DI

Programa fuertemente acoplado

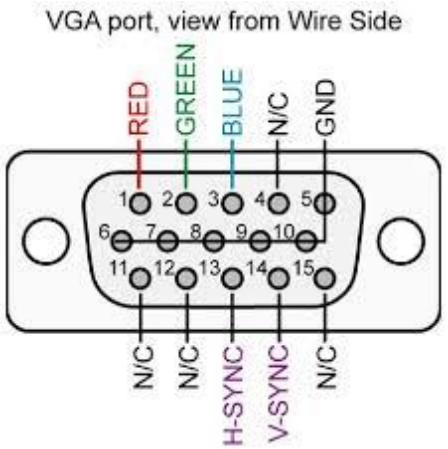
Se necesita intercambiar la dependencia entre `CobolInteropAdapter` y `FortranInteropAdapter` ¿Cómo se puede hacer?

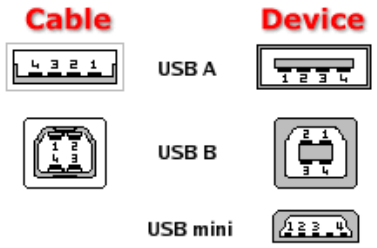


Programa fuertemente acoplado

```
public class LegacyApp implements App {
    public void m1(String ins) {
        XMLAuthenticator authenticator = new XMLAuthenticator();
        CobolInteropAdapter ioAdapter = new CobolInteropAdapter();
        if (authenticator.auth(uname, pwd)) {
            ioadapter.execute(ins);
        } else {
            ioAdapter.sendCloseMsg();
        }
    }
}
```

SOLI(D) - Principio de Inversión de Dependencias.





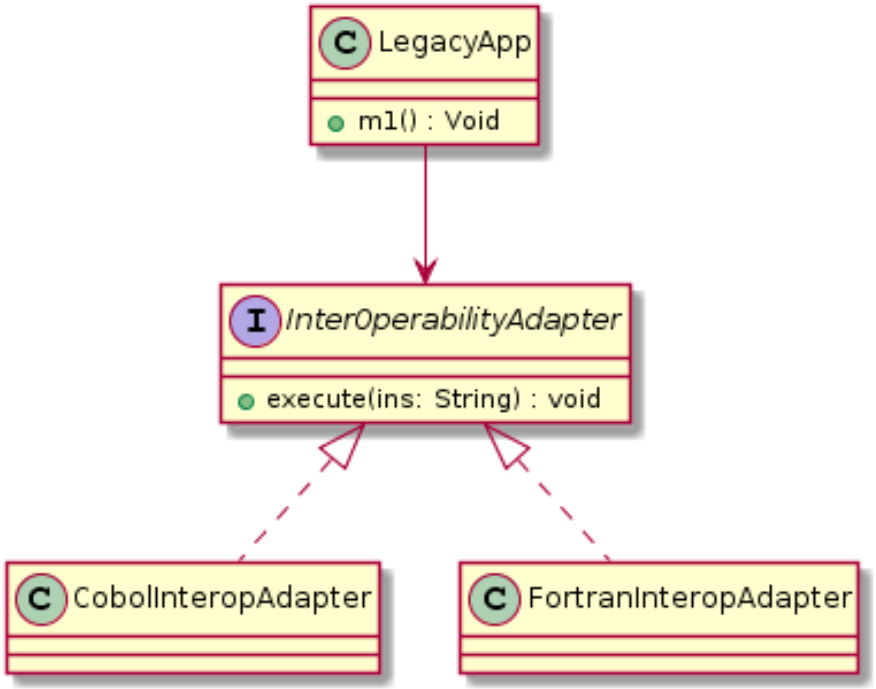
Pin	Signal	Color	Description
1	VCC		+5V
2	D-		Data -
3	D+		Data +
4	GND		Ground

Principio de Inversión de Dependencias.

- Los módulos de alto nivel no deben depender de módulos de bajo nivel.
- Las dependencias deben darse a través de abstracciones:
  - Interfaces o clases abstractas.
- Las abstracciones no deben implicar dependencias en los detalles.

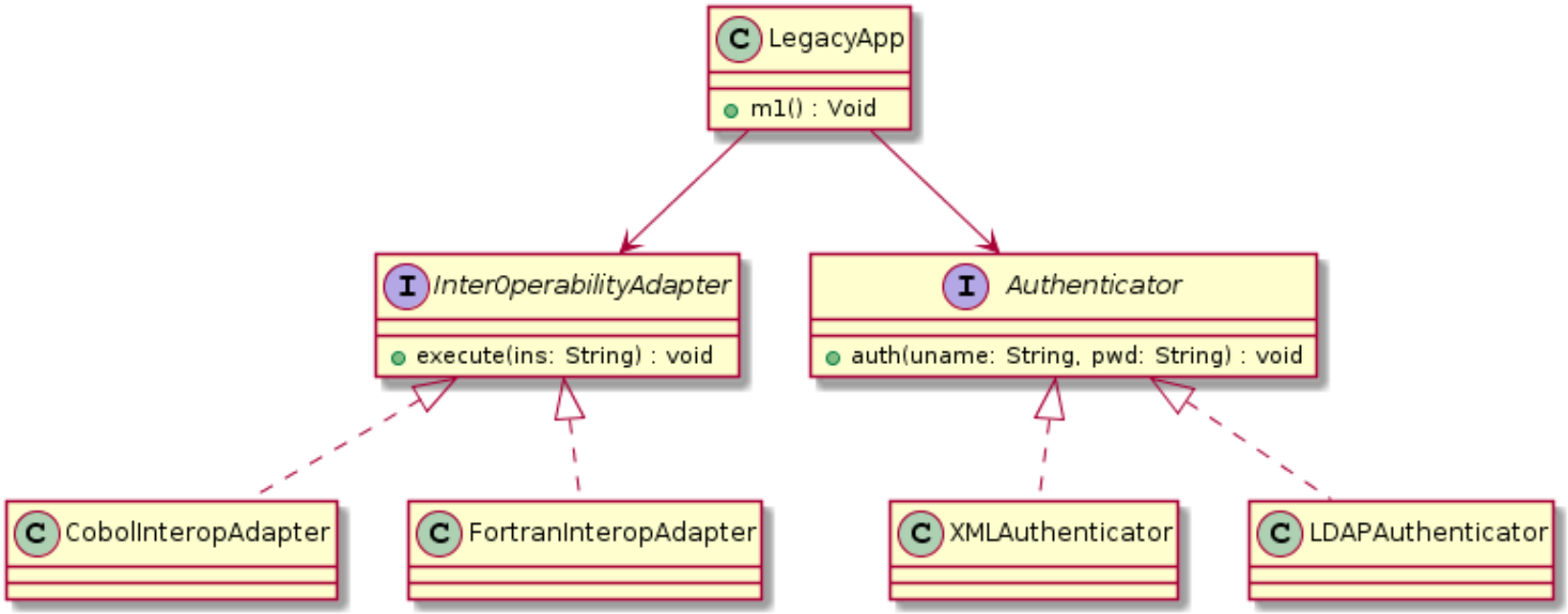
Por qué DiP

- Bajo acoplamiento entre los componentes de alto nivel y sus dependencias.



Revisión a la luz de DiP

- Aplicando en principio de inversión de dependencias:



Revisión a la luz de DiP:

```
public class LegacyApp implements App {
    public void m1(String ins) {
        Authenticator authenticator = ???;
        InterOperabilityAdapter ioAdapter = ???;
        if (authenticator.auth(uname, pwd)) {
            ioadapter.execute(ins);
        } else {
            ioAdapter.sendCloseMsg();
        }
    }
}
```

Sin tener en cuenta los '`???`', el acoplamiento es bajo?

Sustituto de los ??? (alternativas para implementar DiP):

- Construcción de objetos estándar

```
Authenticator authenticator = ???;
```

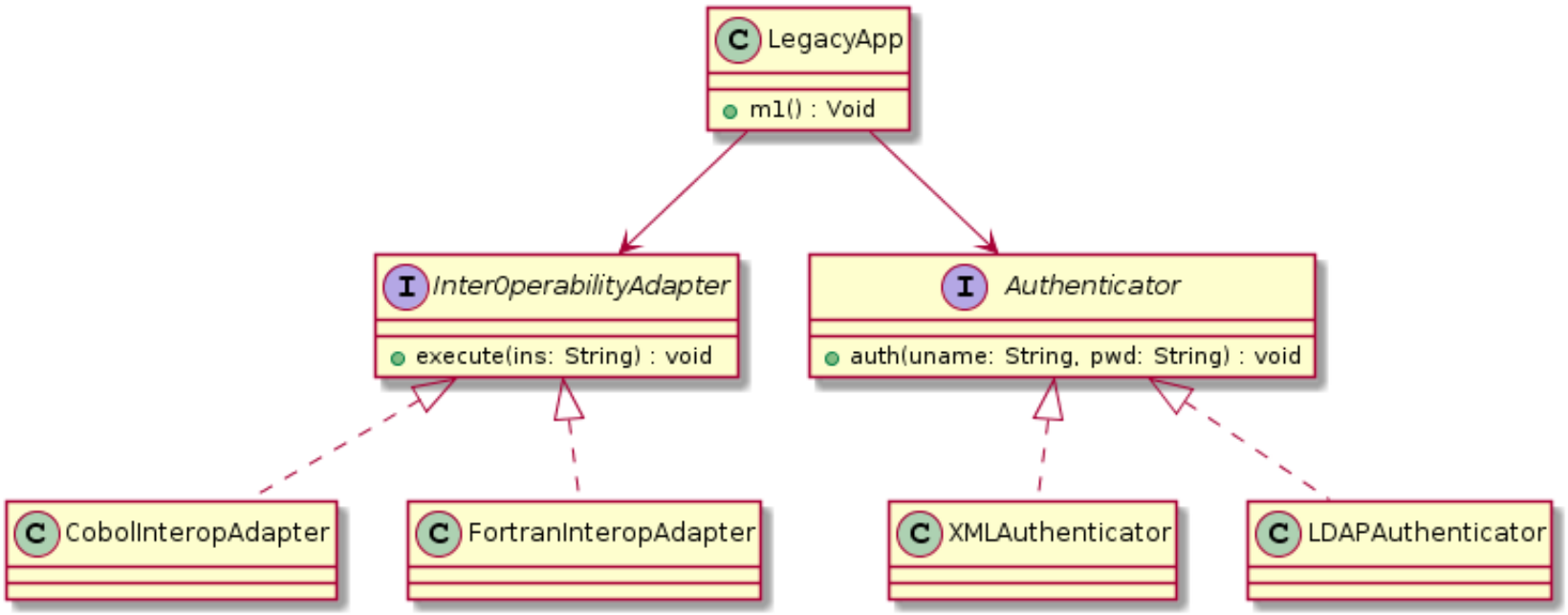
```
Authenticator authenticator = new XMLAuthenticator();
```

- Patrones de diseño
  - Servisa Locator?
  - Abstract Factory / Factory Method?
- Inyección de dependencias

Inyección de dependencias

- Aplicable sólo para soluciones que consideren el principio de inversión de dependencias.
- Delega a un tercero la construcción y asociación de los componentes concretos que un componente de alto nivel requiere.

DiP + diseño para inyección de dependencias



DiP + diseño para inyección de dependencias

```
public class LegacyApp implements App {
    private Authenticator authenticator;
    private InteroperabilityAdapter ioAdapter;
    public void setAuthenticator(Authenticator authenticator) {
        this.authenticator = authenticator;
    }
    public void setIoAdapter(InteroperabilityAdapter ioAdapter) {
        this.ioAdapter = ioAdapter;
    }
    public void m1(String ins) {
        if (authenticator.auth(uname, pwd)) {
            ioadapter.execute(ins);
        } else {
            ioAdapter.sendCloseMsg();
        }
    }
}
```

Inversión de Control - IoC

- Programación basada en interfaces.
- Patrones de creación de bajo acoplamiento.

Contenedores IoC

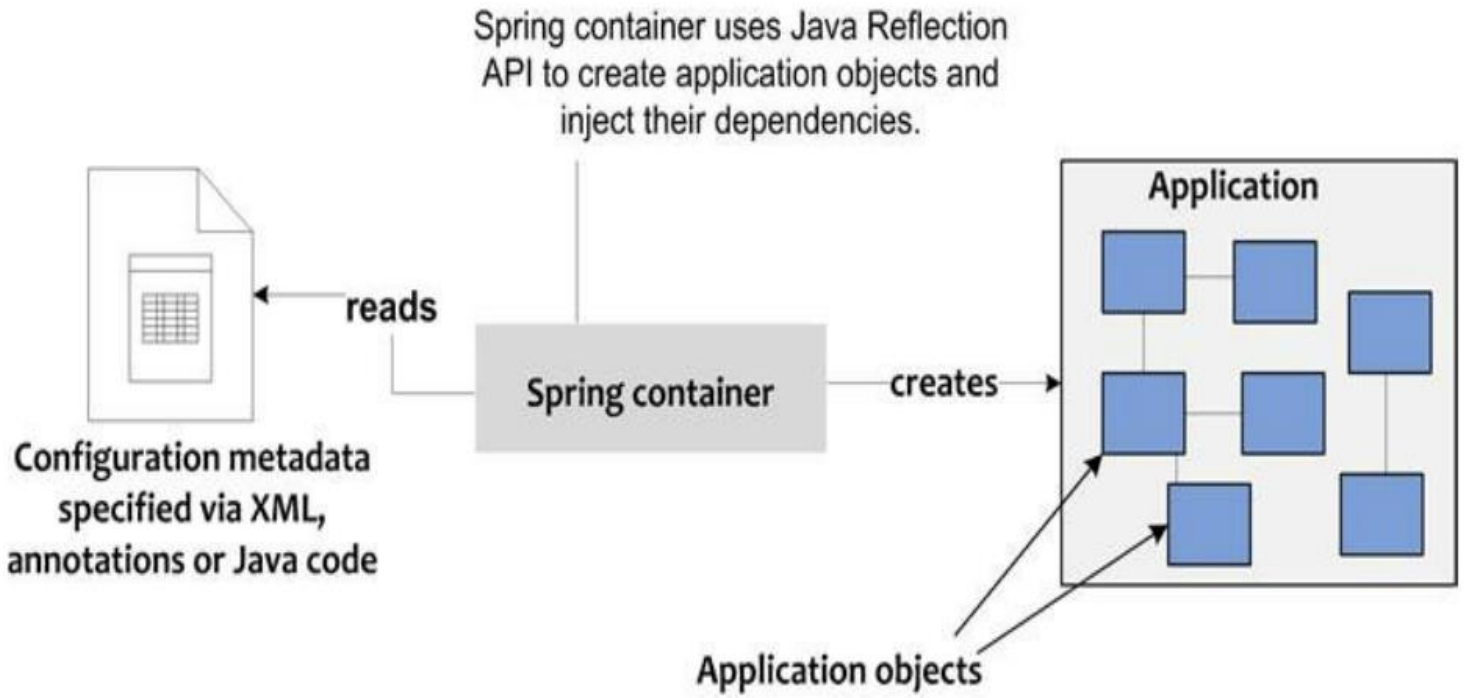
- Su principal función, en el contexto del soporte a la inyección de dependencias, es sustituir el método convencional de creación de objetos:

- Por un método alternativo que garantice la inyección de las dependencias, cuando éstas sean requeridas.
- Este método alternativo dependerá de configuraciones que indiquen qué clase concreta inyectar en cada caso.

Contenedores IoC

- Hivemind
- Google guice
- Spring
- …

Microkernels/contenedores livianos



Guice framework

- Componente para la creación de objetos contemplando la inyección de dependencias:
  - Definiendo una configuración inicial que permite asociar a las abstracciones las clases asociadas:

```
public class LegacyModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Authenticator.class).to(LDAPAuthenticator.class);
    bind(InterOperabilityAdapter.class).to(CobolInteropAdapter.class);
    bind(App.class).to(LegacyApp.class);
  }
}
```

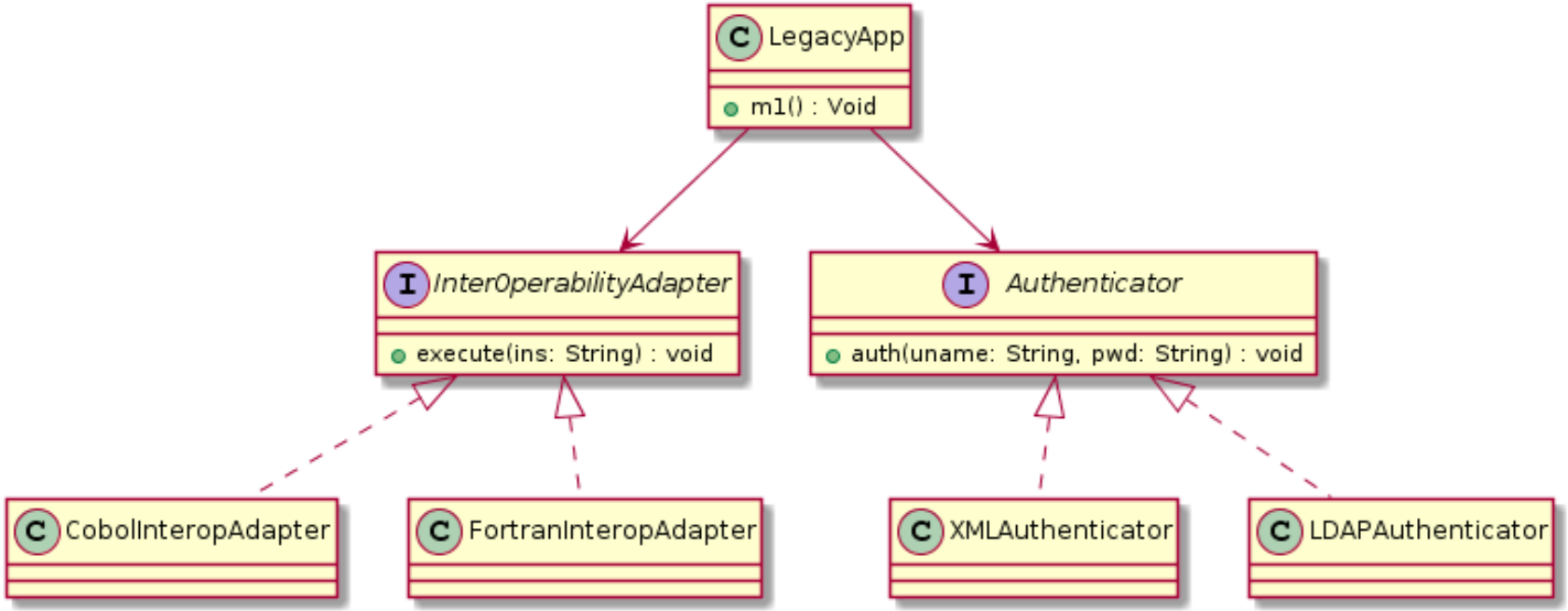
Uso del Framework

- Para utilizar una configuración específica se asocia en el main.
- Se utiliza el `injector` para obtener una clase de una instancia asociada.

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector(new LegacyModule());
    App application = injector.getInstance(App.class);
    ...
  }
```

Configuración de Guice

- Configuración alternativa: basada en anotaciones



Inyección en las Clases

La inyección se basa en anotaciones:

- @Inject: anotación de clase, para asociar a una clase las instancias que se inyectan. Guice inspecciona el constructor anotado y busca los valores asociados a cada parámetro.

```
public class LegacyApp implements App {
  private final Authenticator authenticator;
  private final InteroperabilityAdapter ioAdapter;

  @Inject
  public LegacyApp(Authenticator authenticator, InteroperabilityAdapter ioAdapter) {
    this.authenticator = authenticator;
    this.ioAdapter = ioAdapter;
  }
…
}
```

Alcance (Scope)

Scope	Alcance
`@RequestScoped`	Crea una nueva instancia en cada inyección. Es la utilizada por defecto en Guice.
`@Singleton`	Crea una única instancia que comparte en cada inyección.
`@SessionScoped`	Crea una instancia por sesión (utilizada en aplicaciones Web)

Hay que tener cuidado en las instancias `@Singleton` y `@SessionScoped` que deben ser Thread Safe, i.e, pensar en que los métodos y por consiguiente los atributos de la misma instancia, pueden ser accedida por varios hilos simultáneamente.

Implementación particular

- Guice busca la clase asociada en el bind (para nuestro ejemplo LDAPAuthenticator) e inyecta una instancia a quien requiera algo de tipo ‘Authenticator’:

```
@Singleton
public class LDAPAuthenticator implements Authenticator {
  public boolean auth(String uname, String password) {
      boolean credentialsOk = false;
      // check LDAP
      // ...
      return credentialsOk;
  }
  ...
}
```

Implementación particular

- En lugar de utilizar las anotaciones del alcance en las clases se puede definir el scope en el bind:

```
public class LegacyModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Authenticator.class)
        .to(LDAPAuthenticator.class)
        .in(Singleton.class);
    bind(InteroperabilityAdapter.class).to(CobolInteropAdapter.class);
    bind(App.class).to(LegacyApp.class);
  }
}
```