DI(依存性注入)について

伊藤　結

ところで、DIと聞いて
ピンとくる方はいますか？

DI(依存性注入)とは

依存性の注入（英: Dependency injection）とは、コンポーネント間の依存関係をプログラムのソースコードから 排除し、外部の設定ファイルなどで注入できるようにするソフトウェアパターンである。英語の頭文字からDIと略される。

wikipediaより
https://ja.wikipedia.org/wiki/依存性の注入

なるほど。わからん。

DI(依存性注入)とは

依存性の注入（英: Dependency injection）とは、コンポーネント間の依存関係をプログラムのソースコードから 排除し、外部の設定ファイルなどで注入できるようにするソフトウェアパターンである。英語の頭文字からDIと略される。

wikipediaより
https://ja.wikipedia.org/wiki/依存性の注入

言葉の意味から考えてみよう！

依存性ってなに？

```
public class Siphon implements BrewingMethod {
   @Override
   public String brew() {
       return "サイフォンでいれたコーヒー ";
   }
}

public class CoffeeShop {
   public String brewCoffee() {
       Siphone siphone = new Siphone();
       return siphone.brew() + "が出来上がりました [_]P";
   }
}

public class CoffeeShopApp {
   public static void main(String... args) {
       CoffeeShop coffeeShop = new CoffeeShopApp();
       coffeeShop.brewCoffee();
   }
}
```

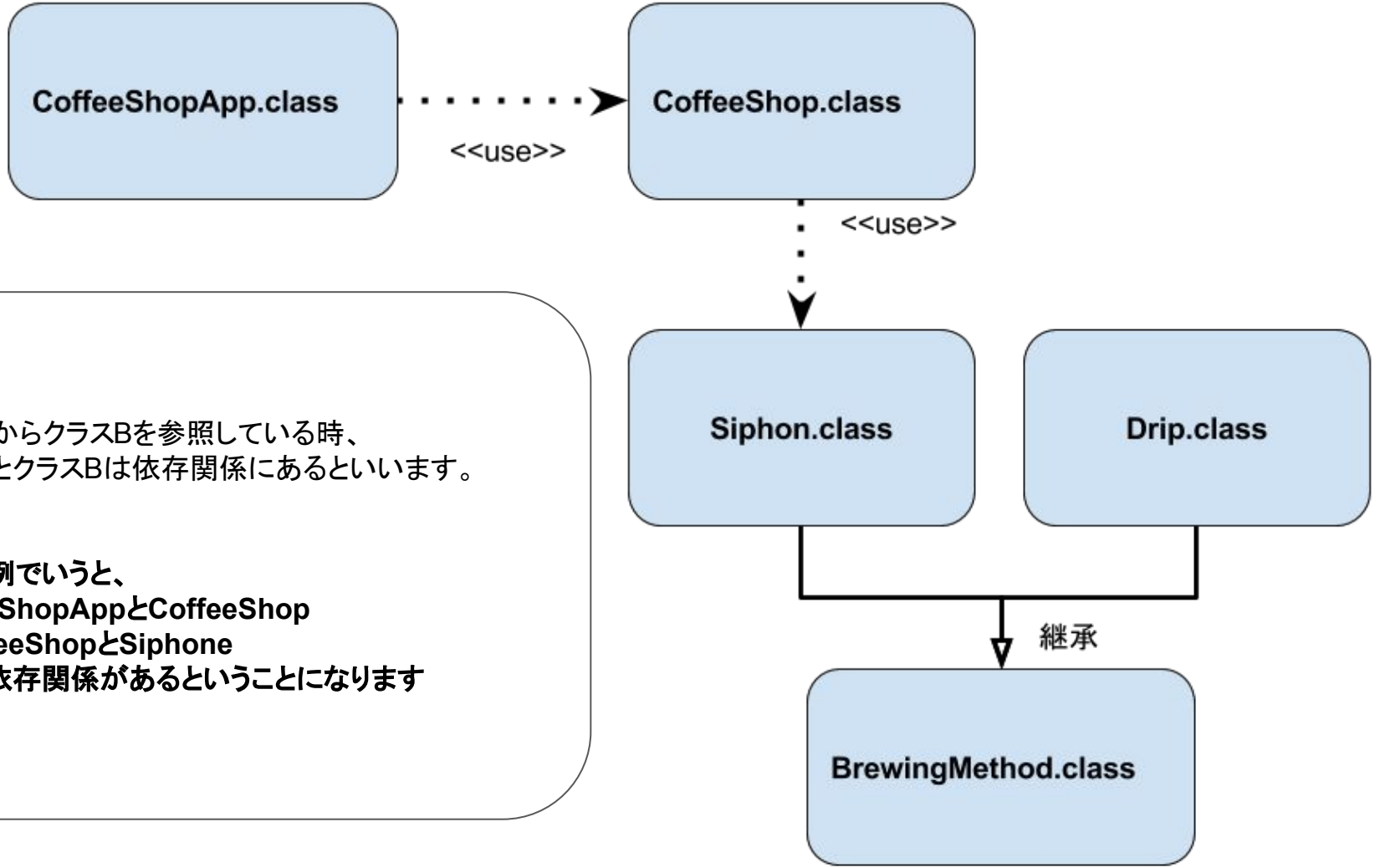
```
public class Siphon implements BrewingMethod {
    @Override
    public String brew() {
        return "サイフォンでいれたコーヒー ";
    }
}

public class CoffeeShop {
    public String brewCoffee() {
        Siphone siphone = new Siphone();
        return siphone.brew() + "が出来上がりました [_]P";
    }
}

public class CoffeeShopApp {
    public static void main(String... args) {
        CoffeeShop coffeeShop = new CoffeeShopApp();
        coffeeShop.brewCoffee();
    }
}
```

BrewingMethod(抽出方法)の実装クラス

プログラムの実行クラス



クラスAからクラスBを参照している時、
クラスAとクラスBは依存関係にあるといいます。

今回の例でいうと、
CfeeShopAppとCoffeeShop
CoffeeShopとSiphone
の間で依存関係があるということになります

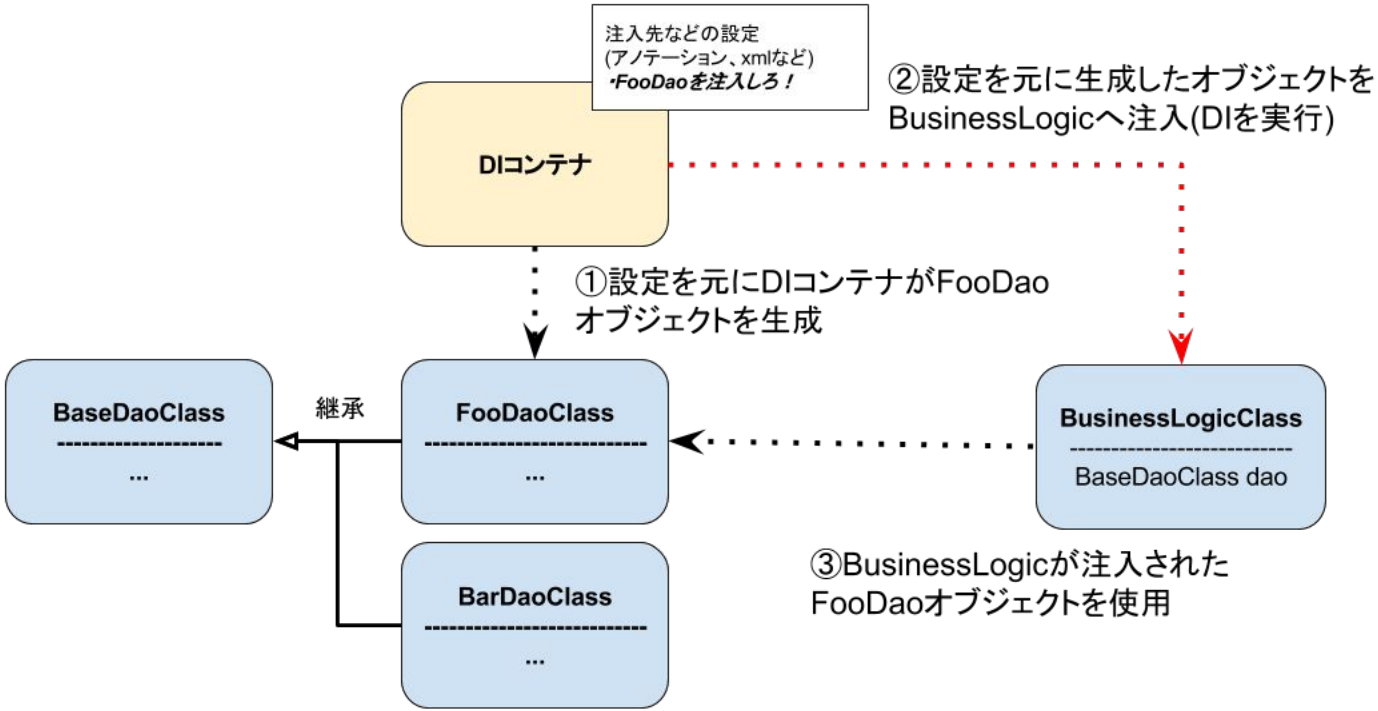
注入ってなに？

実際のコードで説明します

DIコンテナを使って、
実際にDIをやってみよう！

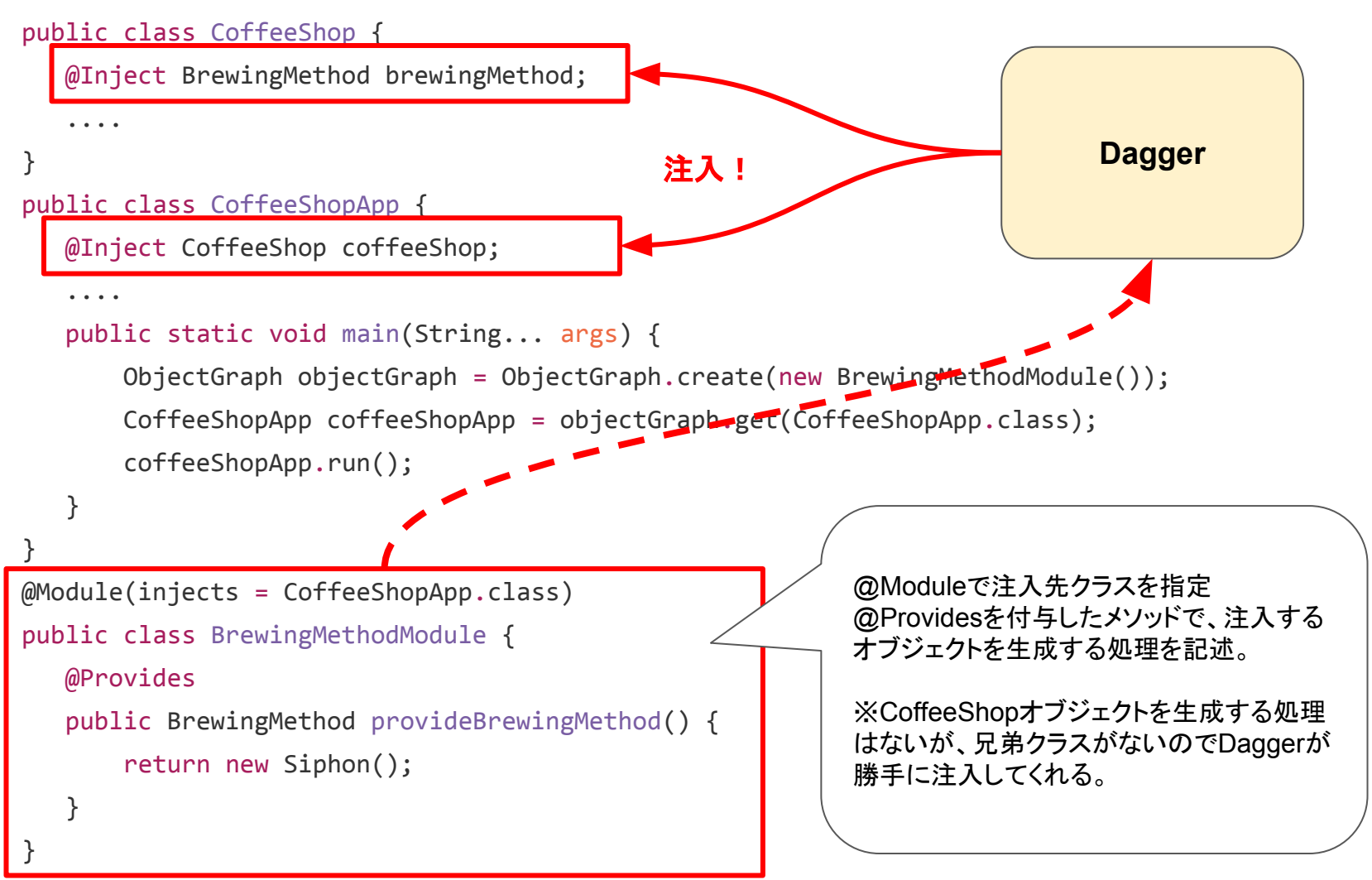
DIコンテナとは

- すごく簡単に言うと、「XXにこのクラスのオブジェクトを注入してね」
  という設定を書いておくとDI(依存性の注入)を実行してくれるやつのこと。



今回はJava向けのDaggerというDIコンテナを使います。

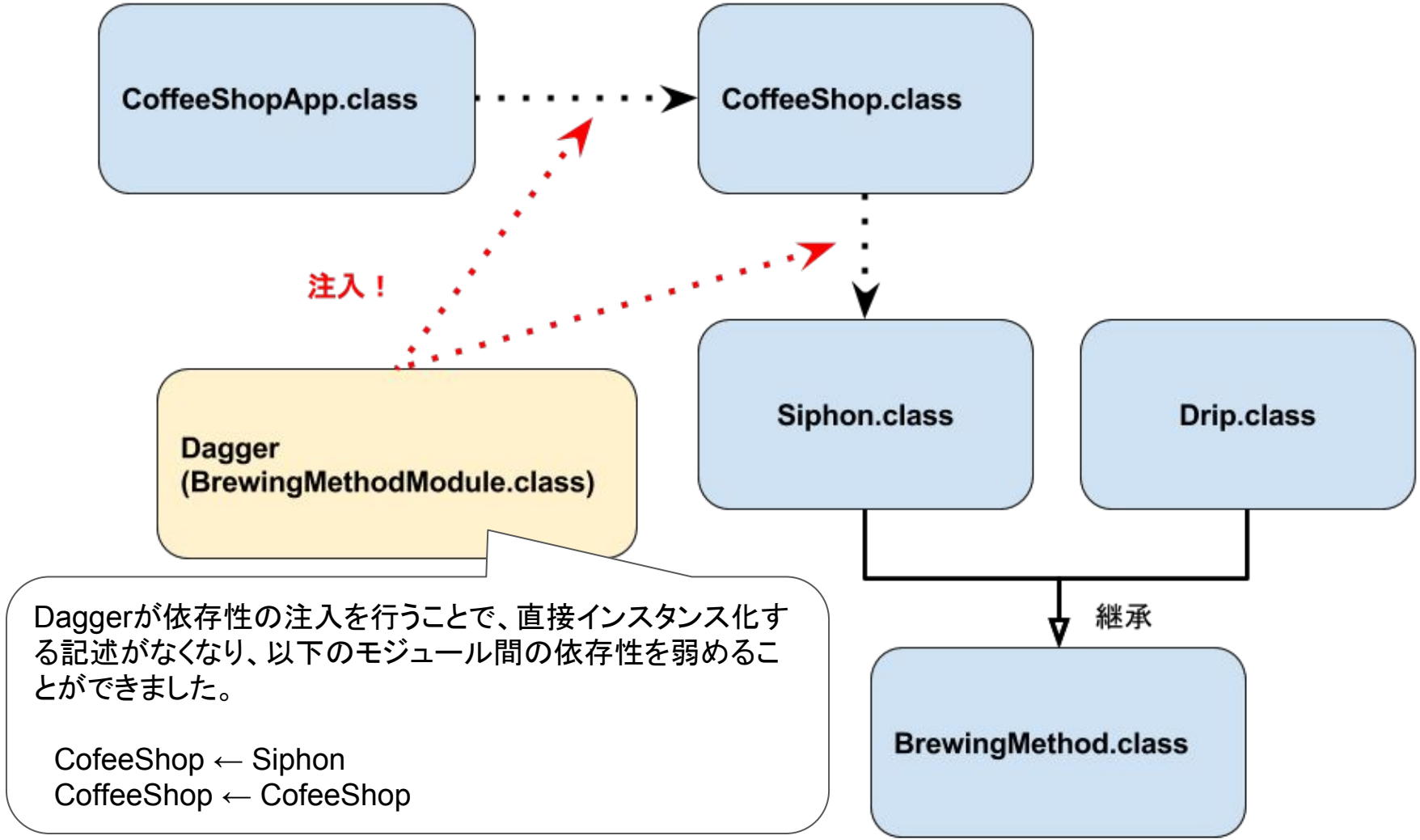
```
public class CoffeeShop {
   @Inject BrewingMethod brewingMethod;
   ....
}
public class CoffeeShopApp {
   @Inject CoffeeShop coffeeShop;
   ....
   public static void main(String... args) {
       ObjectGraph objectGraph = ObjectGraph.create(new BrewingMethodModule());
       CoffeeShopApp coffeeShopApp = objectGraph.get(CoffeeShopApp.class);
       coffeeShopApp.run();
   }
}
@Module(injects = CoffeeShopApp.class)
public class BrewingMethodModule {
   @Provides
   public BrewingMethod provideBrewingMethod() {
       return new Siphon();
   }
}
```


public class CoffeeShop {
@Inject BrewingMethod brewingMethod;
....
}
public class CoffeeShopApp {
@Inject CoffeeShop coffeeShop;
....
public static void main(String... args) {
ObjectGraph objectGraph = ObjectGraph.create(new BrewingMethodModule());
CoffeeShopApp coffeeShopApp = objectGraph.get(CoffeeShopApp.class);
coffeeShopApp.run();
}
}
@Module(injects = CoffeeShopApp.class)
public class BrewingMethodModule {
@Provides
public BrewingMethod provideBrewingMethod() {
return new Siphon();
}
}
注入！
Dagger
@Moduleで注入先クラスを指定
@Providesを付与したメソッドで、注入する
オブジェクトを生成する処理を記述。
※CoffeeShopオブジェクトを生成する処理
はないが、兄弟クラスがないのでDaggerが
勝手に注入してくれる。

```
public class CoffeeShop {
    @Inject BrewingMethod brewingMethod;
    ....
}
public class CoffeeShopApp {
    @Inject CoffeeShop coffeeShop;
    ....
    public static void main(String... args) {
        ObjectGraph objectGraph = ObjectGraph.create(new BrewingMethodModule());
        CoffeeShopApp coffeeShopApp = objectGraph.get(CoffeeShopApp.class);
        coffeeShopApp.run();
    }
}
@Module(injects = CoffeeShopApp.class)
public class BrewingMethodModule {
    @Provides
    public BrewingMethod provideBrewingMethod() {
        return new Siphon();
    }
}
```

ObjectGraph.createで
ObjectGraphオブジェクトを生成。

ObjectGraph#getで、依存性を注入済みの
CoffeeShopAppオブジェクトが返却される。



Daggerが依存性の注入を行うことで、直接インスタンス化する記述がなくなり、以下のモジュール間の依存性を弱めることができました。

CofeeShop ← Siphon
CoffeeShop ← CofeeShop

で、DI使うと
なにがよくなるの？

DIを使う場合の利点

- モジュール間の結合度を弱めることができる(保守性が高まる)
- 単体テストが楽になる
  (Aモジュールが完成する前にBモジュールをテストできる、外から特定のモジュールをモックに差し替えることができる)

実際やってみて、特にテストコードを書くとき、Mockオブジェクトに差し替えてテストながせるのが心地よかったです。

```
public class CoffeeShopTest {
    @Inject CoffeeShop coffeeShop;
    @Inject BrewingMethod brewingMethod;
    @Before
    public void setUp() {
        ObjectGraph.create(new TestModule()).inject(this);
    }
    @Module(includes = BrewingMethodModule.class,
            injects = CoffeeShopTest.class,
            overrides = true)
    static class TestModule {
        @Provides
        @Singleton
        public BrewingMethod provideBrewingMethod() {
            return Mockito.mock(BrewingMethod.class);
        }
    }
    @Test
    public void testBrewCoffee() {
        Mockito.when(brewingMethod.brew()).thenReturn("テストコーヒー");
        String result = coffeeShop.brewCoffee();
        Mockito.verify(brewingMethod, Mockito.times(1)).brew();
        assertThat(result, is("テストコーヒーが出来上がりました [_]P"));
    }
```

```java
public class CoffeeShopTest {
    @Inject CoffeeShop coffeeShop;
    @Inject BrewingMethod brewingMethod;
    @Before
    public void setUp() {
        ObjectGraph.create(new TestModule()).inject(this);
    }
    @Module(includes = BrewingMethodModule.class,
            injects = CoffeeShopTest.class,
            overrides = true)
    static class TestModule {
        @Provides
        @Singleton
        public BrewingMethod provideBrewingMethod() {
            return Mockito.mock(BrewingMethod.class);
        }
    }
    @Test
    public void testBrewCoffee() {
        Mockito.when(brewingMethod.brew()).thenReturn("テストコーヒー");
        String result = coffeeShop.brewCoffee();
        Mockito.verify(brewingMethod, Mockito.times(1)).brew();
        assertThat(result, is("テストコーヒーが出来上がりました [_]P"));
    }
```

BrewingMethodModuleを、
BrewingMethodのモックを返却する処理で
置き換える。

本クラスのCoffeeShopオブジェクト、
BrewingMethodオブジェクト
BrewingMethodのMockを注入する
(Singletonアノテーションを付与しているの
ですべて同じ)。

```
public class CoffeeShopTest {
    @Inject CoffeeShop coffeeShop;
    @Inject BrewingMethod brewingMethod;
    @Before
    public void setUp() {
        ObjectGraph.create(new TestModule()).inject(this);
    }
    @Module(includes = BrewingMethodModule.class,
            injects = CoffeeShopTest.class,
            overrides = true)
    static class TestModule {
        @Provides
        @Singleton
        public BrewingMethod provideBrewingMethod() {
            return Mockito.mock(BrewingMethod.class);
        }
    }
    @Test
    public void testBrewCoffee() {
        Mockito.when(brewingMethod.brew()).thenReturn("テストコーヒー");
        String result = coffeeShop.brewCoffee();
        Mockito.verify(brewingMethod, Mockito.times(1)).brew();
        assertThat(result, is("テストコーヒーが出来上がりました [_]P"));
    }
```

本クラスへの依存性注入の実行

注入した、モックオブジェクトの振る舞いを設定して、テストを実行

DIコンテナって
他にはどんなのがあるの？

代表的なDIコンテナ

- Spring Framework
- JavaEE (CDI)
- Seasar2
- Zend Framework 2

- Dagger
- Dagger2
- Proton
- RoboGuice

なんかJavaばっかりじゃね？

言語とDIコンテナ

- Javaなどの静的言語では色々と種類があるが、
  RubyやPythonなどの動的言語ではあまり使われていない(と思う)。

> 私は一時DIについて関心を持って、いろいろ調べてみたし、自分でDIコンテナを実装してみたりもした。でも、RubyでならDIコンテナがわずか20行で記述できる上、よく考えてみたら、その20行も、なくてもほぼ同じことが簡単に実現できることに気がついた時、DIってのは硬直した言語のための技術なんだと気がついた。
>
> Matzにっき より
> http://www.rubyist.net/~matz/20091003.html

どんな所でDIを使うべき？

DIの使いどころ

- (当たり前ですが) DIを前提にしたフレームワークを使用するとき。
- Javaを使用しており、テスト駆動開発とかテストコードを書くことが前提のプロジェクト。
- 実装があるモジュールに依存してるが、そのモジュールは未完成、でも単体テストを始めないといけないんだよ～みたいなとき。

採用したときのコストに比べて、モジュール間の結合度を弱めることが重要な時に使用するべき。
何でもかんでもDIすればいいってもんでもない。

ご清聴ありがとうございました