**Dependency Injection**

**Dependency injection** is a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time.

More complicated implementations, such as **Spring**, **Google Juice**, and **Microsoft Managed Extensibility Framework (MEF)** automate this procedure.

**Benefits**

One benefit of using the dependency injection approach is the reduction of **boilerplate code** in the application objects since all work to initialize or set up dependencies is handled by a provider component.

Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in **unit testing**, as it is easy to inject a fake implementation of a service into the object being tested by changing the configuration file, or overriding component registrations at run-time.

Furthermore, dependency injection facilitates the writing of testable code.

# Example

**Highly coupled dependency:**

class Taxi{

protected $id;

protected $driver;

protected $pricePerKm;

protected $taxVAT;

public function \_\_construct($taxi\_id, $config)

{

$this->id = $taxi\_id;

$diver = UserModel::getDefaultDriverOfTaxi($this->id);

$this->setDriver($driver);

$this->setPricePerKm($config['price\_per\_km']);

$this->setTaxVAT($config['tax\_vat']);

}

public function setDriver(User $driver)

{

$this->driver = $driver;

return $this;

}

public function calFee($distance)

{

//@todo cal the fee here

return $distance\*$this->getPricePerKm()\*(1+$this->getTaxVAT());

}

}

What if config is not an array but an object.

**Revert the code:**

interface ITaxi{

public function calFee($distance);

}

class Taxi implement ITaxi{

public function \_\_construct($id\_taxi, Config $config)

{

$this->id = $taxi\_id;

$config->setConfigurationToTaxi($this);

}

}

class Service{

protected $arrTaxi;

/\*\*

\* @return ITaxi Trả về một Interface chỉ có phương thức tính tiền.

\*/

public function getTaxi($taxi\_id)

{

if( isset($this->arrTaxi[$taxi\_id]) )

{

return $this->arrTaxi[$taxi\_id];

}

$taxi = new Taxi($taxi\_id, new Config());

$diver = UserModel::getDefaultDriverOfTaxi($taxi\_id);

$taxi->setDriver($driver);

return $this->arrTaxi[$taxi\_id] = $taxi;

}

}

class Config{

public function setConfigurationToTaxi(Taxi $taxi)

{

$taxi->setPricePerKm(

$this->getConfigPrice()

);

$taxi->setTaxVAT(

$this->getConfigVAT()

);

}

public function getConfigPrice()

{

//Lấy giá trị giá tiền trên mỗi từ database, const...

}

public function getConfigVAT()

{

//Lấy giá trị thuế VAT từ database, const...

}

}

**Implement:**

$taxi = Service::getTaxi($taxi\_id); //An instance of ITaxi

$fee = $taxi->calFee($distance = 10);

# Using Gin

## Step 1. Inheriting the GIN module

<module>  
  ...  
  <inherits name="com.google.gwt.inject.Inject"/>  
  ...  
</module>

## Step 2. Defining the Ginjector

Declare an interface with methods that return the desired types:

public interface MyWidgetGinjector extends Ginjector {  
  MyWidgetMainPanel getMainPanel();  
}

Experienced GWT users will notice the similarity to the way GWT image bundles and messages are done: You simply create a method for each object type you want to create, and the an implementation of the interface gets generated for you **at compile time**.

Note that you only need to create injector methods for classes that you would directly access in your top-level initialization code, such as the UI classes to install in your RootPanel. You don't need to create injector methods for lower-level classes that will be automatically injected. So for example, if Class A uses class B which uses class C, you only need to create an injector method for A, as the other classes B and C will automatically be injected into A.

In other words, injector methods provide a bridge between the Guice and non-Guice world.

## Step 3. Declare bindings

The next step is to bind the various classes and providers using a Guice module. The module class looks almost exactly like it would in regular Guice (We use the GinModule and AbstractGinModule instead of Module and AbstractModule.) Here's an example module:

public class MyWidgetClientModule extends AbstractGinModule {  
  protected void configure() {  
    bind(MyWidgetMainPanel.class).in(Singleton.class);  
  }  
}

Note that if GIN can't find a binding for a class, it falls back to calling GWT.create() on that class. What this means that image bundles and translated messages will just magically work.

We emulate most of Guice's types but not modules, because:

* We need to be able to develop GIN at our own pace.
* We will probably never support toInstance(...) and the like because we do our work with modules at compile time, not runtime.

For compatibility with regular Guice we provide a GinModuleAdapter class, which makes your GinModule available as a Module.

## Step 4. Associating the module with the injector

Add the GinModules annotation to your Ginjector, specifying the module(s) needed to configure the application.

@GinModules(MyWidgetClientModule.class)  
public interface MyWidgetGinjector extends Ginjector {  
  MyWidgetMainPanel getMainPanel();  
}

Current project layout:

MyWidgetProject/  
    client/  
        MyWidget.java  
        MyWidgetGinjector.java  
        MyWidgetMainPanel.java  
        MyWidgetClientModule.java  
    public/  
    server/

## Step 5. Creating the Ginjector

To create the injector instance, use the standard GWT.create() call. This can be done during static initialization:

public class MyWidget implements EntryPoint {  
  private final MyWidgetGinjector injector = GWT.create(MyWidgetGinjector.class);  
  
  public void onModuleLoad() {  
    MyWidgetMainPanel mainPanel = injector.getMainPanel();  
    RootPanel.get().add(mainPanel);  
  }  
  
}