**Requirement:** We will get some Customer Address and we need to validate the address. After some evaluation we thought of using Google Address Validation Service.

**Legacy(Bad) Approach:**

Just create an AddressVerificationService class and implement the logic.

Assume GoogleAddressVerificationService is a service provided by Google which takes Address as a String and Return longitude/latitude.

class AddressVerificationService

{

//This method validates the given address and return longitude/latitude details.

public String validateAddress(String address)

{

GoogleAddressVerificationService gavs = new GoogleAddressVerificationService();

String result = gavs.validateAddress(address);

return result;

}

}

**Issues with this approach:**

1. If you want to change your Address Verification Service Provider you need to change the logic.

2. You can't Unit Test with some Dummy AddressVerificationService (Using Mock Objects)

Due to some reason Client ask us to support multiple AddressVerificationService Providers and we need to determine which service to use at runtime.

To accomidate this you may thought of changing the above class as below:

class AddressVerificationService

{

//This method validates the given address and return longitude/latitude details.

public String validateAddress(String address)

{

String result = null;

int serviceCode = 2; // read this code value from a config file

if(serviceCode == 1)

{

GoogleAddressVerificationService googleAVS = new GoogleAddressVerificationService();

result = googleAVS.validateAddress(address);

} else if(serviceCode == 2)

{

YahooAddressVerificationService yahooAVS = new YahooAddressVerificationService();

result = yahooAVS.validateAddress(address);

}

return result;

}

}

**Issues with this approach:**

1. Whenever you need to support a new Service Provider you need to add/change logic using if-else-if.

2. You can't Unit Test with some Dummy AddressVerificationService (Using Mock Objects)

**IOC/DI Approach:**

In the above approaches AddressVerificationService is taking the control of creating its dependencies.

So whenever there is a change in its dependencies the AddressVerificationService will change.

Now let us rewrite the AddressVerificationService using IOC/DI pattern.

class AddressVerificationService {

private AddressVerificationServiceProvider serviceProvider;

public AddressVerificationService(AddressVerificationServiceProvider serviceProvider) {

this.serviceProvider = serviceProvider;

}

public String validateAddress(String address)

{

return this.serviceProvider.validateAddress(address);

}

}

interface AddressVerificationServiceProvider

{

public String validateAddress(String address);

}

Here we are injecting the AddressVerificationService dependency AddressVerificationServiceProvider.

Now let us implement the AddressVerificationServiceProvider with multiple provider services.

class YahooAVS implements AddressVerificationServiceProvider

{

@Override

public String validateAddress(String address) {

System.out.println("Verifying address using YAHOO AddressVerificationService");

return yahooAVSAPI.validate(address);

}

}

class GoogleAVS implements AddressVerificationServiceProvider

{

@Override

public String validateAddress(String address) {

System.out.println("Verifying address using Google AddressVerificationService");

return googleAVSAPI.validate(address);

}

}

Now the Client can choose which Service Provider's service to use as follows:

AddressVerificationService verificationService = null;

AddressVerificationServiceProvider provider = null;

provider = new YahooAVS();//to use YAHOO AVS

provider = new GoogleAVS();//to use Google AVS

verificationService = new AddressVerificationService(provider);

String lnl = verificationService.validateAddress("HitechCity, Hyderabad");

System.out.println(lnl);

For Unit Testing we can implement a Mock AddressVerificationServiceProvider.

class MockAVS implements AddressVerificationServiceProvider

{

@Override

public String validateAddress(String address) {

System.out.println("Verifying address using MOCK AddressVerificationService");

return "<response><longitude>123</longitude><latitude>4567</latitude>";

}

}

AddressVerificationServiceProvider provider = null;

provider = new MockAVS();//to use MOCK AVS

AddressVerificationServiceIOC verificationService = new AddressVerificationServiceIOC(provider);

String lnl = verificationService.validateAddress("Somajiguda, Hyderabad");

System.out.println(lnl);

With this approach we elemenated the issues with above Non-IOC/DI based approaches.

1. We can provide support for as many Provides as we wish. Just implement AddressVerificationServiceProvider and inject it.

2. We can unit test using Dummy Data using Mock Implementation.

***So by following Dependency Injection principle we can create interface-based loosely-coupled and easily testable services.***