Like Martin Fowler said in TwoHardThings:

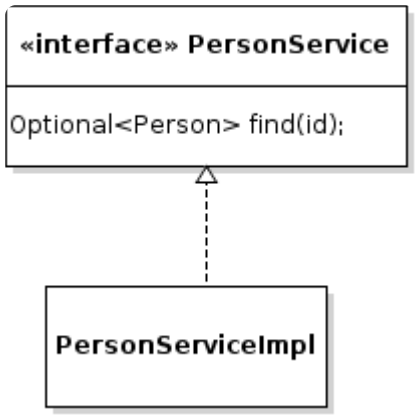
> There are only two hard things in Computer Science: cache invalidation and naming things.

I've just seen too many times developers using Dependency Injection frameworks like Spring or Guice or Dagger the wrong way. Naming classes with **Impl** suffix is an Anti-pattern and i'm going to explain why.

# Why Impl is Bad

## ServiceImpl is a common practice

Many developers, including myself years ago, are using the Interface + Impl pattern to create services which are injected by their interface. This usually looks like this:




*Interface and its associated implementation*

I think many developers simply don't understand the point of dependency injection because they are just doing Impl services. I strongly agree with this: there is no point of creating an interface for a service which has only a single known implementation. **Decoupling for the sake of decoupling has no sense**.

Many things are wrong with this:

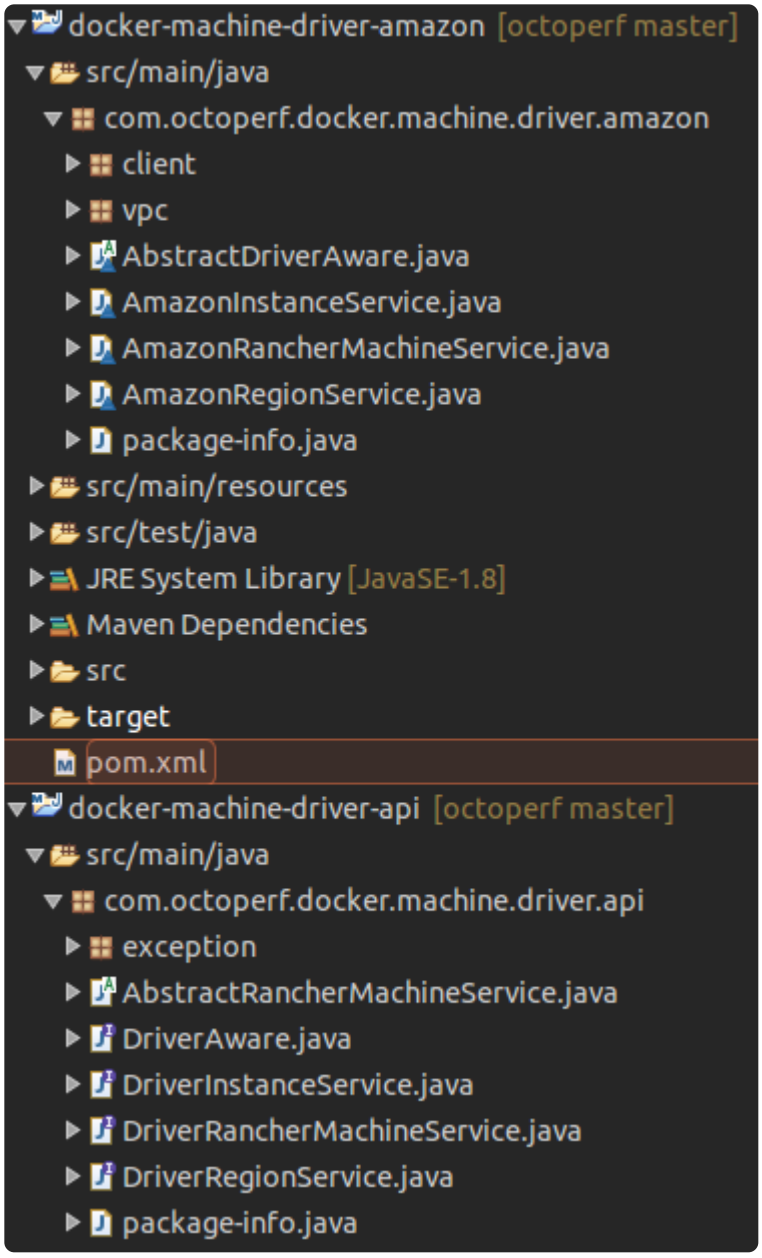
- How does the PersonServiceImpl actually implement the interface contract? Does it look for a Person in a database?
- If another implementation of the interface is needed, is it going to be **PersonServiceImpl2**?

You start to see what's wrong here.

## Impl is noise

In fact, naming an implementing class with the **Impl** suffix is a tautology. Every concrete class is of course an implementation of something. It's like naming the service interface **IPersonService**, the **I** prefix brings nothing more than noise to the name.

All modern Java IDE already point the different between implementations and interfaces in their UI.




*Eclipse Type indicators in the Package Explorer*

On the screenshot above, you can see that implementing classes inside the **docker-machine-driver-amazon** maven module are package protected. The implemented interfaces are located inside the **docker-machine-driver-api** maven module. We'll see later why interfaces and implementing classes must be separated from each other.

Good developers aim to be efficient. By putting unnecessary noise in class naming, the developer looses time by having to check what the implementation actually does. He needs to read the service implementation to know how it acts. This must be avoided at all cost to preserve the developers productivity.

## Impl is meaningless

Naming a class with the **Impl** suffix is like telling *I don't know how to name it*. If you can find a simple name for your implementing class, it's a sign of code smell. Things that can't be named properly are either too complex or doing too many things, thus usually breaking the Single Responsibility principle.

## Separation of Things

Most of the time, the interface and the implementation are inside the same package, or at least inside the same maven module. Sure, you have just one **ServiceImpl** class, why would you separate the interface from the implementation?

**It's useless if you tie the contract and the implementation together**. The purpose of an implementation is to be replaceable when it comes to dependency injection. By injecting interfaces instead of implementations, you allow the implementation to be replaced by another as the code doesn't depend on it.

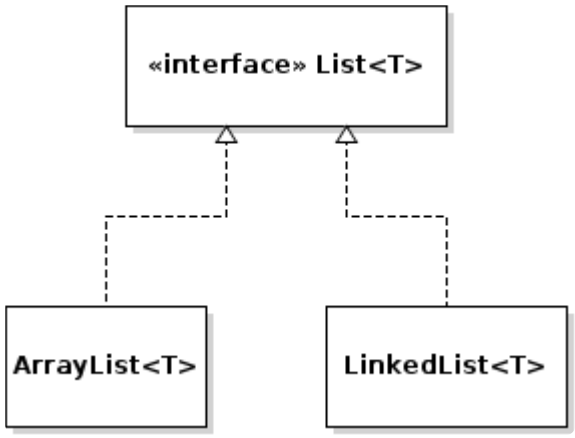
By shipping both the interface and the implementation tied together, **you will never be able to replace the implementation with another without editing the code**. That's what most of the developers don't understand, because they've just shipped interfaces with Impl all their life.

Shipping interface and its implementation together is like shipping rims with unreplaceable tires. Who wants this?

## The List Example

Let's consider one of the most common interface used by Java Developers: **List**. There are several implementations of the *List* interface:

- **ArrayList**: stores items inside an array,
- **LinkedList**: stores items by double-chaining the elements together,
- and more.



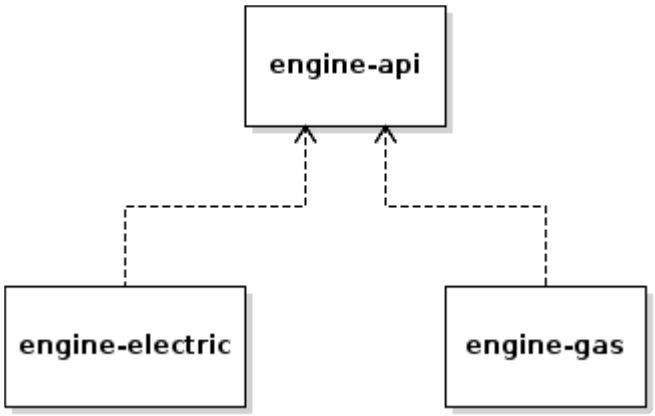

*Java List interface and a few implementations*

Have you ever seen a **ListImpl** inside the JDK? No, you don't. Why would you name your classes with the **Impl** suffix then?

# The API Pattern

This is where the API pattern comes to the rescue. We've seen that most developers just ship interfaces with their associated **Impl**together. The API pattern aims to separate the contract, namely the interface, and the implementation. This way, when you need to replace one implementation by

another inside your application, you just need to swap the dependency pointing to the right implementation. **No code change**.

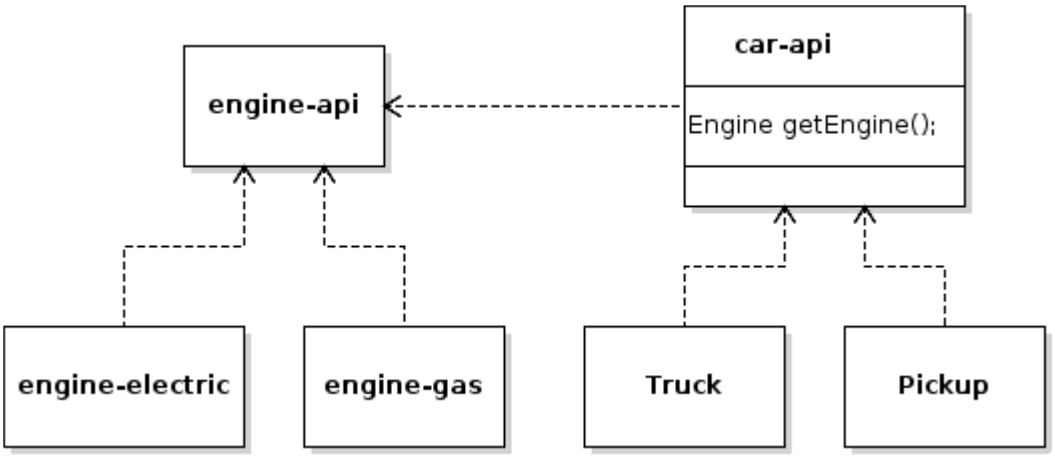



*Engine API Maven modules*

The screenshot above shows the Maven modules which have been created:

- **engine-api**: contains the interfaces defining the engine contract,
- **engine-electric**: implementation of an electric engine which depends on engine-api,
- **engine-gas**: implementation of a gas engine which depends on engine-api.

Any other maven module which depends on the engine only relies on the API.




*A car relies on a Engine, without an assumption on the engine type*

The beauty behind this pattern is that the car doesn't care about which engine has been put under the roof. Each engine provide the same *ignite* method to start them. **Implementing modules are only depending on API modules**.

The final application module is responsible for wiring all the things together by specifying the implementation to use:

```
<dependency>
  <groupId>com.octoperf</groupId>
  <artifactId>engine-electric</artifactId>
  <version>1.0.0</version>
</dependency>
```

If the electric engine needs to be replaced by a Gas engine, the dependency is simply replaced in the application dependency list. **The application module is the only one to depend on concrete implementations**.

## The API module

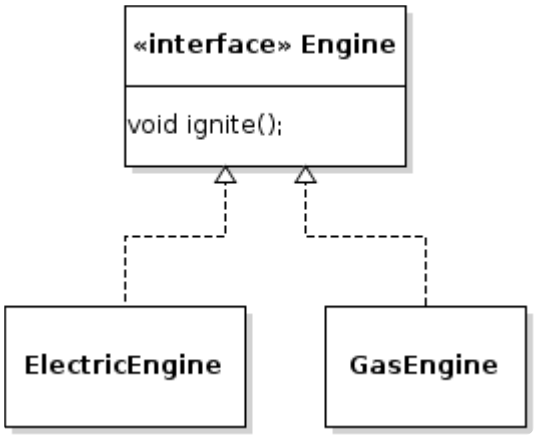
The API module provides the contract to implement, also known as interfaces in Java. It also provides the unit tests to verify that an implementation strictly follows the contract. It should never contain anything else. In this case, it's the **engine-api** module.

## The implementation modules

One implementation module must contain one type of implementation. This is why we have the **engine-electric** and **engine-gas**module. Those modules only contain implementing classes of a given contract. These modules should never contain any contract as they already ship implementations.

## Separation of concerns

I have created a full example application here: Design by contract. Suppose we want to model a *Car* which has an *Engine*. Nowadays, it's possible to order a Car with a **Petrol or an Electric engine**.




*Electric and Gas Engines implementing the Engine contract*

There is **no EngineImpl**. Would it be an electric or a Petrol engine? In this example, the implementations are quite obvious because the code relates to something physical. **ElectricEngine** and **GasEngine** have nothing in common, this is why each is in a separate maven module.

## Unit testability

Both engines can be unit tested independently. Even better, the classes depending on an *Engine* instance can be unit tested too without being forced to use any of the implementations above. There could even be a **TestEngine** inside a **engine-test** module whose purpose is to provide an engine for the tests.

**It often happens that your unit test may require some dependencies to be injected**. It can quickly become a nightmare to get your service properly injected and started if all the dependencies are following the **Impl** pattern. The whole services tree will be initialized and there are big chances it

will fail due to a missing resource like a database. This results in poor **test isolation**: the test is likely to fail when a dependency fails too. This is also known as cascading failure, and leads to hours of time wasted debugging.

On the other side, if all your implementing modules are depending only on API modules, then you can inject *mock* dependencies instead of the real services. All the services which need to be injected can be replaced by test classes, ensuring proper test isolation. If the test fails, you can be sure it's because your service fails and nothing else. No time is wasted debugging obscure dependencies.

```
@RunWith(MockitoJUnitRunner.class)
public abstract class AbstractClusterServiceTest {

  @Test
  public void shouldBeLeader() {
    final ClusterService service = newService();
    final boolean isLeader = service.isLeader();
    assertTrue(isLeader);
  }

  protected abstract ClusterService newService();
}
```

*Example of an API providing a unit test*

And the implementing service which simply relies on the contract unit test and provides the implementation:

```
public class JVMClusterServiceTest extends AbstractClusterServiceTest {
  @Override
  protected ClusterService newService() {
    return new JVMClusterService();
  }
}
```

*JVM wide clustering service being tested*

## Parallelizing developments

One of the greatest benefits of decoupling contracts (interfaces) from implementations (concrete classes) **is the increased productivity**:

- The API is being designed with both the client team and the API team,
- The client team can start working on the client code which uses the API once the contract has been defined,
- While the client team works on the client code, the API teams works on desired implementations.

This provides a highly parallelizable pattern which greatly helps **to scale a development team**. It reduces the wait time between the teams as everyone can get to work even if each team hasn't finished their part yet. The only bottleneck is the API contract definition. One could even imagine delegating part of the developments by outsourcing the implementation of the required contracts. External developers can be in charge of providing an implementation to the exposed contract.

# Conclusion

The goal of eliminating bad habits and practices in software development is to increase the development productivity and speed. Sure, we as good developers, like to work with pristine code. Why? Because it puts much less stress on our mind to work with well structured code. **We can focus on shipping rather than messing around with bad code and wasting time debugging.**