FACTORY

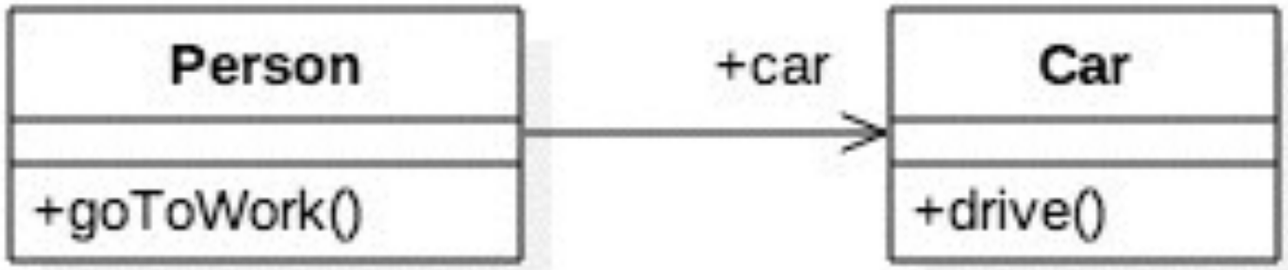
Krishna Mohan Koyya

krishna@glarimy.com | www.glarimy.com

FACTORY

- **This is a Creational Pattern.**
- **It deals with the issue of who should create an object.**
- **Not to be confused with GoF Factory Method Pattern.**

FACTORY – PROBLEM ILLUSTRATION



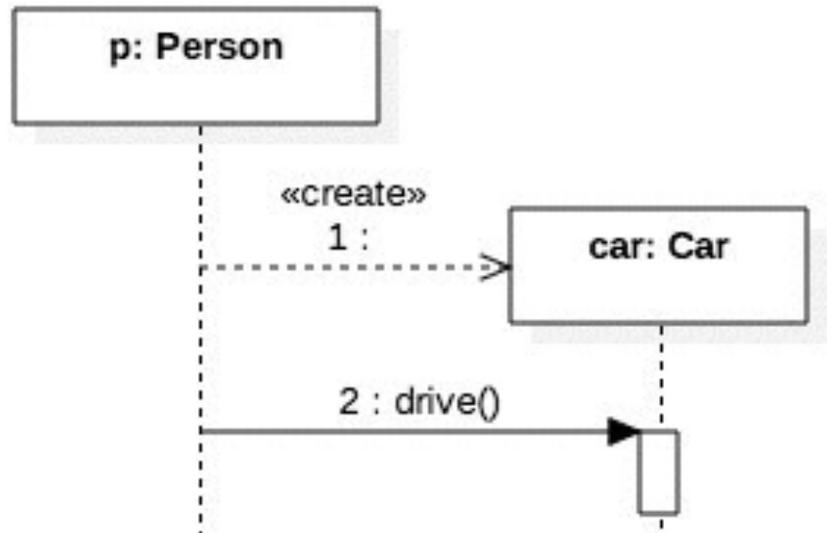
- This diagram depicts a relation between Person and Car.
- The Person uses the Car.
  - The Person is the client and the Car is the product.
- How the Person gets the Car?

FACTORY – PROBLEM ILLUSTRATION

- **One of the most common practice:**

```
public class Person{
    Car car = new Car();

    public void goToWork(){
        car.drive();
    }
```

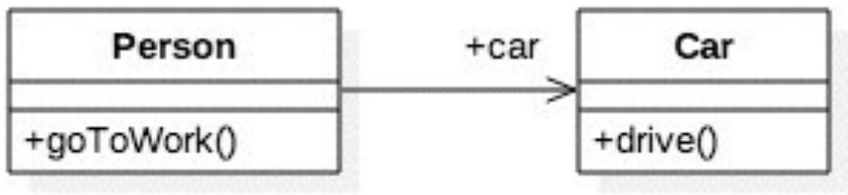


- **Person creates the car in order for using it.**
- **By using the new operator.**

FACTORY – PROBLEM ILLUSTRATION

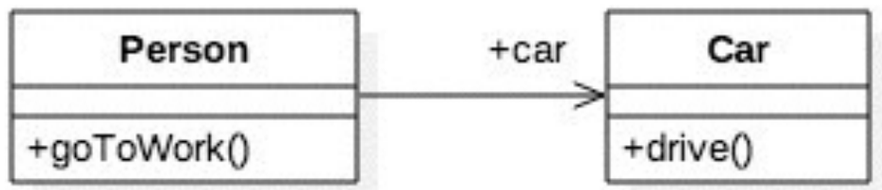
- **This design delivers the expected functionality**
- **However, it suffers from the non-functional point of view**
  - Because of the tight coupling between Person and Car
- **Tight coupling leads to several issues like**
  - Testability
  - Productivity
  - Flexibility
  - Maintainability
  - Extendability
  - And etc.,

FACTORY – PROBLEM ILLUSTRATION



- **This solution can not be unit tested**
  - The Person class can't be separated from Car class
    - And hence Person can't be tested at the unit level.
      - The overall quality of the application suffers.
- **This design impacts the productivity**
  - The Person class can't be coded
    - Unless the class of Car is made available.
    - Means, parallel development is not possible.

FACTORY – PROBLEM ILLUSTRATION

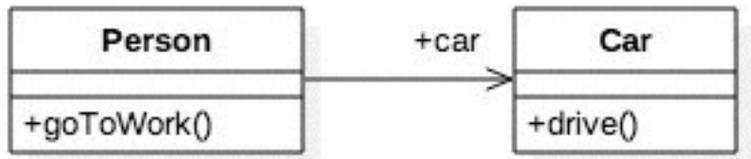


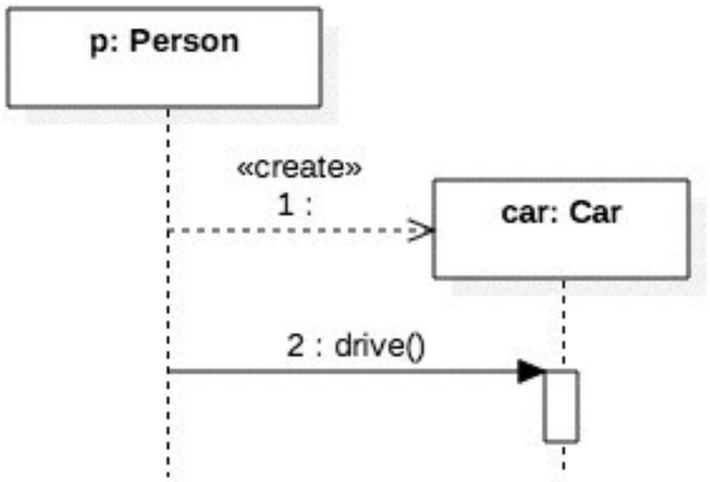
- **This solution is not flexible**
  - With this arrangement, the Person can't drive anything other than Car.
  - If the requirement is changed (after the coding) to use a Jeep
  - The Person class needs to be re-written, re-tested and re-distributed

```
public class Person{
      Jeep jeep = new Jeep();
      public void goToWork(){
          jeep.drive();
          …
        }
}
```

FACTORY – PROBLEM DEFINITION

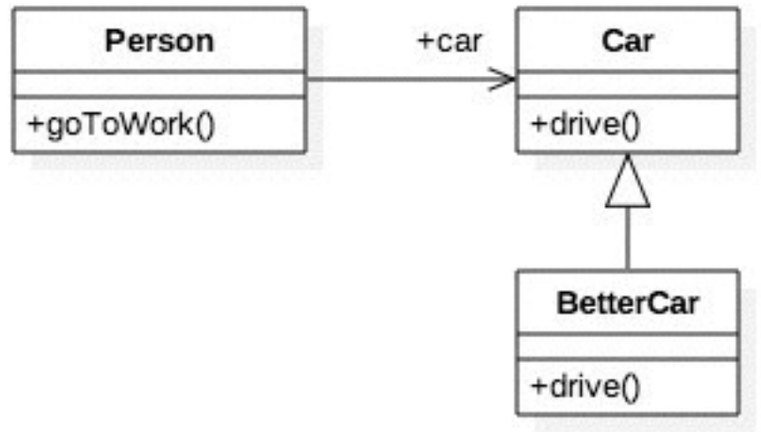
- **Maintaining the code becomes very difficult**
  - If different customers want support for different vehicles,
    - We have to maintain different versions
      - Of the Person class in the source code repo.
  - One version will have Person coupled with Car
  - Whereas the other version will have Person coupled Jeep
  - This leads to exponential costs in terms of building, testing, documenting and distributing





FACTORY – PROBLEM DEFINITION

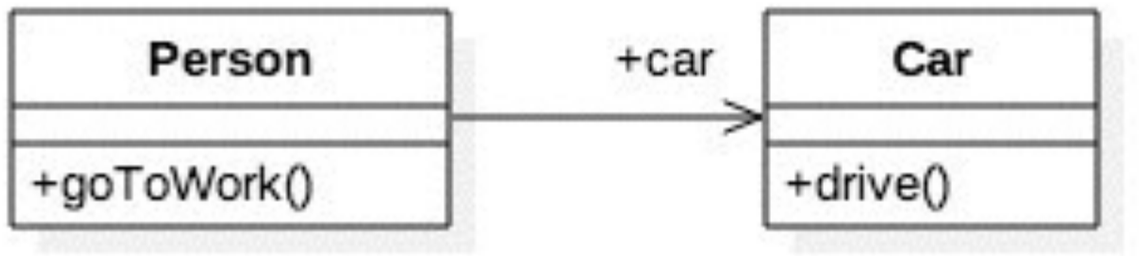
- **This design makes it difficult to extend it further**
- If 3rd party developer wants the Person to use an extended Car
  - it is simply impossible without changing the Person class.




```
public class Person{
    Car car = new BetterCar();

    public void goToWork(){
          car.drive();
    }
}
```

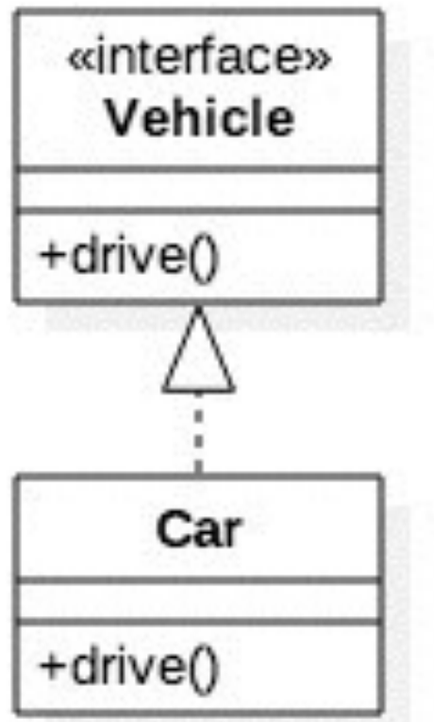
FACTORY – PROBLEM ILLUSTRATION



- The fundamental problem
  - The Person is tightly coupled with the Car.
  - **The Client is tightly coupled with the Provider**
  - **The Client is creating the Provider**

FACTORY – EVOLVING A SOLUTION

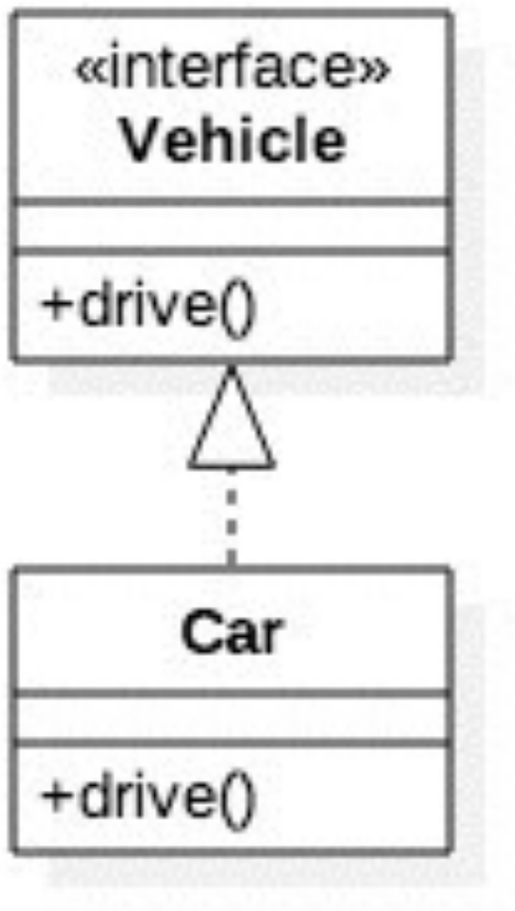
- The factory pattern gives us a way to break this coupling.
- Let's go one step at a time.
- **Step 1: Build an interface to the provider**
  - Car is the provider class
- Let's introduce an interface called Vehicle
  - Which the Car implements.
- In fact, all business classes are better to have interfaces



FACTORY – EVOLVING A SOLUTION

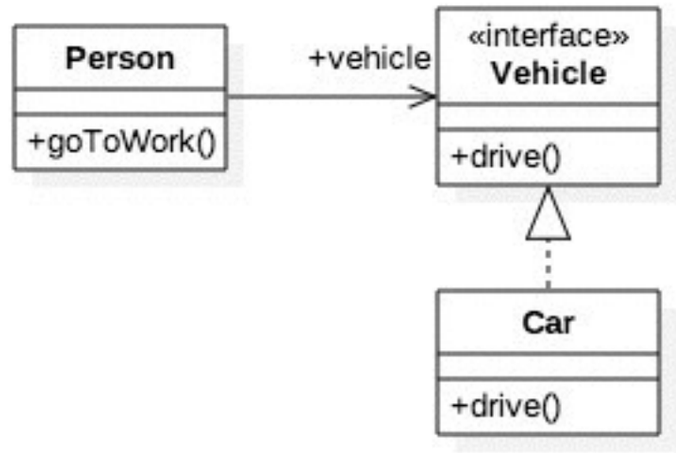
```
public interface Vehicle{
  public void drive();
}

public class Car implements Vehicle{
  public void drive(){
     ...
  }
}
```



FACTORY – SOLUTION

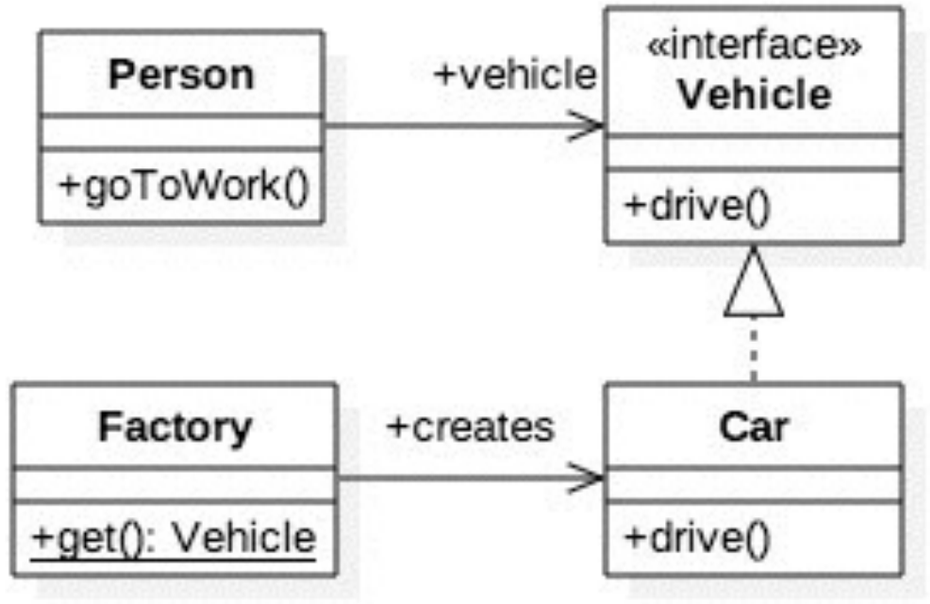
- **Step 2: Code against Interface**
  - Let the Client access the Provider through its interface
    - Let the Person access the Car through Vehicle interface.




```
public class Person{
      Vehicle vehicle = new
      Car();

      public void goToWork(){
            vehicle.drive();
            ...
       }
}
```

FACTORY – SOLUTION



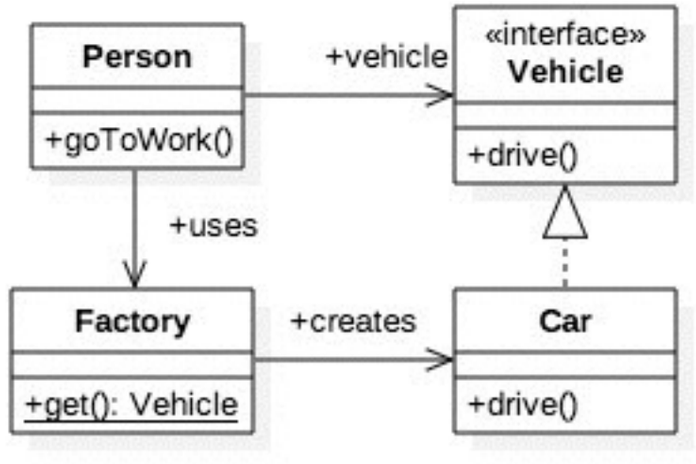
- **Step 3: Introduce a Factory to create Provider**

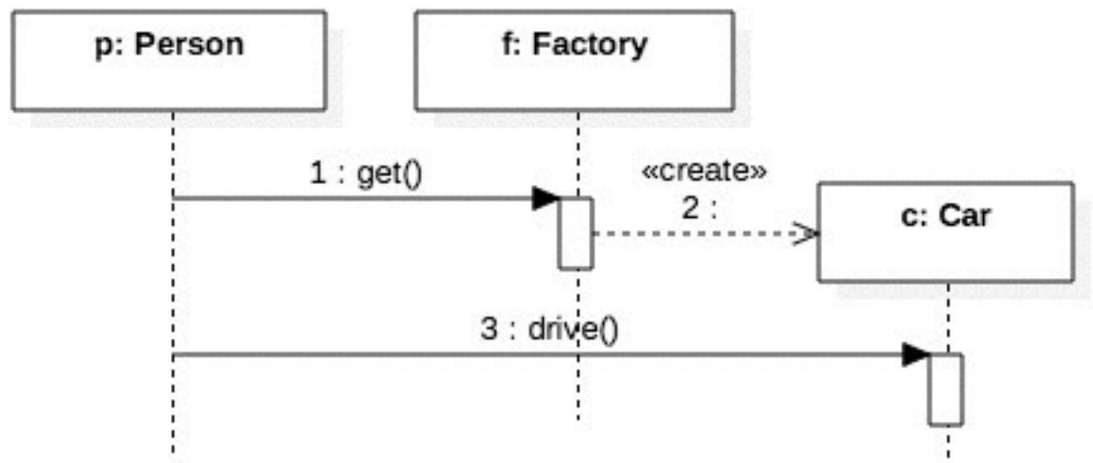
```
public class Factory{
   public static Vehicle getVehicle(){
      return new Car();
   }
}
```

FACTORY – SOLUTION

- **Step 4: Let Client use the Factory for getting Provider**
- Let the Person use the Factory to get a Car.

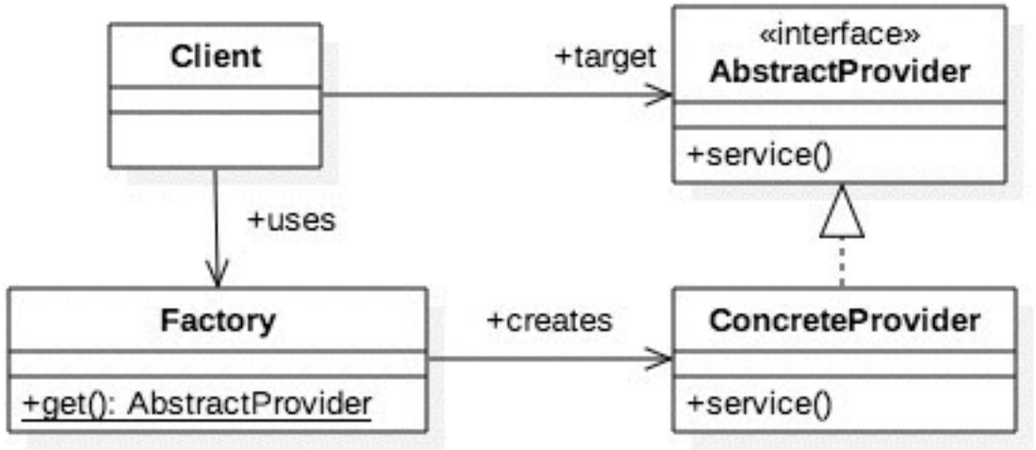
```
public class Person{
  Vehicle vehicle =
Factory.get();
  public void goToWork(){
    vehicle.drive();
    ...
  }
}
```

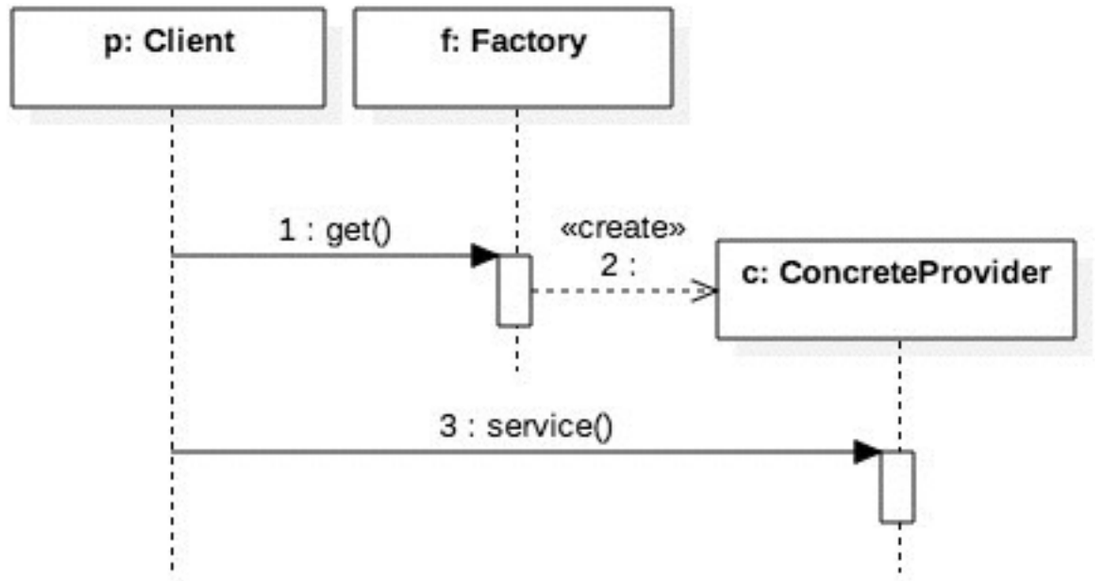




- **This arrangement sums up the Factory pattern.**

FACTORY – THE PATTERN





- Intent: Separating the responsibility of object creation from the user of the object.
- Players: Abstract Provider, Concrete Provider, Factory, Client
- When to use: Whenever a client needs a service provider, use the factory to get one.

FACTORY – ILLUSTRATION

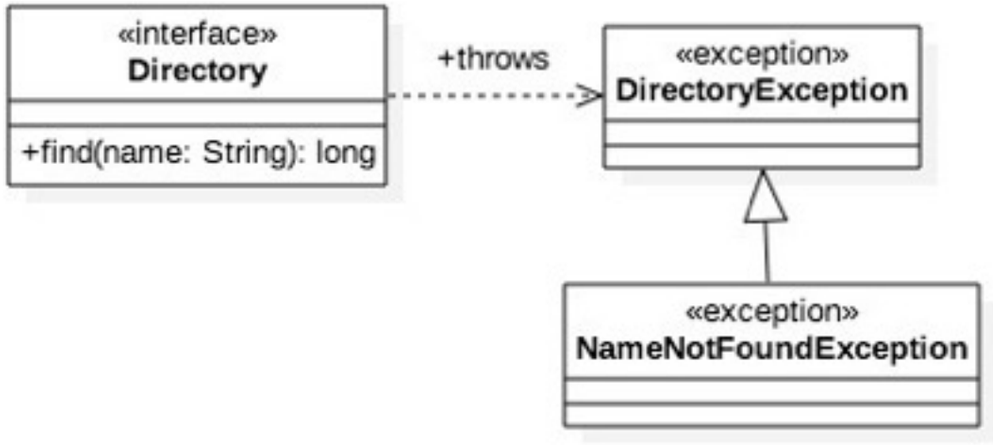
Case Study

We would like to develop a directory component using which the clients can find the phone number of a person by supplying the name of the person.

FACTORY – IMPLEMENTATION 1

Interface and Exceptions

This prototype is having only one business method and a common exception class.




```
public interface Directory {
   public long find(String name) throws
NameNotFoundException, DirectoryException;
}

public class DirectoryException extends
RuntimeException {
}

public class NameNotFoundException extends
DirectoryException {
}
```

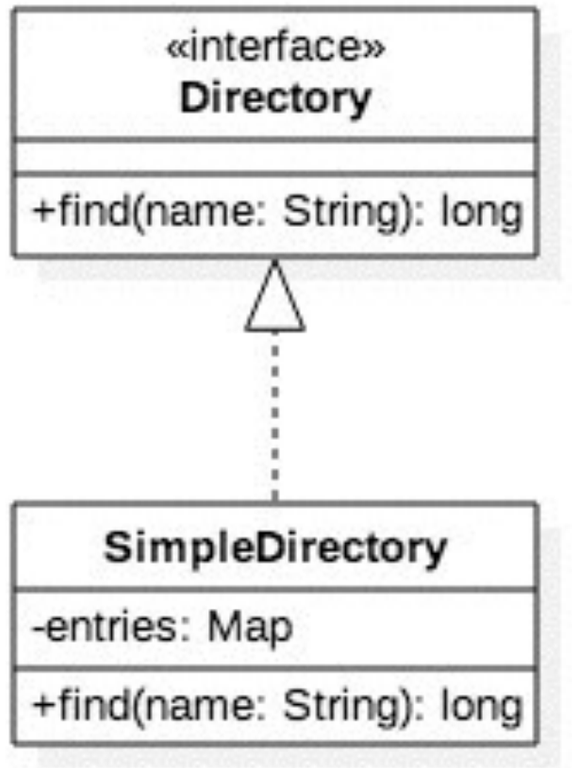
FACTORY – IMPLEMENTATION 1

Possible Business Implementation

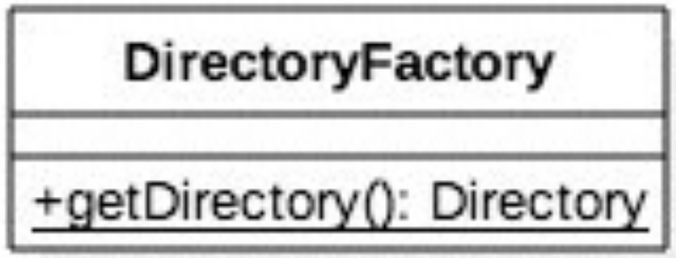
```
public class SimpleDirectory implements Directory {
    private Map<String, Long> entries;

    public SimpleDirectory() {
        entries = new HashMap<>();
        entries.put("Krishna", 9731423166L);
    }

    public long find(String name) {
        Long phoneNumber = entries.get(name);
        if (phoneNumber == null)
            throw new NameNotFoundException();
        return phoneNumber;
    }
}
```



FACTORY – IMPLEMENTATION 1

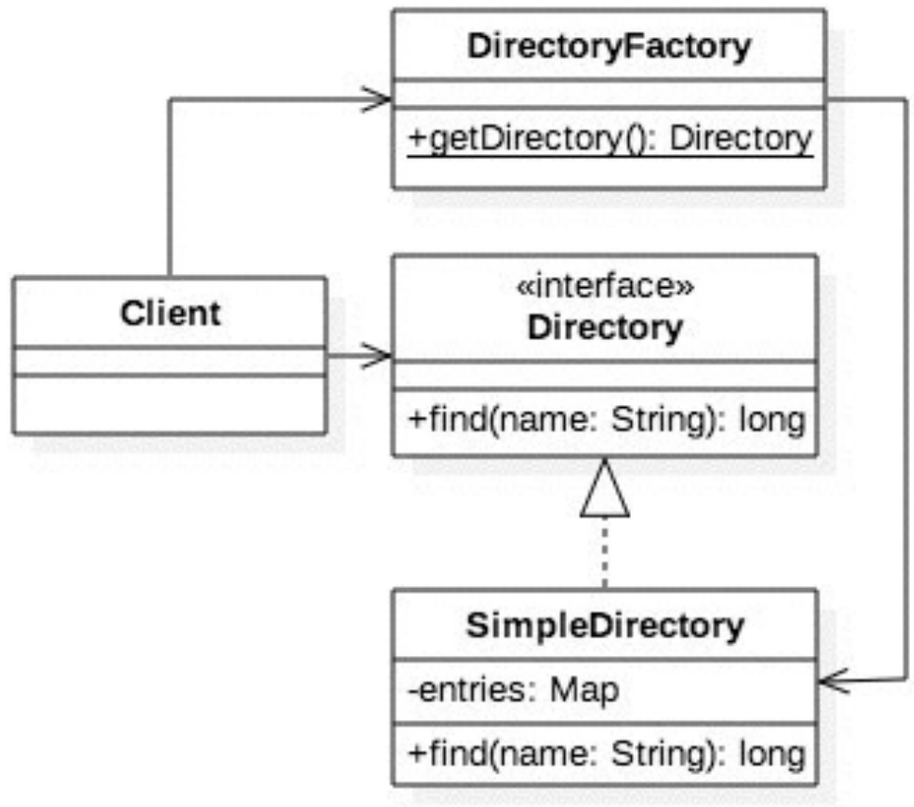


➤ This is the simplest possible implementation of DirectoryFactory.

```
public class DirectoryFactory {
   public static Directory getDirectory() throws Exception {
      return new SimpleDirectory();
   }
}
```

FACTORY – IMPLEMENTATION 1

- Though there is not much use in this factory as it always returns a new instance of the same SimpleDirectory, it is still a best practice to delegate this job to the DirectoryFactory.




```
public class DirectoryClient {
   public static void main(String[] args) throws Exception {
      Directory dir = DirectoryFactory.getDirectory();
      long phoneNumber = dir.find("Krishna");
      System.out.println(phoneNumber);
   }
}
```

FACTORY – IMPLEMENTATION 2

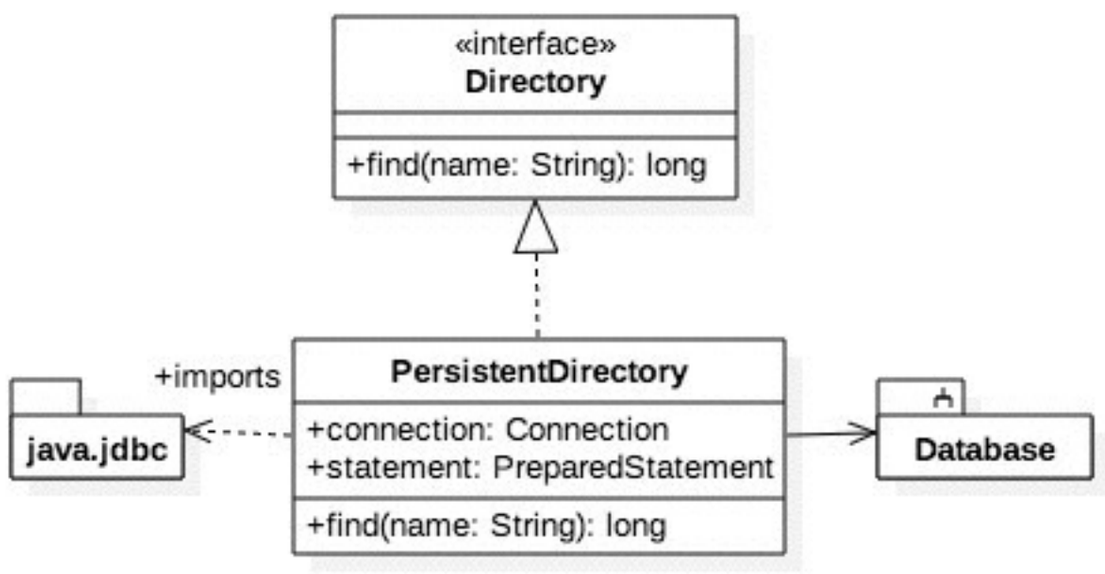
Another Possible Business Implementation of Directory

```
public class PersistentDirectory implements Directory {
	private Connection connection;
	private PreparedStatement statement;

	public PersistentDirectory() throws DirectoryException {
		try {
			Class.forName("com.mysql.jdbc.Driver");
			connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/glarimy?user=root&password=admin");
			statement = connection.prepareStatement("select * from directory where name=?");
		} catch (Exception e) {
			throw new DirectoryException(e);
		}
	}

	public long find(String name) throws DirectoryException {
		try {
			statement.setString(1, name);
			ResultSet rs = null;
			try {
				rs = statement.executeQuery();
				if (rs.next())
					return rs.getLong("phonenumber");
				else
					throw new DirectoryException("No contact found!");
			} finally {
				rs.close();
			}
		} catch (Exception e) {
			throw new DirectoryException(e);
		}
	}

	public void finalize() {
		try {
			statement.close();
			connection.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

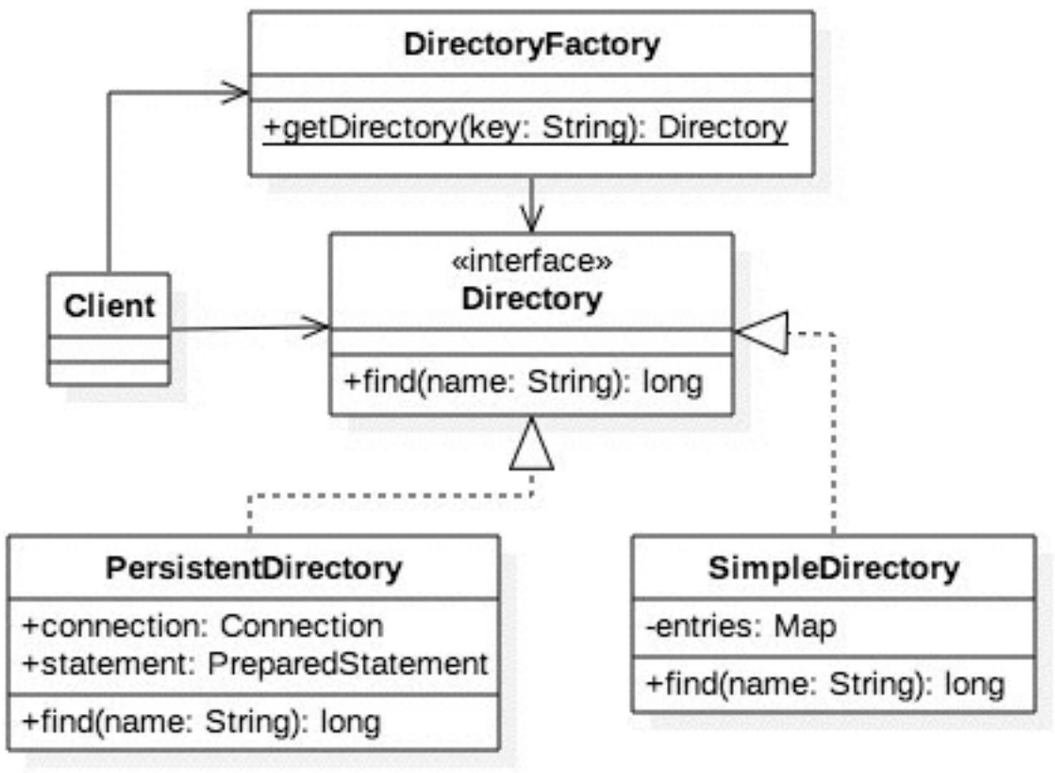


FACTORY – IMPLEMENTATION 2

- This DirectoryFactory that supplies an implementation based on the client requirement.
- Again hard coding if...else conditions is not a great practice, but still much better than the above implementation.
- Whenever a new alternative is available, only the DirectoryFactory undergoes the change, not the rest of the system.

```
public class DirectoryFactory {
    public static Directory getDirectory(String req) throws Exception {
        if (req.equalsIgnoreCase("memory"))
            return new SimpleDirectory();
        else if (req.equalsIgnoreCase("db"))
            return new PersistentDirectory();
        else
            throw new Exception("Unknown Directory");
    }
}

public class DirectoryClient {
    public static void main(String[] args) throws Exception {
        Directory dir = DirectoryFactory.getDirectory("db");
        long phoneNumber = dir.find("Krishna");
        System.out.println(phoneNumber);
    }
}
```



FACTORY – IMPLEMENTATION 3

- This DirectoryFactory selects implementation based on configuration and uses Java Reflection to load the class and instantiates it.
- Though there is a possibility of runtime issues (in case the class is not available for loading and etc.,) it is still better than the earlier implementation.
- No code needs to be changed in order to support any new implementations of the Directory interface.

directory.properties

```
directory=com.glarimy.factory.SimpleDirectory
```

```
public class DirectoryFactory {
  public static Directory getDirectory() throws Exception{
      Properties props = new Properties();
      props.load(new FileReader("directory.properties"));
      String name = props.getProperty("directory");
      return (Directory) Class.forName(name).newInstance();
  }
}

public class DirectoryClient {
  public static void main(String[] args) throws Exception {
      Directory dir = DirectoryFactory.getDirectory();
      long phoneNumber = dir.find("Krishna");
      System.out.println(phoneNumber);
  }
}
```

FACTORY – IMPLEMENTATION 4

This CommonFactory supplies any of the interfaces from configuration.

Hence, there is no need of developing different factories for different implementations.

The applications defines various keys to pickup various components.

factory.properties

```
directory=com.glarimy.factory.SimpleDirectory
```

```
public class CommonFactory {
    public static Object get(String key) throws Exception {
        Properties props = new Properties();
        props.load(new FileReader("factory.properties"));
        String name = props.getProperty(key);
        return Class.forName(name).newInstance();
    }
}

public class DirectoryClient {
    public static void main(String[] args) throws Exception {
        Directory dir = (Directory) CommonFactory.get("directory");
        long phoneNumber = dir.find("krishna");
        System.out.println(phoneNumber);
    }
}
```

FACTORY – IMPLEMENTATION 5

This CommonFactory is capable of loading even the Singletons as well.

As a singleton does not have a public constructor, there should be a mechanism to identify the static method through which the object can be obtained.

This implementation goes by convention and assumes that the singleton have a method by name getInstance().

factory.properties

```
directory=com.glarimy.factory.SimpleDirectory
```

```
public class CommonFactory {
    @SuppressWarnings("unchecked")
    public static Object get(String key) throws Exception {
        Properties props = new Properties();
        props.load(new FileReader("factory.properties"));
        String name = props.getProperty(key);
        @SuppressWarnings("rawtypes")
        Class claz = Class.forName(name);

        try {
            return claz.newInstance();
        } catch (Exception e) {
            return claz.getMethod("getInstance").invoke(claz);
        }
    }

}
```

FACTORY – IMPLEMENTATION 5

```
public class SimpleDirectory implements Directory {
    private Map<String, Long> entries;
    private static SimpleDirectory instance;

    public static SimpleDirectory getInstance() {
        if (instance == null)
            instance = new SimpleDirectory();
        return instance;
    }

    private SimpleDirectory() {
        entries = new HashMap<>();
        entries.put("Krishna", 9731423166L);
    }

    public long find(String name) {
        Long phoneNumber = entries.get(name);
        if (phoneNumber == null)
            throw new NameNotFoundException();
        return phoneNumber;
    }
}
```

FACTORY – IMPLEMENTATION 6

This CommonFactory is similar to the previous one.

However, instead of forcing the singleton developers to have the method named getInstance(), it identifies the appropriate method by using annotations.

factory.properties

```
directory=com.glarimy.factory.SimpleDirectory
```

Annotations

```
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface Singleton {
}

@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface FactoryMethod {

}
```

FACTORY – IMPLEMENTATION 6

```
public class CommonFactory {
    @SuppressWarnings("unchecked")
    public static Object get(String key) throws Exception {
        Properties props = new Properties();
        props.load(new FileReader("factory.properties"));
        String name = props.getProperty(key);
        @SuppressWarnings("rawtypes")
        Class claz = Class.forName(name);

        try {
            return claz.newInstance();
        } catch (Exception e) {
            if (claz.getAnnotation(Singleton.class) == null)
                throw new Exception();
            Method[] methods = claz.getDeclaredMethods();
            for (Method method : methods) {
                if (method.getAnnotation(FactoryMethod.class) != null)
                    return method.invoke(claz);
            }
            throw new Exception();
        }
    }
}
```

FACTORY – FRAMEWORK EXAMPLE

- Spring Framework provides us with ApplicationContext which does lot besides acting as a Factory.
- It loads the interface implementations based on an xml configuration.

beans.xml

```
<beans>
   <bean name="directory" class="com.glarimy.SimpleDirectory"/>
</beans>
```

Spring Application

```
public class DirectoryClient {
   public static void main(String[] args){
      ApplicationContext ctx;
      ctx = new ClasspathXmlApplicationContext("beans.xml");
      Directory dir = (Directory) ctx.getBean("directory");
      Contact contact = dir.find("krishna");
      System.out.println(contact.getPhoneNumber);
   }
}
```

FACTORY – RESOURCES

- GIT Source Code
  - /glarimy
- YouTube Channel
  - sversity-glarimy
- Website
  - www.glarimy.com
  - http://sversity.glarimy.com
- Facebook
  - /glarimy