The Hollywood principle: "don't call us, we call you!"

A parade of software patterns

Def.: Component and services

Component: a glob of software intended to be used, without change, by an application that is out of the control of the writers of the component.

'without change': the using application doesn't change the source code of the components, although they may alter the component's behavior by extending it in ways allowed by the component writers.

Service: a piece of software similar to a component in that it's used by foreign applications.

- a component to be used locally (e.g. jar file, assembly, dll, or a source import).
- a service will be used remotely through some remote interface, either synchronous or asynchronous (e.g. web service, messaging system, RPC, or socket.)

from: http://martinfowler.com/articles/injection.html

Implicit invokation

- Instead of invoking a procedure directly, a component announces (broadcasts) an event.
- Other components register their interest in a given event, and associate a procedure with the event.
- When the event is announced the system itself invokes all of the procedures that have been registered for the event.

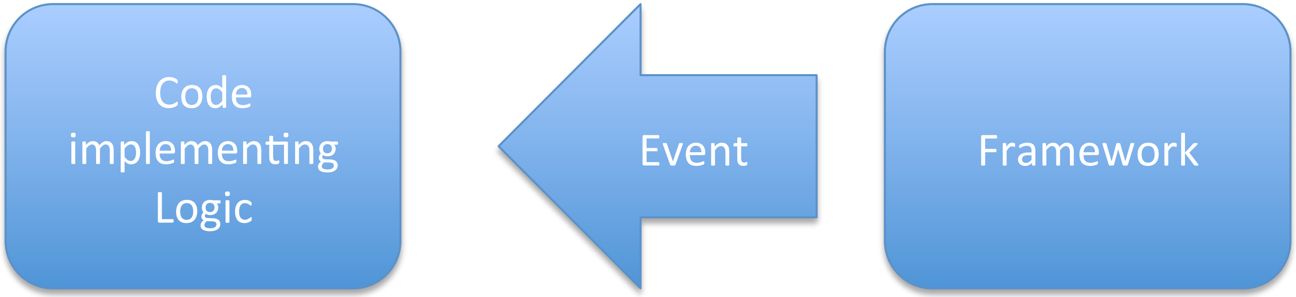
==> an event announcement implicitly causes the invocation of procedures in other modules.

A form of Inversion of Control

Adapted from David Garlan and Mary Shaw, "An Introduction to Software Architecture"

JDK Examples:

- The Swing GUI-event model
- SAX
- The Applet Framework
- The Servlet Framework



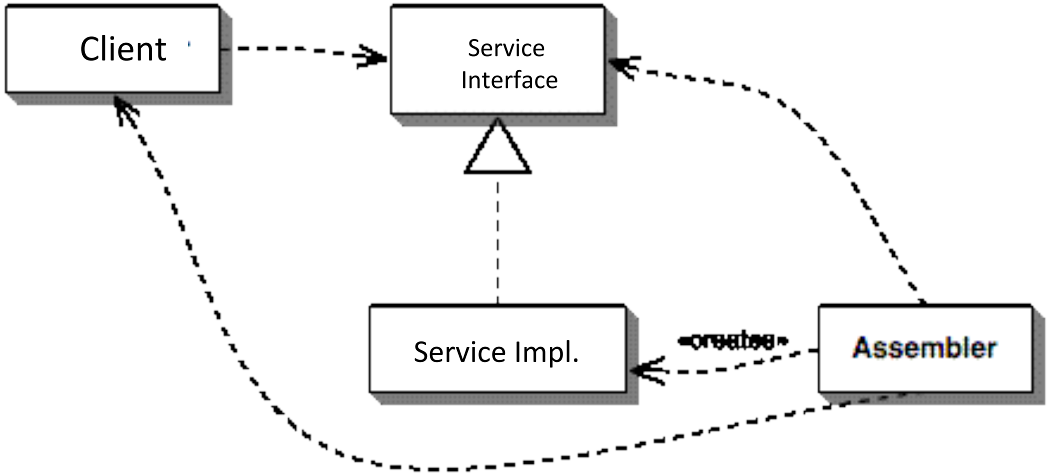
Code implementing Logic

Event

Framework

Inversion of Control (IoC)

A programming technique in which
object coupling is bound at run time
by an assembler object
and is typically not known at compile time



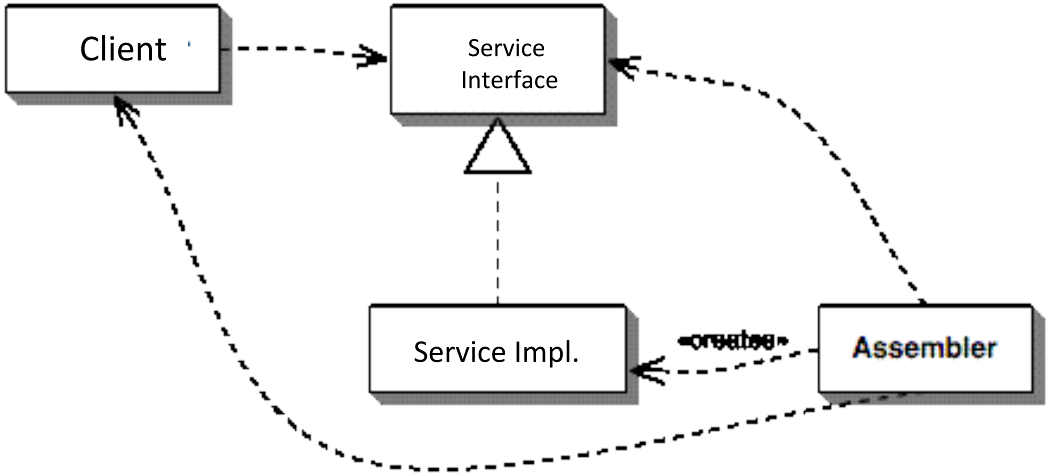
Inversion of control: some techniques

- factory pattern
- service locator pattern
- using a dependency injection, for example:
  - a constructor injection
  - parameter injection
  - a setter injection
  - an interface injection

see http://martinfowler.com/articles/injection.html

Factory

Factories are used to encapsulate instantiation.

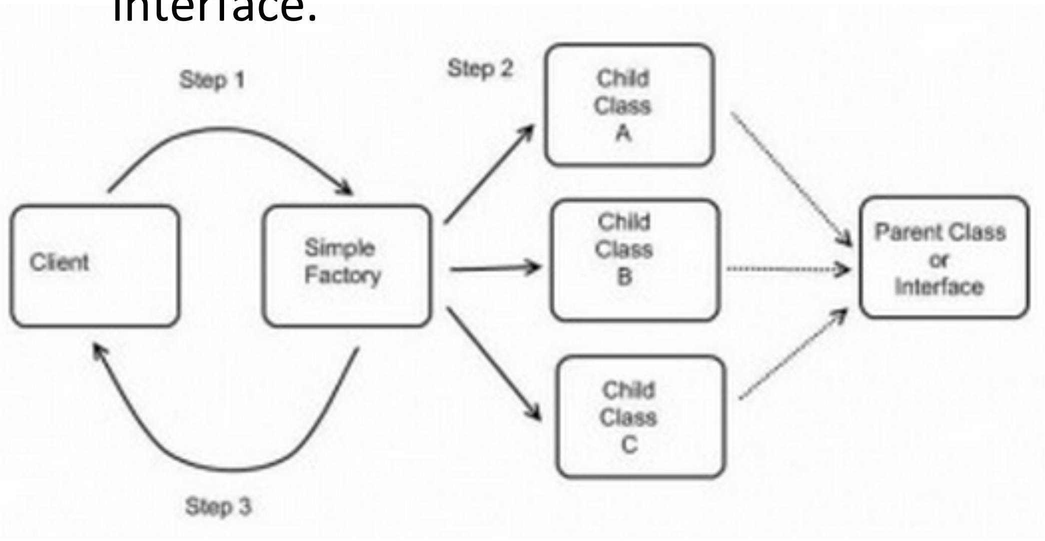


using a Simple Factory

1) you call a (possibly static) method in the factory. The call parameters tell the factory which class to create.

2) the factory creates your object. All the objects it can create either have the same parent class, or implement the same interface.

3) factory returns the object, the client expect is it to match the parent class / interface.



```
Parent x=Factory.create(p);


class Factory{
        static Parent create(Param p) {
                if (p…) return new ChildA();
                 else return new ChildB();
        }
}
```

Singleton

- Ensure a class has only one instance and provide a global point of access to it.

```
class Referee{
    static Referee instance= null;
    private Referee() {
      String s = "";
    }
    public static Referee getReferee() {
      if (instance ==null) instance=new Referee();
      return instance;
    }
    public void whistle() {
      //...
    }
}
```

Singleton usage

```
package myPackage;

public class Game{
      public static void main(String a[]) {
    new Game ();
  }

  Game () {
    //Referee a=new Referee (); // would give an error!
    Referee b=Referee.getReferee();
    Referee c=Referee.getReferee();
    System.out.println(b==c);
  }
}
```

Example

```
SAXParserFactory factory = SAXParserFactory.newInstance();  // singleton
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();  // simple factory
```

Factory method

1) the client maintains a reference to the abstract Creator, but instantiates it with one of the subclasses. (i.e. Creator c = new ConcreteCreator1();)

2) the Creator has an abstract method for creation of an object, which we'll call "Create". It's an abstract method which all child classes must implement. This abstract method also stipulates that the type that will be returned is the Parent Class or the Interface of the “product”.

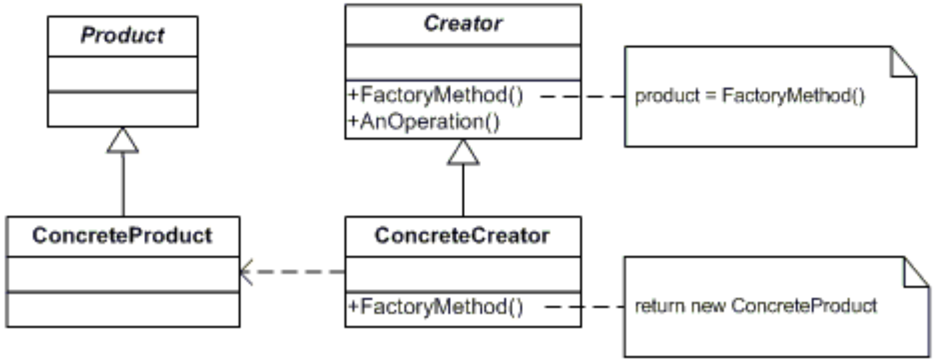
3) the concrete creator creates the concrete object. In the case of Step One, this would be "Child Class A".

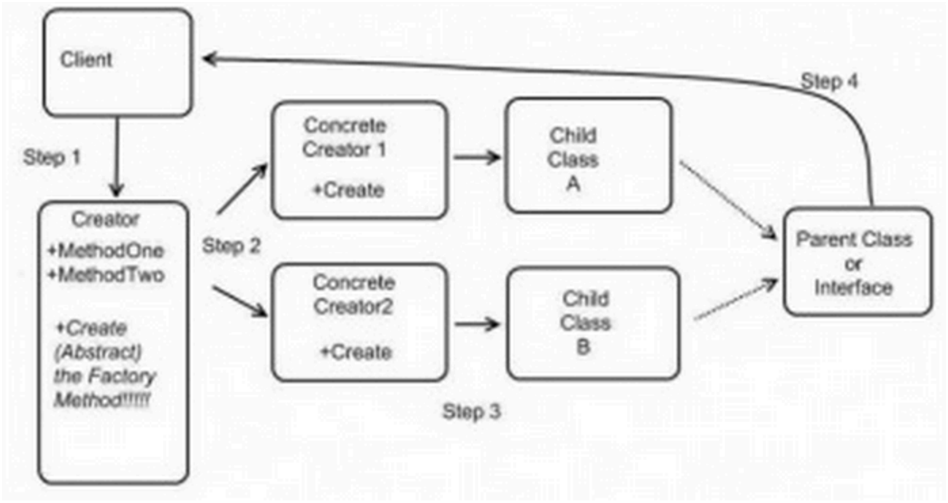
4) the concrete object is returned to the client. The client doesn’t really know what the type of the object is, just that it is a child of the parent.

Factory method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.





Factory method - example

The products:

```
abstract class Document{…}
class Report extends Document{…}
class Resume extends Document{…}
```

The factories:

```
abstract class DocumentCreator{
        abstract Document create();
}
class ReportCreator extends DocumentCreator {
        Document create() return new Report();
}
class ResumeCreator extends DocumentCreator {
        Document create() return new Resume();
}
```

The client:

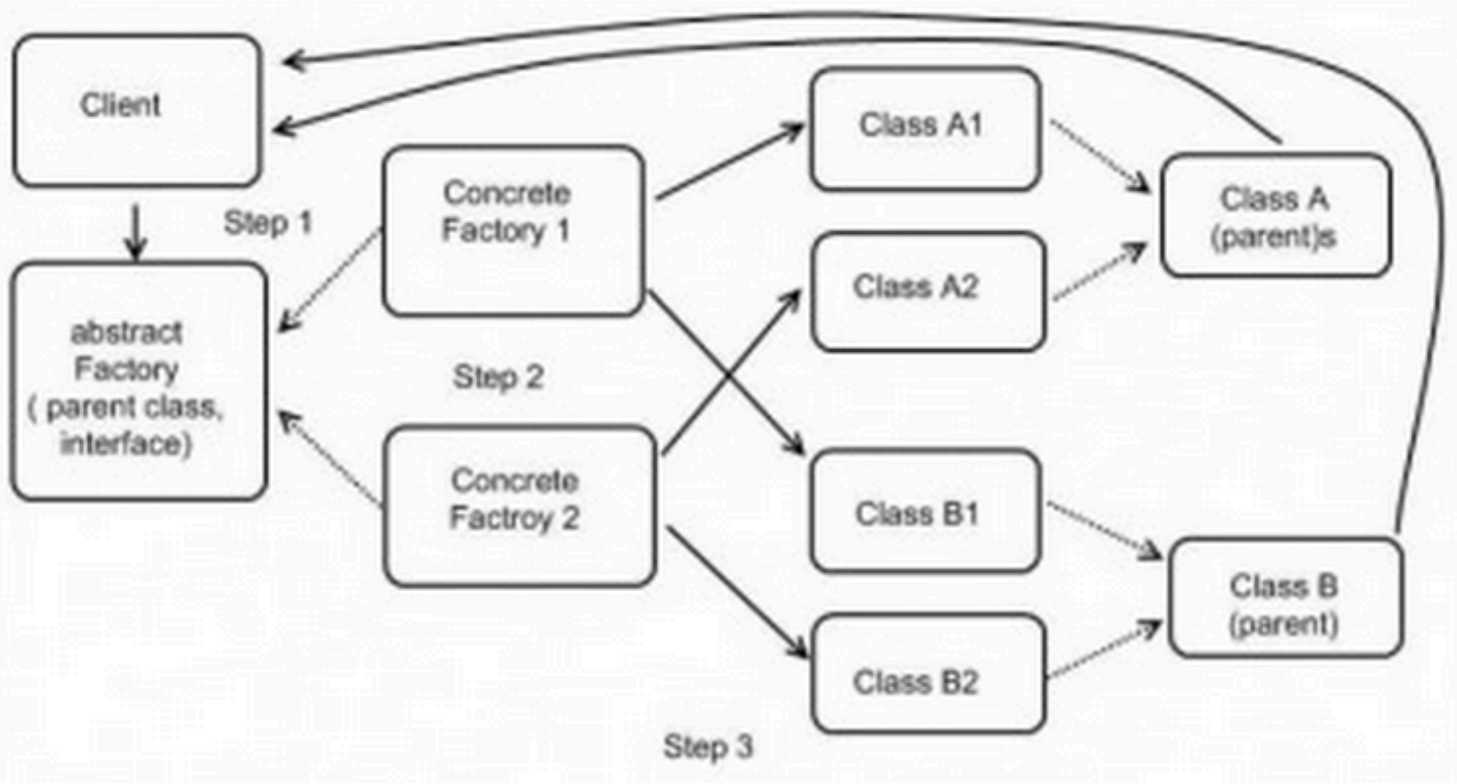
```
Document x=null;
String choice=JOptionPane.showInputDialog("Choose Report (1) or Resume (2)", null);
if (choice.equals("1") x=ReportCreator.create();
if (choice.equals("2") x=DocumentCreator.create();
```

AbstractFactory

Provide an interface for creating

families of related or dependent objects

without specifying their concrete classes.



AbstractFactory

The big difference is that by its own definition,

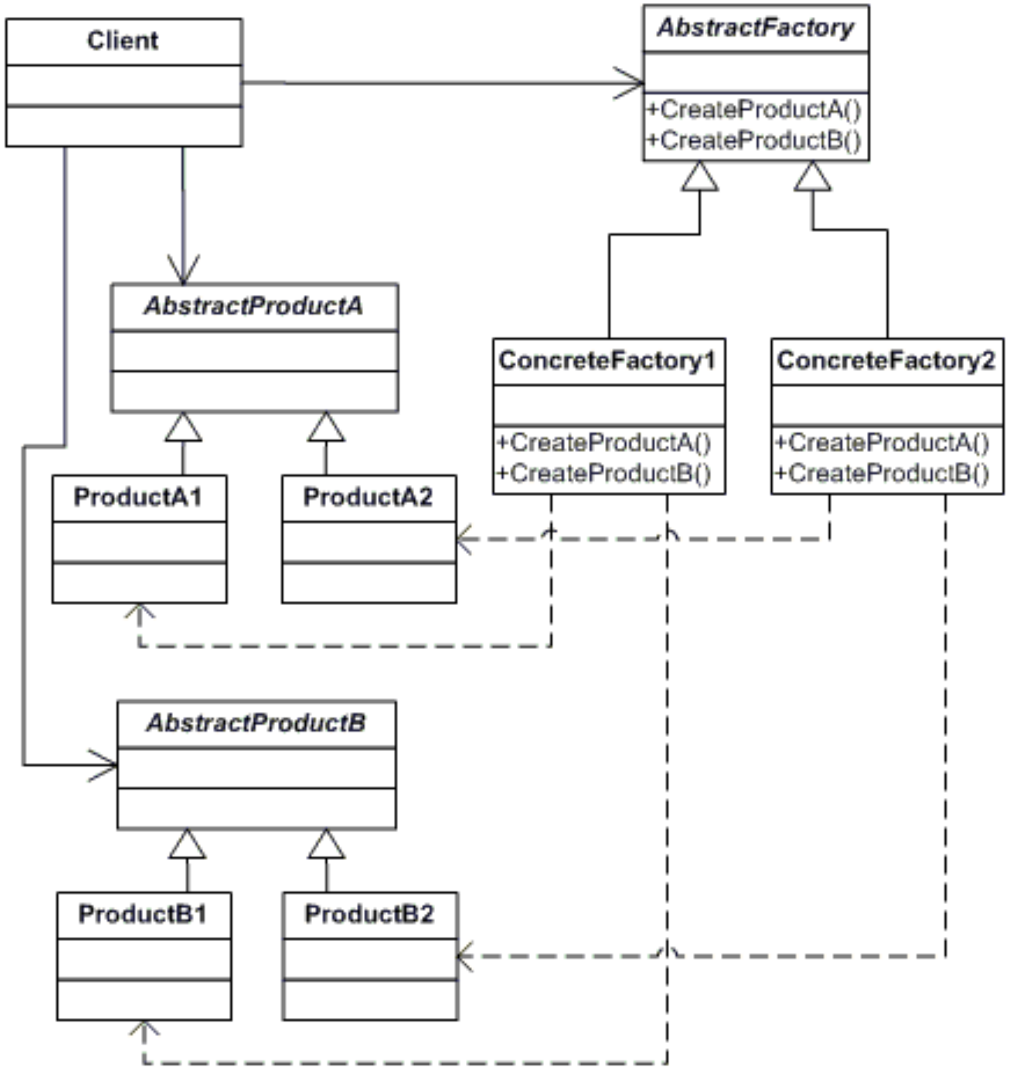
an Abstract Factory

is used to create a family

of related products,

while Factory Method

creates one product.



Summary of Factory types

- A Simple Factory is normally called by the client via a static method, and returns one of several objects that all inherit/implement the same parent.
- The Factory Method design is really all about a “create” method that is implemented by sub classes.
- Abstract Factory design is about returning a family of related objects to the client. It normally uses the Factory Method to create the objects.

Service Locator

Have an object that knows how to get hold of all of the services that an application might need.

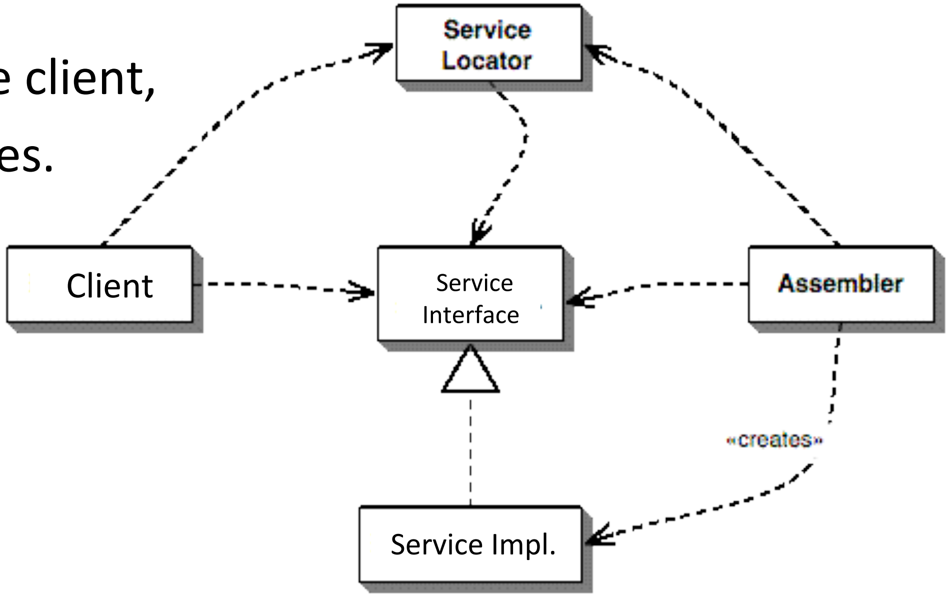
A service locator has a method that, given a key value, returns the implementation of a service when one is needed.

Of course this just shifts the burden:

we still have to get the locator into the client,

but this scales well for multiple services.

Example: the rmi registry



Dependency Injection

Definition

A software design pattern that allows:

- removing hard-coded dependencies and makes it possible to change them at run-time or compile-time
- selecting among multiple implementations of a given dependency interface at run time, or via configuration files, instead of at compile time

It is used to (e.g.):

- load plugins dynamically
- choose stubs or mock objects in test environments vs. real objects in production environments.
- adapt a generic framework to specific needs
- locating and initializing software services

A core principles is the separation of behavior from dependency resolution.

What does it do?

It injects the depended-on element (object or value, etc.) to the destination automatically by knowing the requirement of the destination.

Another pattern, called dependency lookup, is a regular process and reverse process to dependency injection.

Constructor Injection

```
public void testWithSpring() throws Exception {
    ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");
    MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());
  }
```

Setter Injection

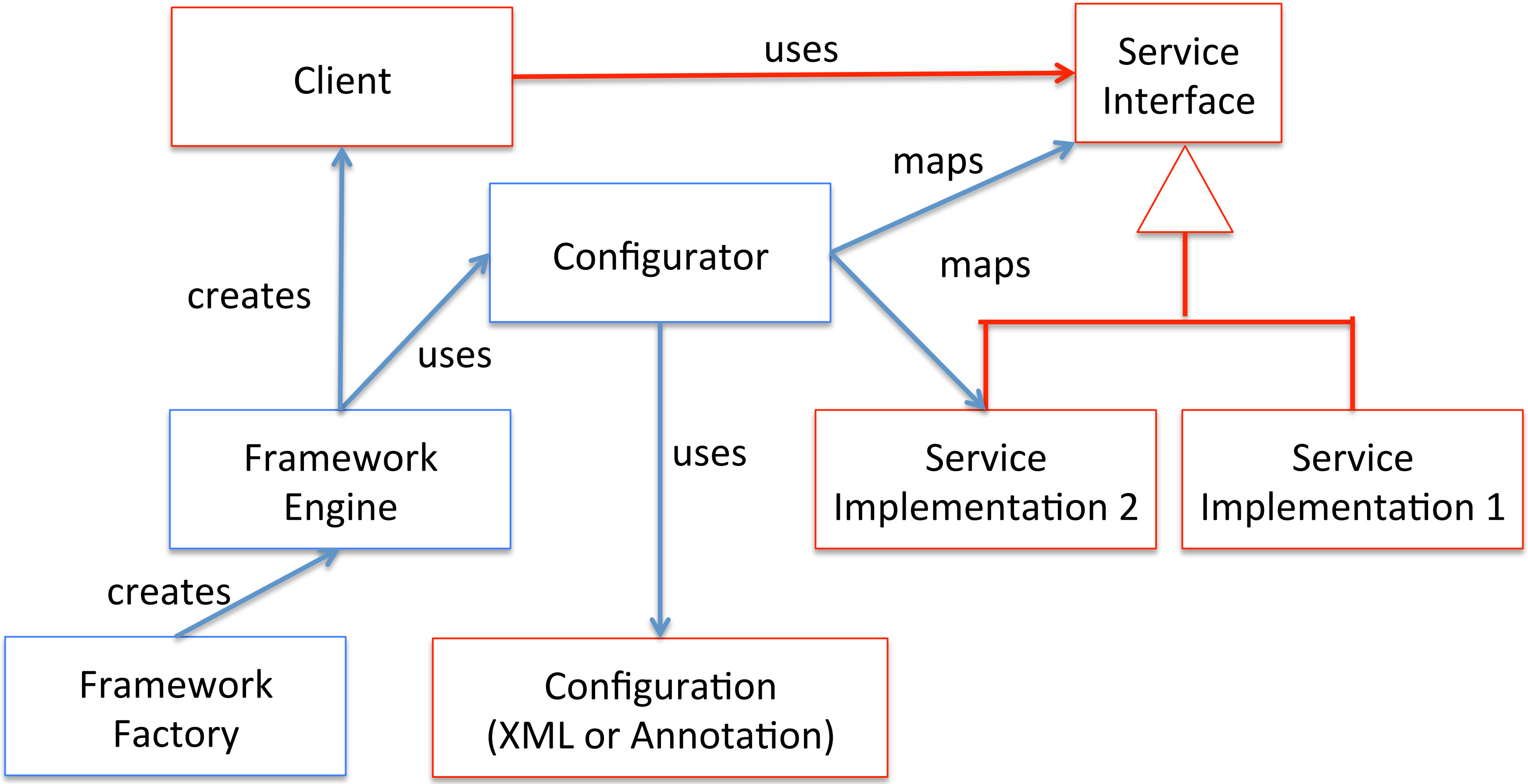
```
public void testWithPico() {
    MutablePicoContainer pico = configureContainer();
    MovieLister lister = (MovieLister) pico.getComponentInstance(MovieLister.class);
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());
  }
```

Constructor Injection

```
public void testIface() {
    configureContainer();
    MovieLister lister = (MovieLister)container.lookup("MovieLister");
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());
  }
```

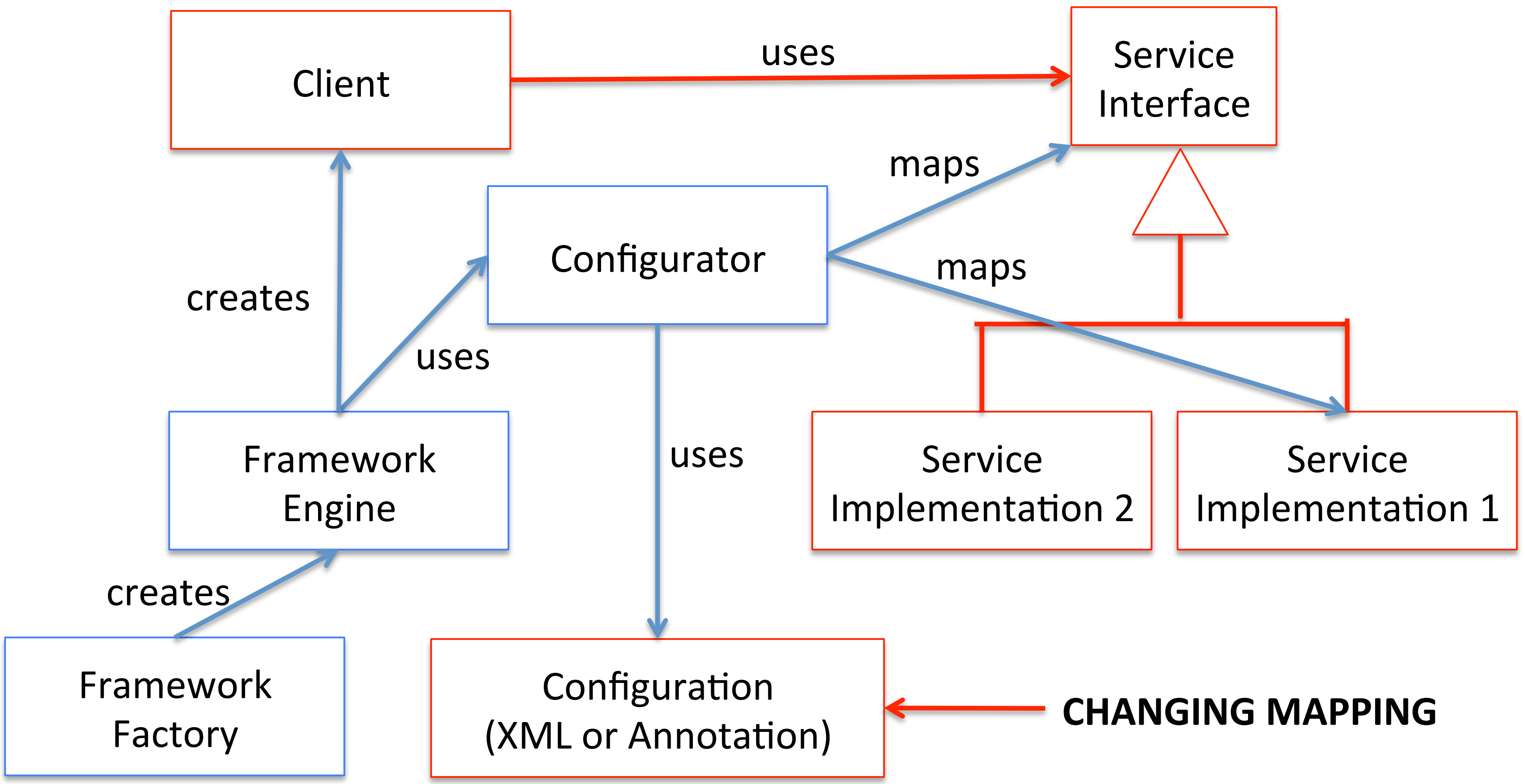
Interface Injection

Dependency Injection
Framework and Client



Dependency Injection

Framework and Client



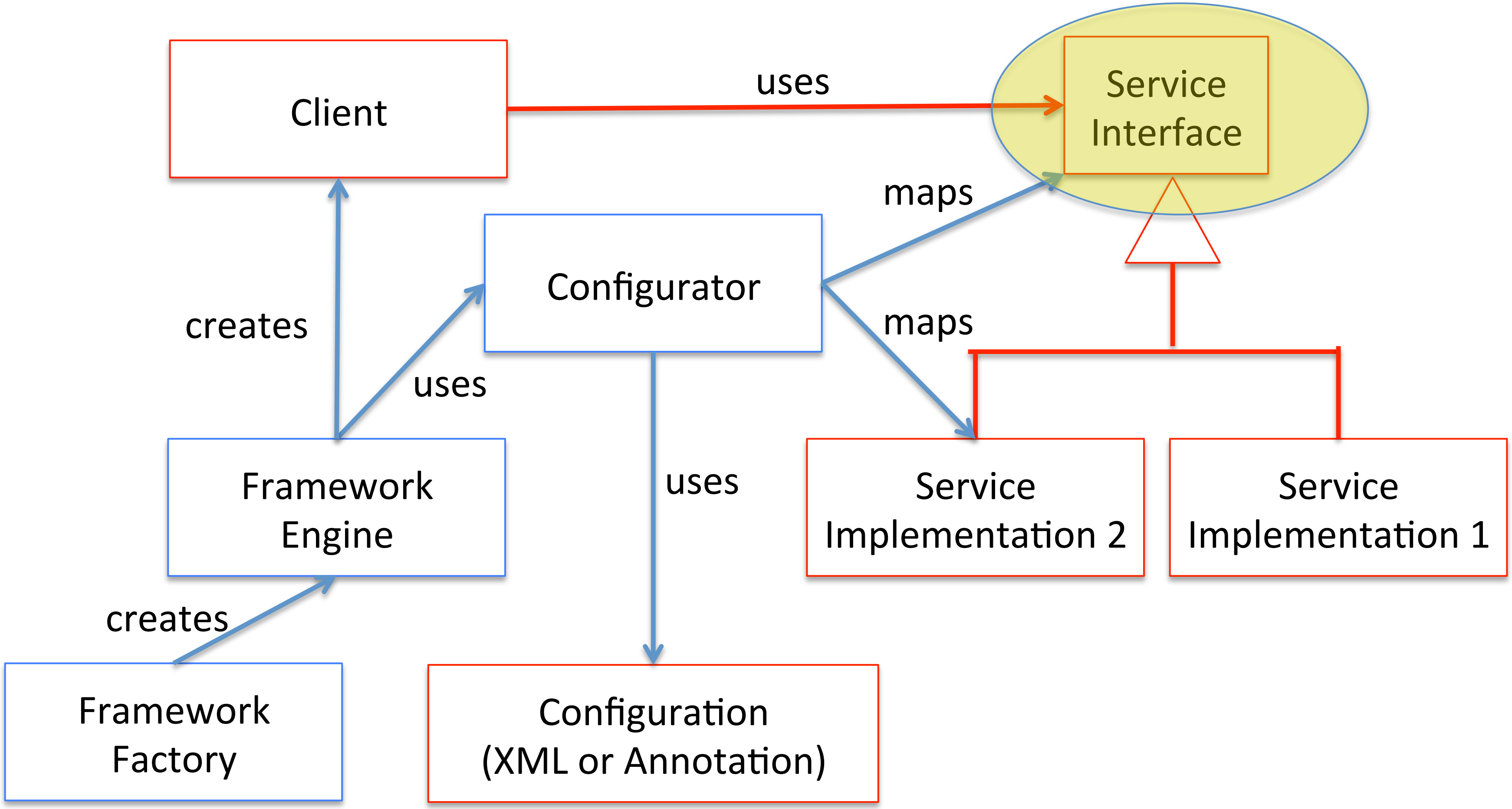
Dependency Injection: implementation of a simple framework

MiniDIFramework

1 - Framework usage

inspired by and modified from
http://java-bytes.blogspot.it/2010/04/create-your-own-dependency-injection.html

Dependency Injection
Framework and Client

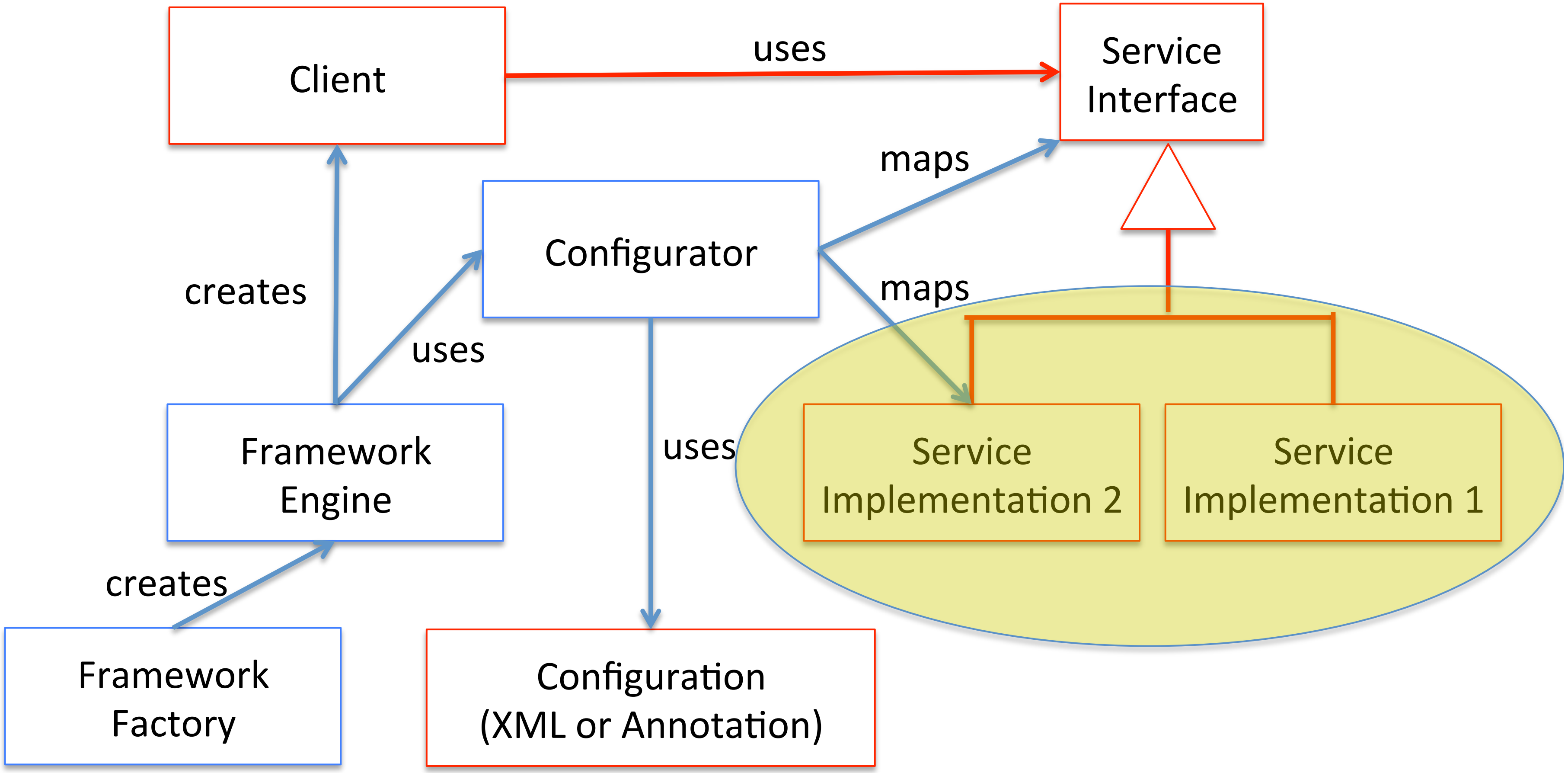


Define the Interfaces for your services

```
package miniDI.Demo.serviceDeclaration;
public interface iReader {
   public String readData();
}
```

```
package miniDI.Demo.serviceDeclaration;
public interface iWriter {
   public void writeData(String val);
}
```

Dependency Injection
Framework and Client



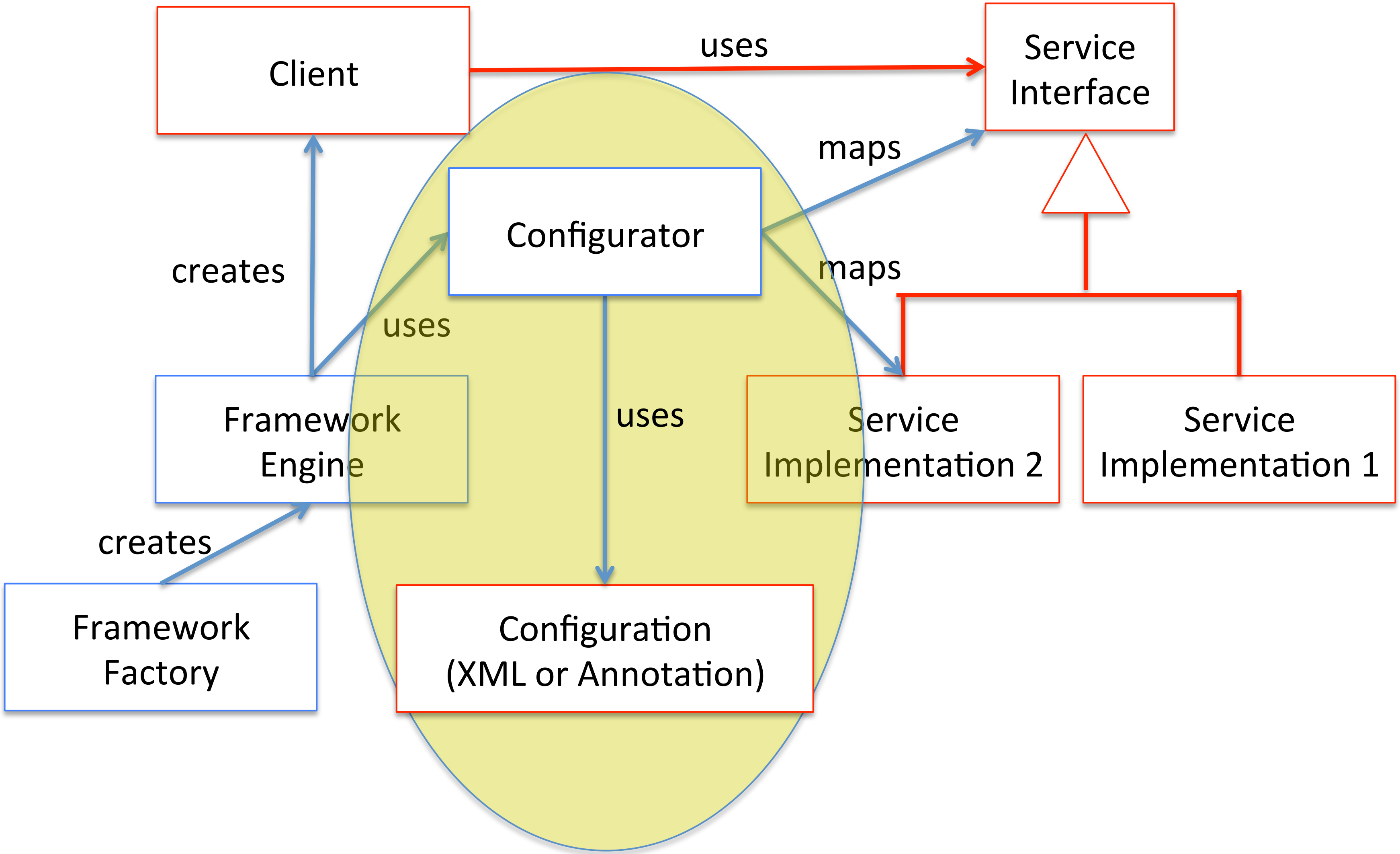
Define the implementations for you services

```
public class WriterToMemory implements iWriter {
    static String inMemory=null;
    @Override
    public void writeData(String data) {  inMemory=data;  }
}

public class WriterToFile implements iWriter {
    @Override
    public void writeData(String data) {
        File f=new File("data.txt");
        try {
            FileOutputStream fis=new FileOutputStream(f);
            ObjectOutputStream ois=new ObjectOutputStream(fis);
            ois.writeObject(data);
            ois.close();    fis.close();
        } catch (IOException ex) {  ex.printStackTrace(); }
    }
}
```

Define the implementations for you services

```
public class ReaderFromMemory implements iReader {
   @Override
   public String readData() { return "read from memory: "+WriterToMemory.inMemory;  }
}
public class ReaderFromFile implements iReader {
   @Override
   public String readData() {
      String retval=null;
      File f=new File("data.txt");
      ObjectInputStream ois=null;
      try {
         FileInputStream fis=new FileInputStream(f);
         ois=new ObjectInputStream(fis);
         retval=(String)ois.readObject();
         ois.close();   fis.close();
      } catch (IOException | ClassNotFoundException ex) {ex.printStackTrace(); }
      return "read from file: "+retval;
 } }
```

Dependency Injection
Framework and Client



Prepare your configurator

```
@ServiceDeclarationPackage("miniDI.Demo.serviceDeclaration")
@ServiceImplementationPackage("miniDI.Demo.serviceImplementation")
@Pairs({
   @Pair(key="iReader", value="ReaderFromFile"),
   @Pair(key="iWriter", value="WriterToFile"),
   //@Pair(key="iReader", value="ReaderFromMemory"),
   //@Pair(key="iWriter", value="WriterToMemory")
})
/**
 * The Configurator class must extend Abstract Configurator and provide (via annotation)
 * the mappings between the declared service and the implemented services
 */
public class Configurator extends AbstractConfigurator {
}
```

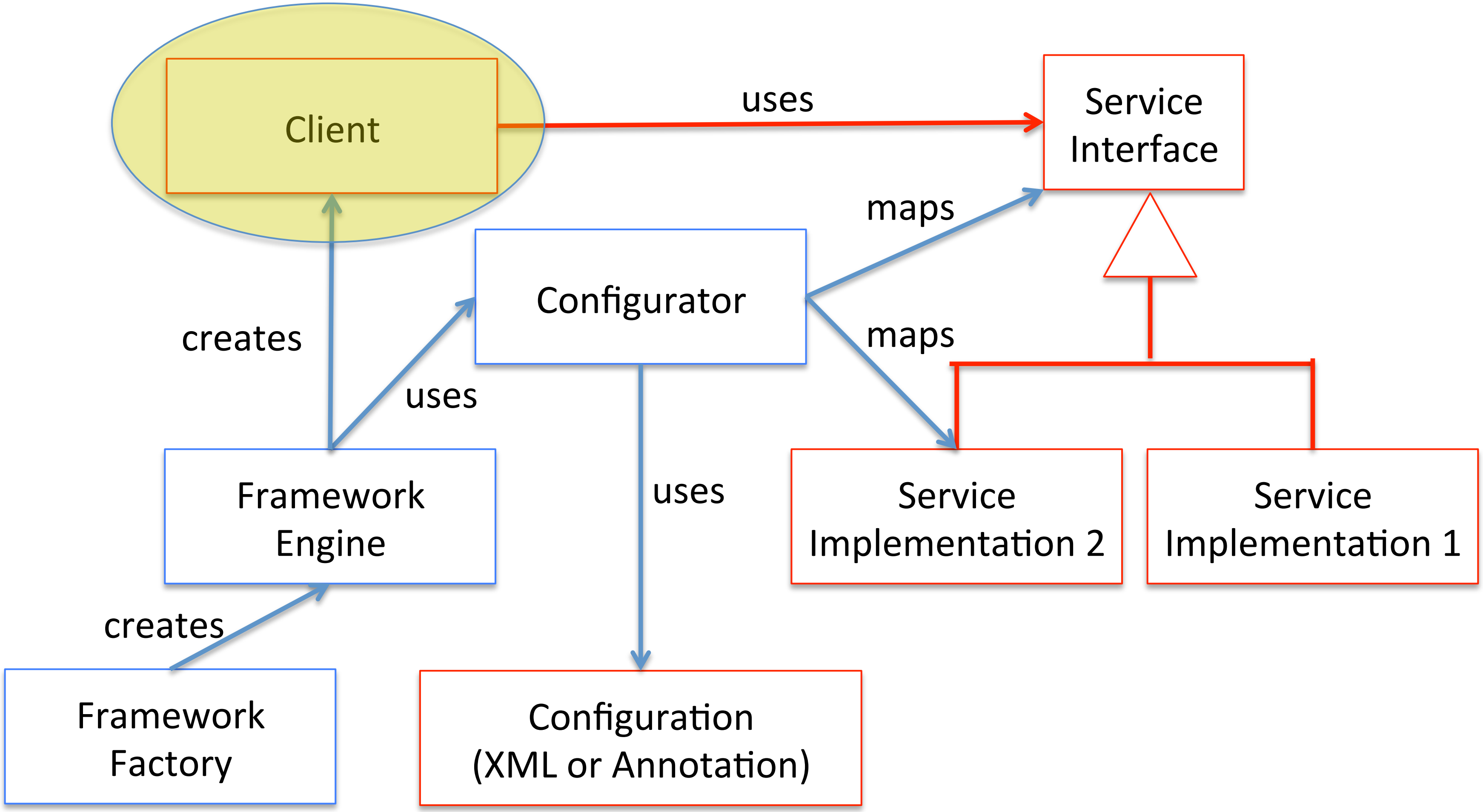
or prepare your xml config file

```
?xml version="1.0" encoding="UTF-8"?>
<miniDI>
  <ServiceDeclarationPackage>
  miniDI.Demo.serviceDeclaration
  </ServiceDeclarationPackage>
  <ServiceImplementationPackage>
  miniDI.Demo.serviceImplementation
  </ServiceImplementationPackage>
  <Pairs>
    <Pair>
      <key>iReader</key>
      <value>ReaderFromMemory</value>
    </Pair>
    <Pair>
      <key>iWriter</key>
      <value>WriterToMemory</value>
    </Pair>
  </Pairs>
</miniDI>
```

```
<!--
    <Pair>
      <key>iReader</key>
      <value>ReaderFromFile</value>
    </Pair>
    <Pair>
      <key>iWriter</key>
      <value>WriterToFile</value>
    </Pair>
-->
```

Dependency Injection

Framework and Client



Client

uses

Service Interface

maps

Configurator

maps

creates

uses

Framework Engine

uses

Service Implementation 2

Service Implementation 1

creates

Framework Factory

Configuration (XML or Annotation)

Write your business class, decoupled from the service implementation and choice

```
public class Client {
	private iReader service;
	private iWriter newService;

	@Inject
	public Client(iReader service, iWriter newService) {
		this.service = service;
		this.newService = newService;
	}

	public void doSomething() {
		String data=JOptionPane.showInputDialog("give me a string");
		newService.writeData(data);
		String a=service.readData();
		System.out.println(a);
	}
}
```

Instantiate your business class by using the framework and the configurator

```
public class App {
  public static void main(String[] args) throws Exception {
     MiniDIFramework f = MiniDIFrameworkFactory.getMiniDIFramework(new Configurator());
     Client client = (Client) f.createAndInjectDependencies(Client.class);
     if (client==null) {
      System.err.println("Client has not been correctly instantiated");
      System.exit(1);
    }
    client.doSomething();
  }
}
```

Simple Factory

Factory Method

THE MAIN PROGRAM

Run

- output with xml file

```
looking for file /Users/ronchet/NetBeansProjects/MiniDi/miniDiCli/miniDI.xml...
...found. Proceeding with XML configuration
Found mapping:
==>miniDI.Demo.serviceDeclaration.iReader==>miniDI.Demo.serviceImplementation.ReaderFromMemory
Found mapping:
==>miniDI.Demo.serviceDeclaration.iWriter==>miniDI.Demo.serviceImplementation.WriterToMemory
read from memory: test String
```

- output without xml file

```
looking for file /Users/ronchet/NetBeansProjects/MiniDi/miniDiCli/miniDI.xml...
...not found. Proceeding with annotation configuration
Found mapping:
==>miniDI.Demo.serviceDeclaration.iReader==>miniDI.Demo.serviceImplementation.ReaderFromFile
Found mapping:
==>miniDI.Demo.serviceDeclaration.iWriter==>miniDI.Demo.serviceImplementation.WriterToFile
read from file: Test String
```

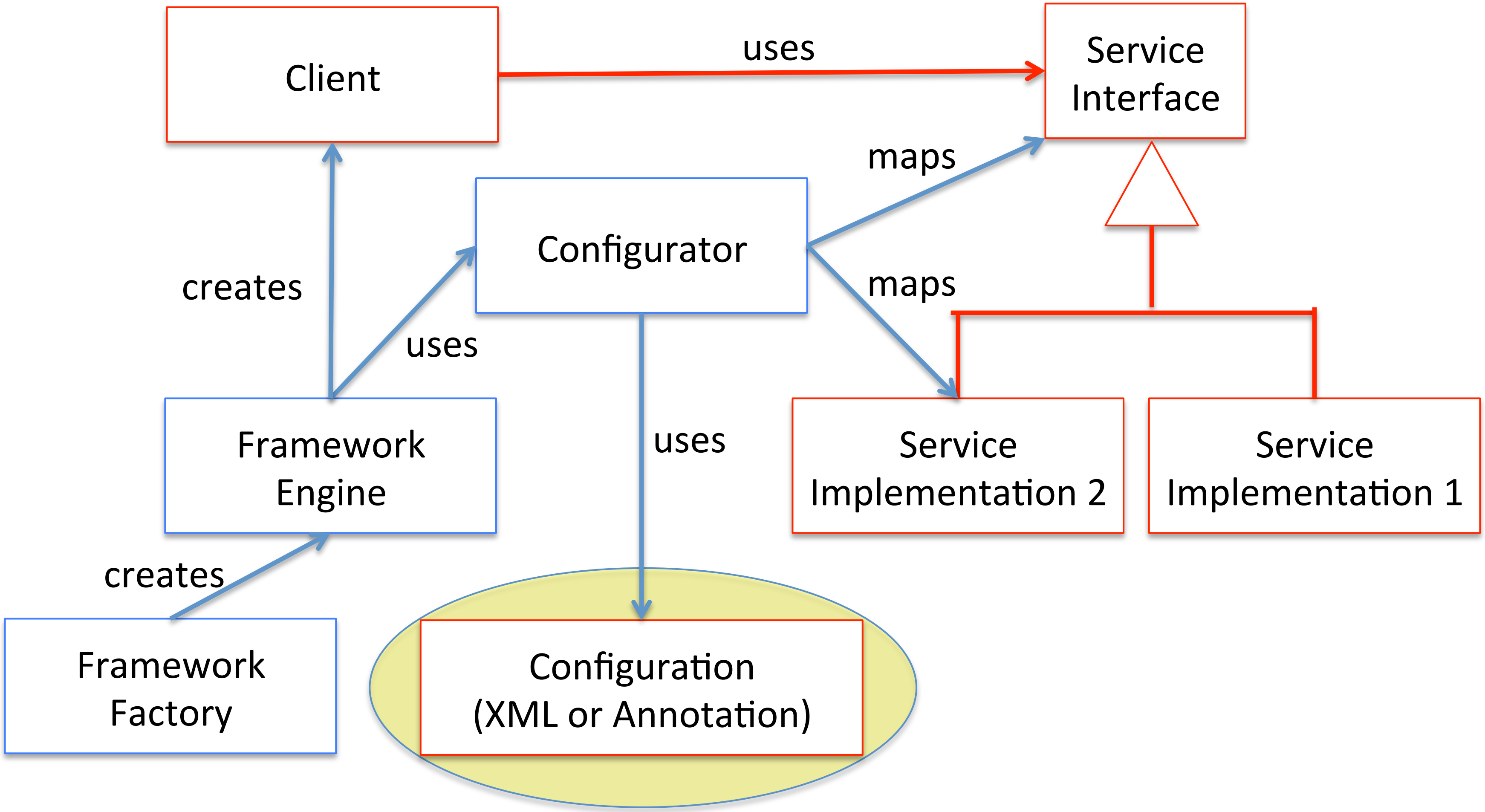
Dependency Injection: implementation of a simple framework

MiniDIFramework

2 - Framework implementation

inspired by and modified from
http://java-bytes.blogspot.it/2010/04/create-your-own-dependency-injection.html

Dependency Injection
Framework and Client



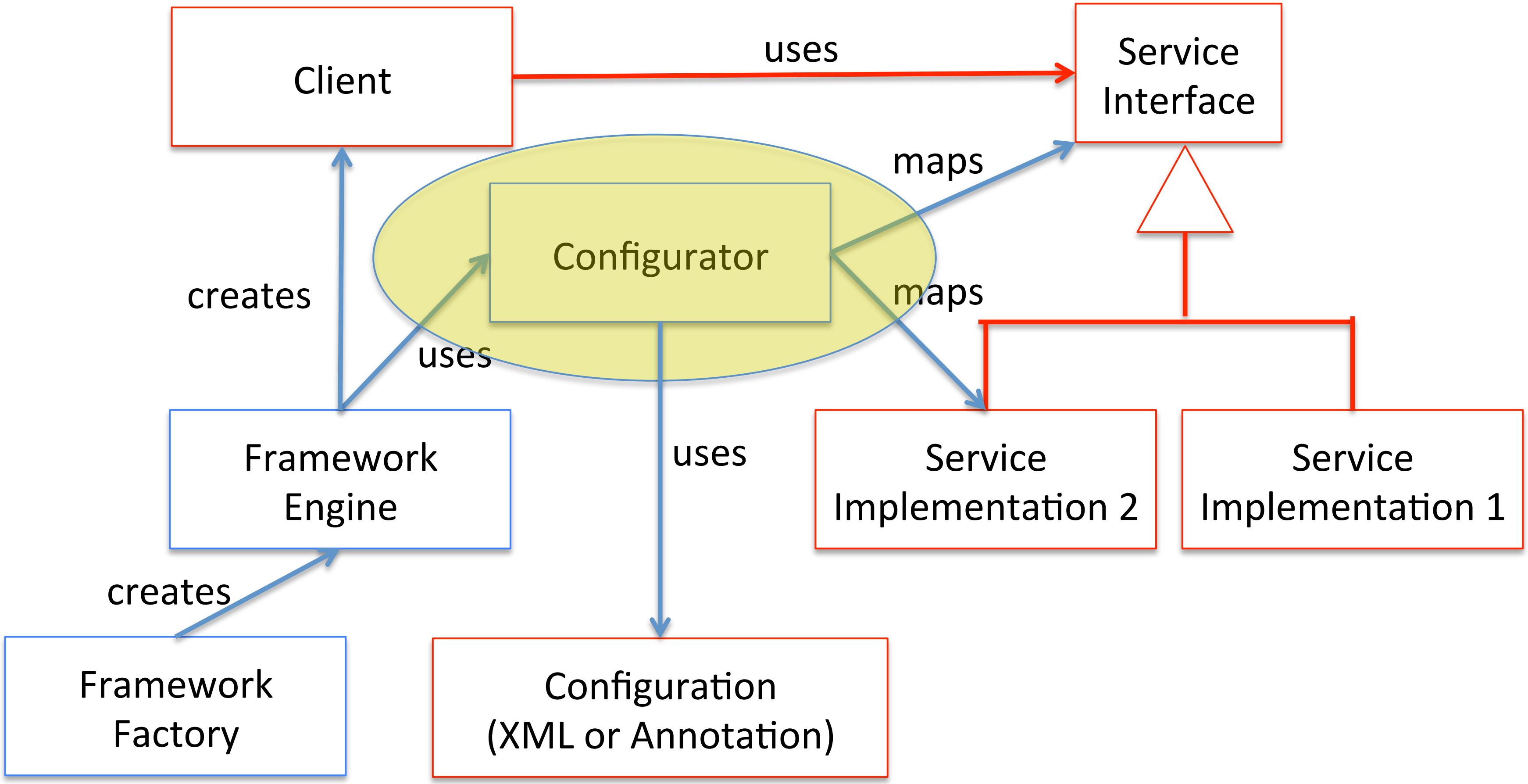
The annotations - 1

```
@Target({ CONSTRUCTOR })
@Retention(RUNTIME)
public @interface Inject {}

@Target({ ElementType.TYPE })
@Retention(RUNTIME)
public @interface ServiceImplementationPackage {
   String value();
}

@Inherited
@Target({ ElementType.TYPE })
@Retention(RUNTIME)
public @interface ServiceDeclarationPackage {
   String value() default "miniDI.framework.serviceDeclaration";
}
```

The annotations - 2

```
@Target({ ElementType.ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface Pair {
   String key();
   String value();
}

@Target({ ElementType.TYPE })
@Retention(RUNTIME)
@Documented
public @interface Pairs {
   Pair[] value();
}
```

Dependency Injection
Framework and Client



AbstractConfigurator

```
@ServiceDeclarationPackage
public abstract class AbstractConfigurator {

  // this HashMap keeps the mappings between interfaces and implementations
  private Map<Class<?>, Class<?>> classMap = new HashMap<>();

  // this method extracts the info from annotations, and saves them in the Map
  protected AbstractConfigurator() {
    File xml = new File("miniDI.xml");
    System.out.println("looking for file " + xml.getAbsolutePath() + "...");
    if (xml.exists()) {
      System.out.println("...found. Proceeding with XML configuration");
      configureWithXML(xml);
    } else {
      System.out.println("...not found. Proceeding with annotation configuration");
      configureWithAnnotations();
    }
  }
```

AbstractConfigurator - 2

```
private void insertDataIntoMap(String serviceDeclarationPackage, key,
                                                     serviceImplementationPackage,value) {
     Class keyClass = null, valueClass = null;
     try {
        keyClass = Class.forName(serviceDeclarationPackage + "." + key);
        valueClass = Class.forName(serviceImplementationPackage + "." + value);
        System.out.println("Found mapping:\n"+
               "==>"+serviceDeclarationPackage + "." + key+
               "==>"+serviceImplementationPackage + "." + value);
     } catch (ClassNotFoundException ex) {ex.printStackTrace); }
     createMapping(keyClass, valueClass);
  }

  // record a mapping in the Map
  private <T> void createMapping(Class<T> baseClass, Class<? extends T> subClass) {
     classMap.put(baseClass, subClass.asSubclass(baseClass));
  }
```

AbstractConfigurator -3

```
// find which implementation (return value) corresponds to a
given interface (input parameter)
  <T> Class<? extends T> getMapping(Class<T> type) {
    Class<?> implementation = classMap.get(type);
    if (implementation == null) {
      throw new IllegalArgumentException("Couldn't
                                find the mapping for : " + type);
    }
    return (Class<T>)implementation;// .asSubclass(type);
  }
```

AbstractConfigurator - 4

```
private void configureWithAnnotations() {
   // From the annotation find out which package hosts the service declarations
   ServiceDeclarationPackage sd = (ServiceDeclarationPackage)
                  this.getClass().getAnnotation(ServiceDeclarationPackage.class);
   // From the annotation find out which package hosts the service implementation
   ServiceImplementationPackage si = (ServiceImplementationPackage)
                  this.getClass().getAnnotation(ServiceImplementationPackage.class);
   // From the annotation, get all the pairs declaration-imlementation,
   // and create a mapping between the components of each pair
   Pairs p = (Pairs) this.getClass().getAnnotation(Pairs.class);
   Pair[] coppie = p.value();
   for (Pair c : coppie) {
      insertDataIntoMap(sd.value(),c.key(),si.value(),c.value() );
   }
}
```

AbstractConfigurator - 5

```
private void configureWithXML(File xmlFile) {
    DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
    domFactory.setNamespaceAware(true);
    Document doc = null;
    try {
        DocumentBuilder builder = domFactory.newDocumentBuilder();
        doc = builder.parse(xmlFile.getAbsolutePath());
    } catch (ParserConfigurationException | SAXException | IOException ex) {
        Logger.getLogger(AbstractConfigurator.class.getName()).log(Level.SEVERE, null, ex);
    }
    // prepare the XPath expression
    XPathFactory factory = XPathFactory.newInstance();
    XPath xpath = factory.newXPath();
    String serviceDeclarationPackage = null, serviceImplementationPackage = null;
    NodeList keys = null, values = null;
```

AbstractConfigurator - 6

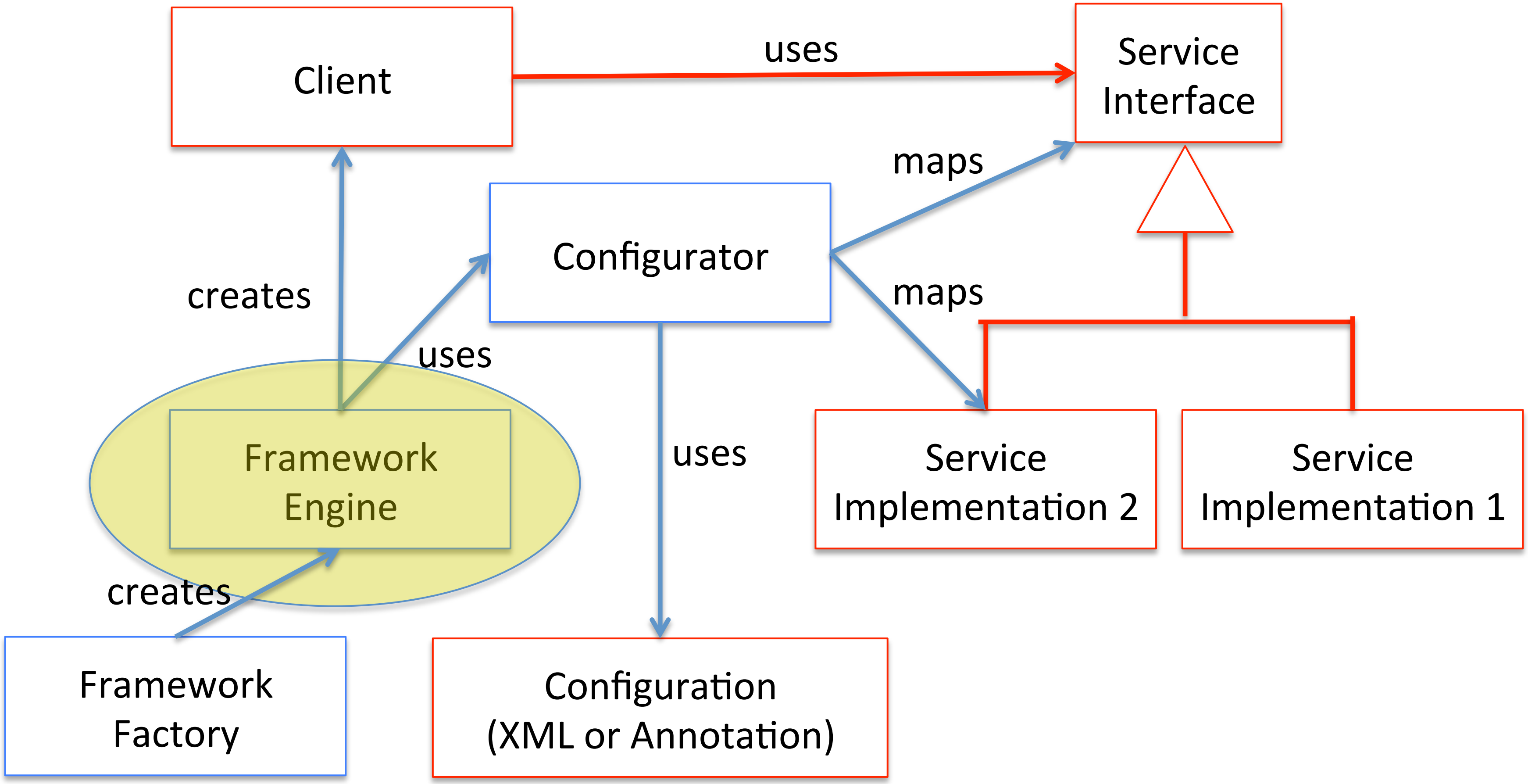
```
try {
        XPathExpression expr1 = xpath.compile("/miniDI/ServiceDeclarationPackage/text()");
        serviceDeclarationPackage = ((Node) expr1.evaluate(doc,
              XPathConstants.NODE)).getNodeValue().trim();
        expr1 = xpath.compile("/miniDI/ServiceImplementationPackage/text()");
        serviceImplementationPackage = ((Node) expr1.evaluate(doc,
              XPathConstants.NODE)).getNodeValue().trim();
      expr1 = xpath.compile("/miniDI/Pairs/Pair/key/text()");
        keys = (NodeList) expr1.evaluate(doc, XPathConstants.NODESET);
      expr1 = xpath.compile("/miniDI/Pairs/Pair/value/text()");
        values = (NodeList) expr1.evaluate(doc, XPathConstants.NODESET);
    } catch (XPathExpressionException ex) { ex.printStackTrace() }
    // From the xml, get all the pairs declaration-implementation,
    // and create a mapping between the components of each pair
    for (int i = 0; i < keys.getLength(); i++) {
        insertDataIntoMap(
              serviceDeclarationPackage,  keys.item(i).getNodeValue().trim(),
              serviceImplementationPackage,  values.item(i).getNodeValue().trim()
              );
    }
  }
```

MiniDIFramework

```
public class MiniDIFramework {

    private AbstractConfigurator configurator;

    MiniDIFramework(AbstractConfigurator configurator) { // package visibility!
        this.configurator = configurator;
    }
    public Object createAndInjectDependencies (Class clazz) throws Exception {
        …}
}
```

Dependency Injection

Framework and Client



MiniDIFramework

```
@SuppressWarnings("unchecked")
public Object createAndInjectDependencies(Class clazz) throws Exception {
  if (clazz != null) {
    int index = 0;
     for (Constructor constructor : clazz.getConstructors()) {
       if (constructor.isAnnotationPresent(Inject.class)) {
         if (index == 0) { //restrict to only one constructor injection
           index++;
           Class[] parameterTypes = constructor.getParameterTypes();
           Object[] objArr = new Object[parameterTypes.length];
           for (Class c : parameterTypes) {
             Class dependency = configurator.getMapping(c);
             if (c.isAssignableFrom(dependency)) {
               objArr[i++] = dependency.getConstructor().newInstance();
            } }
           Object resObj = clazz.getConstructor(parameterTypes).newInstance(objArr);
           return resObj;
   } }  }
  return null; // if class was null or no annotated constructor was found return null
 }
}
```

finds which constructor of the given class (input parameter) is annotated,

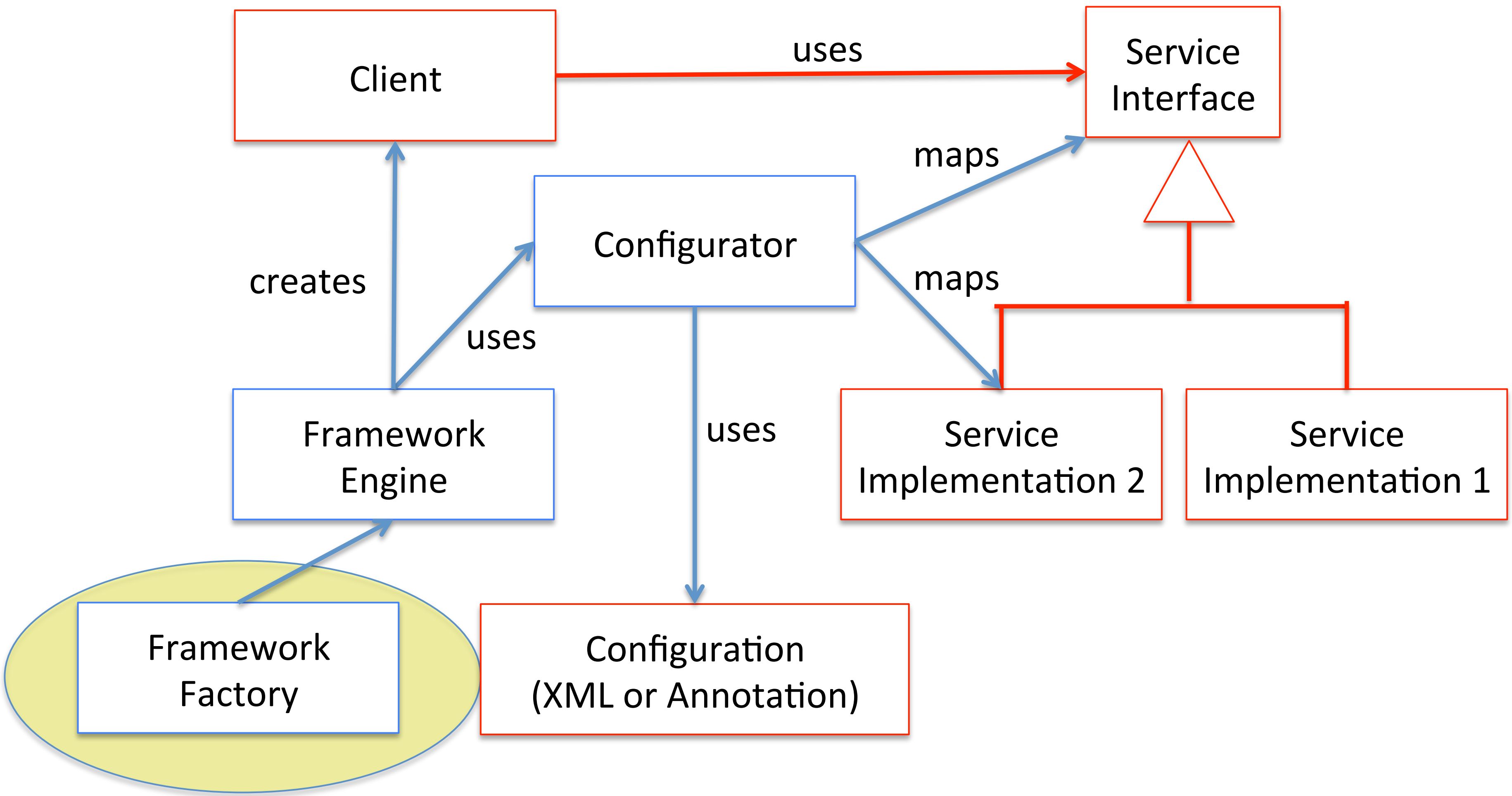
discover the dependency types

find an implementation bound to each parameter type,

create an instance of each implementation

create the desired object, with instances of its dependencies injected

Dependency Injection
Framework and Client



MiniDIFrameworkFactory

```
public class MiniDIFrameworkFactory {
        private static boolean hasFrameworkBeenInstanciated=false;
        public static MiniDIFramework
                    getMiniDIFramework(AbstractConfigurator configurator) {
            if (hasFrameworkBeenInstanciated) {
                System.err.println ("MiniDIFramework
                            can only be istantiated once!");
                System.exit(1);
            }
            hasFrameworkBeenInstanciated=true;
            return new MiniDIFramework(configurator);
        }
}
```

An example of using DI in JEE7 with facelets

see http://docs.oracle.com/javaee/7/tutorial/doc/cdi-basicexamples001.htm

Types of D.I.

Martin Fowler identifies three ways in which an object can get a reference to an external module, according to the pattern used to provide the dependency:

- interface injection, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.
- setter injection, in which the dependent module exposes a setter method that the framework uses to inject the dependency.
- constructor injection, in which the dependencies are provided through the class constructor.