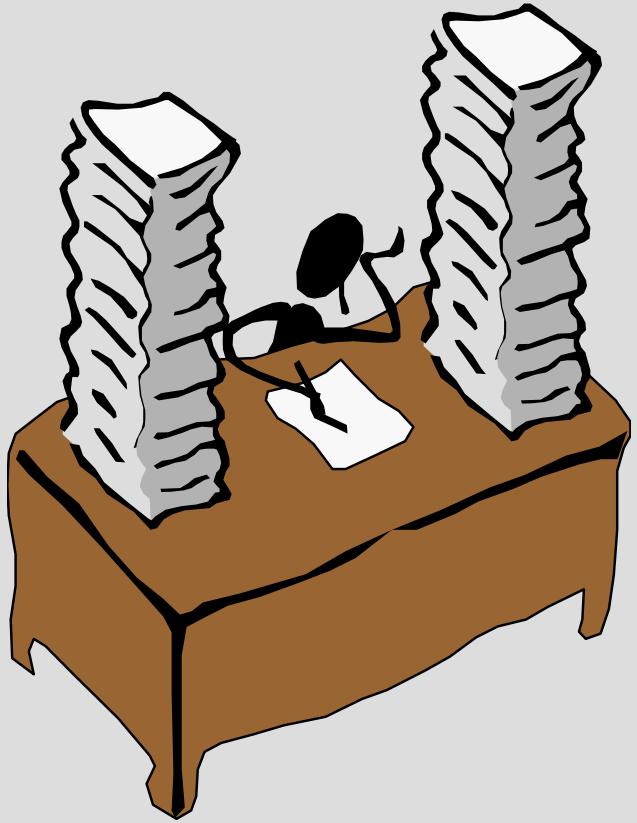
Service Oriented Computing: Web Service Development

Dr. Cristian Mateos Diaz
(http://users.exa.unicen.edu.ar/~cmateos/cos)
ISISTAN-UNICEN-CONICET

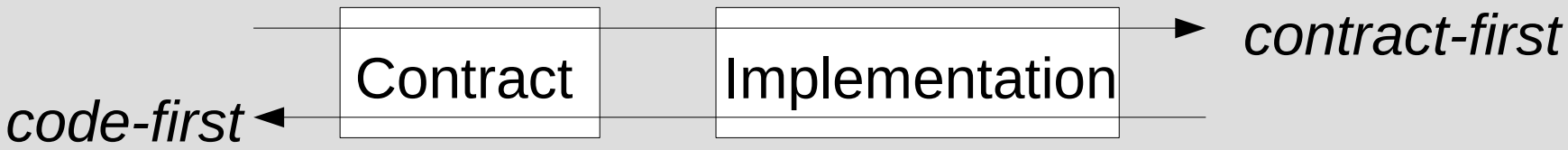
Talk outline

- Web Services
  - SOAP, WSDL (we will discuss WADL latter), UDDI
- Developing Web Services
- Testing Web Services
  - soapUI
- Consuming and invoking Web Services
- Advanced WS programming: DI and Spring



Developing Web Services: Two main approaches

- ***Contract-first*** and ***code-first*** (also ***contract-last***):
  - Under contract-first, we specify the WSDL for our service and then we provide an implementation (Eclipse WTP, Spring Web Services)
  - Under code-first, it is the other way around (Eclipse WTP)



- Which one should be followed? Well, it depends...

Developing Web Services: Contract-first vs code-first

- ***Code-first*** is said to be more appropriate for beginners (no need to write service specifications)
- ***Code-first*** is convenient and practical (e.g., automatic deployment, short-lived services, etc.)

  But...

- ***Contract-first*** ensures that contracts will not change too much over time (the code is more likely to change)
- ***Contract-first*** *better aligns with business goals*
- ***Contract-first*** enables for more reusable, rich schema types

Contract-first or code-first? An industry perspective

- The Spring Source Community: http://spring.io
- Explicitly discourages code-first because of the impedance mismatch when converting Java objects to XML
  - Loose of accuracy: for example when using **xsd:restriction**

```
<simpleType name="AirportCode">
  <restriction base="string">
     <pattern value="[A-Z][A-Z][A-Z]"/>
  </restriction>
</simpleType>
```

    could not be associated to a plain java.lang.String...
  - Unportable types (e.g., TreeMap) or type ambiguity when invoking services (Date? Calendar?)
- Another problem is technological details sneaking in service specifications (e.g., binding-related classes)

Contract-first or code-first: Is there something in between?

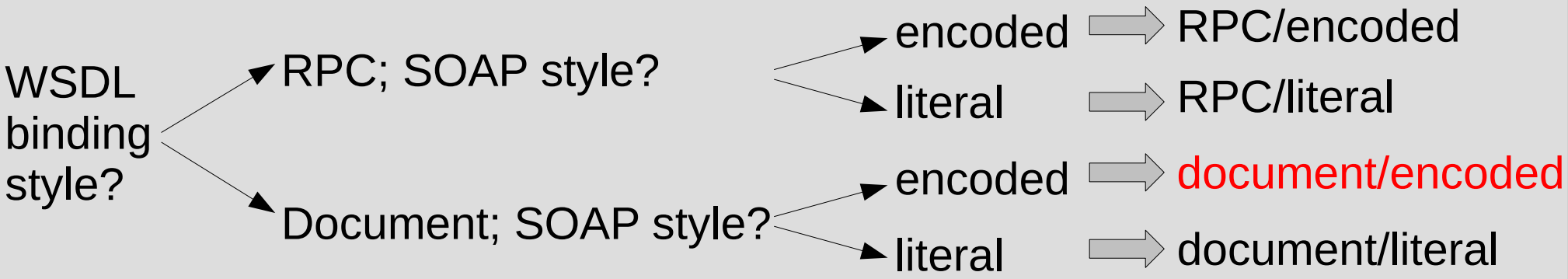
- An example: relax-ws (https://github.com/dlukyanov/relax-ws)
  - A more lightweight contract-first approach
  - Focuses on simplifying WSDL specification/generation

```
# This is "hello world" in relax-ws.
service Hello {

    port {
        operation SayHello {
            in {
                element name {xsd:string}
            }
            out {
                element message {xsd:string}
            }
        }
    }
}
```

Developing Web Services: Summary (so far)

- Contract-first vs code-first (a.k.a. contract-last)
  - Plus intermediate approaches
- “RPC” oriented versus “document” oriented…
  - Aren't these already considered in the specifications? Yes!

Developing Web Services: Binding styles



... plus another style known as *Document/literal wrapped*

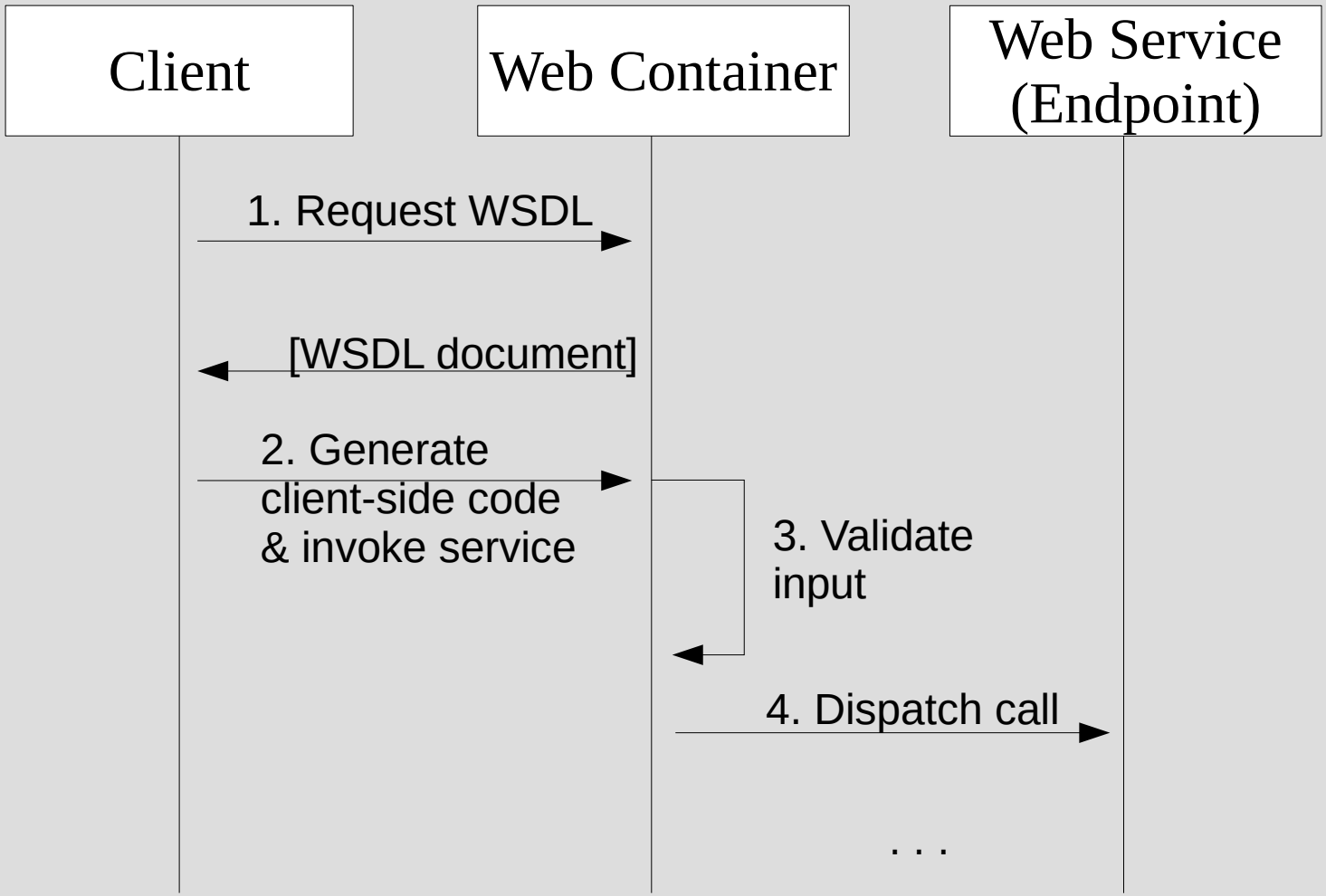
- "RPC" and "Document" merely dictate how to translate a WSDL binding to a SOAP message... the terminology is indeed unfortunate!
- "Encoded" and "Literal" make a lot more sense :)

Developing Web Services: Binding styles (cont.)

```
<wsdl:definitions .... >
    <wsdl:binding name="FooSoapBinding" type="...">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                      style="rpc|document"?/>
        <wsdl:operation name="myMethod">
            <wsdl:input name="myMethodRequest">
                <soap:body use="encoded|literal"?/>
            </wsdl:input>
            <wsdl:output name="myMethodResponse">
                <soap:body use="encoded|literal"?/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

- style= “rpc”: Input is a service name plus a number of arguments
- style=“document”: Input is any XML document
- use=“literal” versus “encoded”: whether the body of SOAP messages conform to a specific schema or explicitly indicate datatypes

Developing Web Services: Binding styles (cont.)



Example: A service for adding two numbers (rpc/encoded)

public int add(int number1, int number 2)

WSDL

```
<message name="addRequest">
        <part name="number1" type="xsd:int"/>
        <part name="number2" type="xsd:int"/>
</message>
<message name="addResponse">
        <part name="return" type="xsd:int"/>
</message>
<portType name="AddPortType">
        <operation name="add"
          parameterOrder="number1 number2">
        <input message="tns:addRequest"/>
        <output message="tns:addResponse"/>
        </operation>
</portType>
```

- *Pros*: Simplicity, dispatching time
- *Cons*: Overhead

SOAP

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:add>
         <number1 xsi:type="xsd:int">1</number1>
         <number2 xsi:type="xsd:int">2</number2>
      </ns0:add>
   </env:Body>
<env:Envelope>
```

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:addResponse>
         <return xsi:type="xsd:int">3</return>
      </ns0:addResponse>
   </env:Body>
<env:Envelope>
```

Example: A service for adding two numbers (rpc/literal)

WSDL

```
<message name="addRequest">
        <part name="number1" type="xsd:int"/>
        <part name="number2" type="xsd:int"/>
</message>
<message name="addResponse">
        <part name="return" type="xsd:int"/>
</message>
<portType name="AddPortType">
        <operation name="add"
                parameterOrder="number1
number2">
                <input message="tns:addRequest"/>
                <output
message="tns:addResponse"/>
        </operation>
</portType>
```

- *Pros*: Simplicity, dispatching time
- *Cons*: Slower validation

SOAP

```
<env:Envelope ...
    xmlns:ns0="our namespace">
    <env:Body>
        <ns0:add>
            <number1>1</number1>
            <number2>2</number2>
        </ns0:add>
    </env:Body>
<env:Envelope>
```

```
<env:Envelope ...
    xmlns:ns0="our namespace">
    <env:Body>
        <ns0:addResponse>
            <return>3</return>
        </ns0:addResponse>
    </env:Body>
<env:Envelope>
```

Example: A service for adding two numbers (document/literal)

WSDL

```
<types>
   <schema  targetNamespace="our namespace" ...>
      <element name="request">
         <xsd:complexType>
            <sequence>
               <element name="number1" type="xsd:int"/>
               <element name="number2" type="xsd:int"/>
            </sequence>
         </xsd:complexType>
      </element>
      <element name="response">
         <xsd:complexType>
            <xsd:sequence>
               <element name="result" type="xsd:int"/>
            </xsd:sequence>
         </xsd:complexType>
      </element>
   </schema>
</types>
<message name="request">
   <part name="param" element="ns0:request"/>
</message>
<message name="response">
   <part name="return" element="ns0:response"/>
</message>
```

SOAP

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:request>
         <number1>1</number1>
         <number2>2</number2>
      </ns0:request>
   </env:Body>
<env:Envelope>
```

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:response>
         <result>3</result>
      </ns0:response>
   </env:Body>
<env:Envelope>
```

- *Pros*: Faster validation
- *Cons*: Verbosity, dispatch time (Operation name is absent)

Example: A service for adding two numbers (doc/lit. wrapped)

WSDL

```
<types>
 <schema targetNamespace="our namespace" ...>
  <element name="add"><!-- named after the operation-->
   <xsd:complexType>
    <sequence>
     <element name="number1" type="xsd:int"/>
     <element name="number2" type="xsd:int"/>
    </sequence>
   </xsd:complexType>
  </element>
  <element name="addResType">
   <xsd:complexType>
    <xsd:sequence>
     <element name="result" type="xsd:int"/>
    </xsd:sequence>
   </xsd:complexType>
  </element>
</schema>
</types>
<message name="addRequest">
    <part name="param" element="ns0:add"/>
</message>
<message name="addResponse">
    <part name="return" element="ns0:addResType"/>
</message>
```

SOAP

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:add>
         <number1>1</number1>
         <number2>2</number2>
      </ns0:add>
   </env:Body>
<env:Envelope>
```

```
<env:Envelope ...
   xmlns:ns0="our namespace">
   <env:Body>
      <ns0:addResponse>
         <return>3</return>
      </ns0:addResponse>
   </env:Body>
<env:Envelope>
```

- *Pros*: Dispatching time, Fast validation
- *Cons*: Verbosity, Overloading

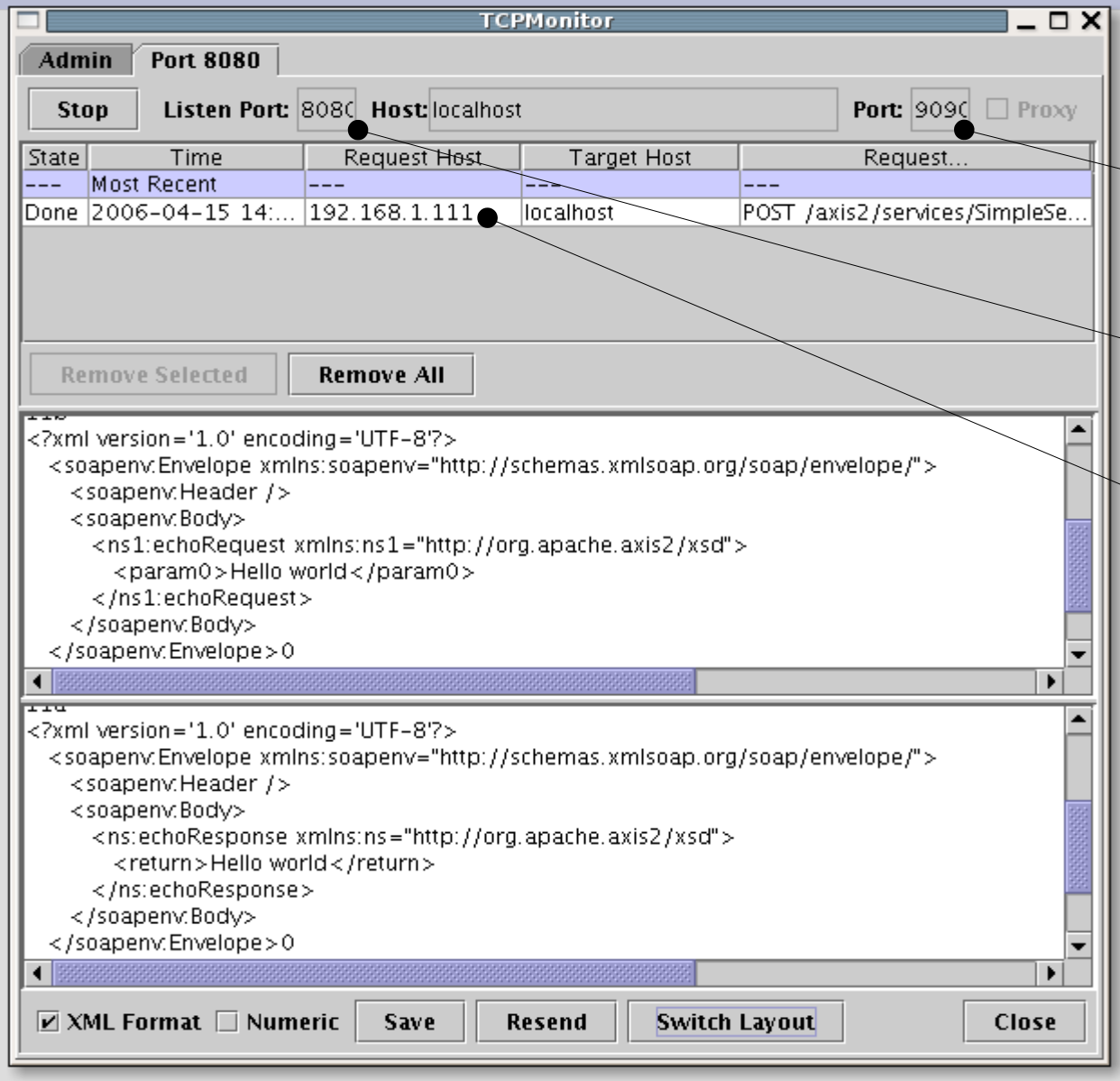
Developing Web Services: Available tools

- Eclipse WTP (Web Tools Platform)
  - http://www.eclipse.org/webtools
  - Allows users to create/publish Web Services (code/contract-first)

- Apache CXF
  - http://cxf.apache.org
  - Up-to-date framework that supports many Web Service standards

- soapUI
  - http://www.soapui.org
  - Comprehensive tool for testing Web Services
  - Integration with tcpmon

Developing Web Services: Axis tcpmon



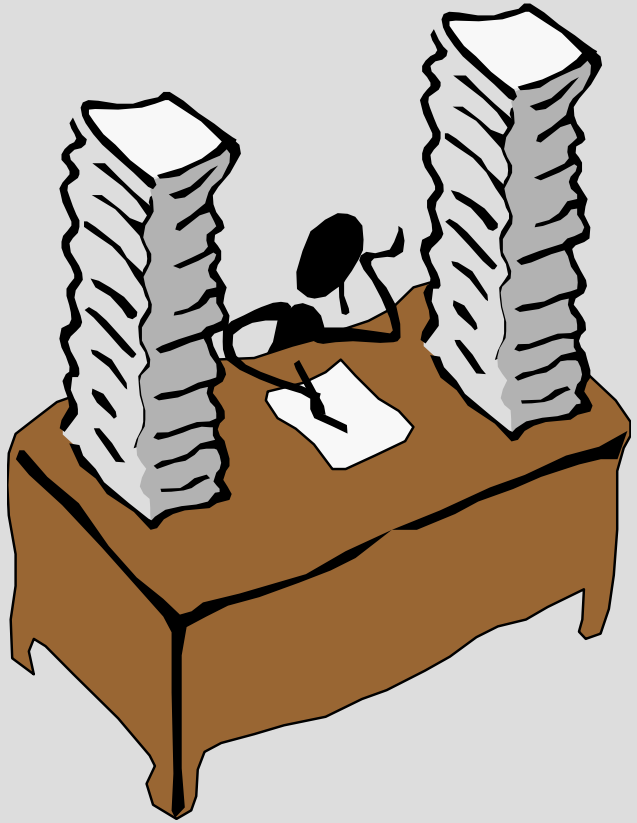
Target port

Tcpmon port

Client-side application

Talk outline

- Web Services
  - SOAP, WSDL (we will discuss WADL latter), UDDI
- Developing Web Services
- Testing Web Services
  - soapUI
- Consuming and invoking Web Services
- Advanced WS programming: DI and Spring



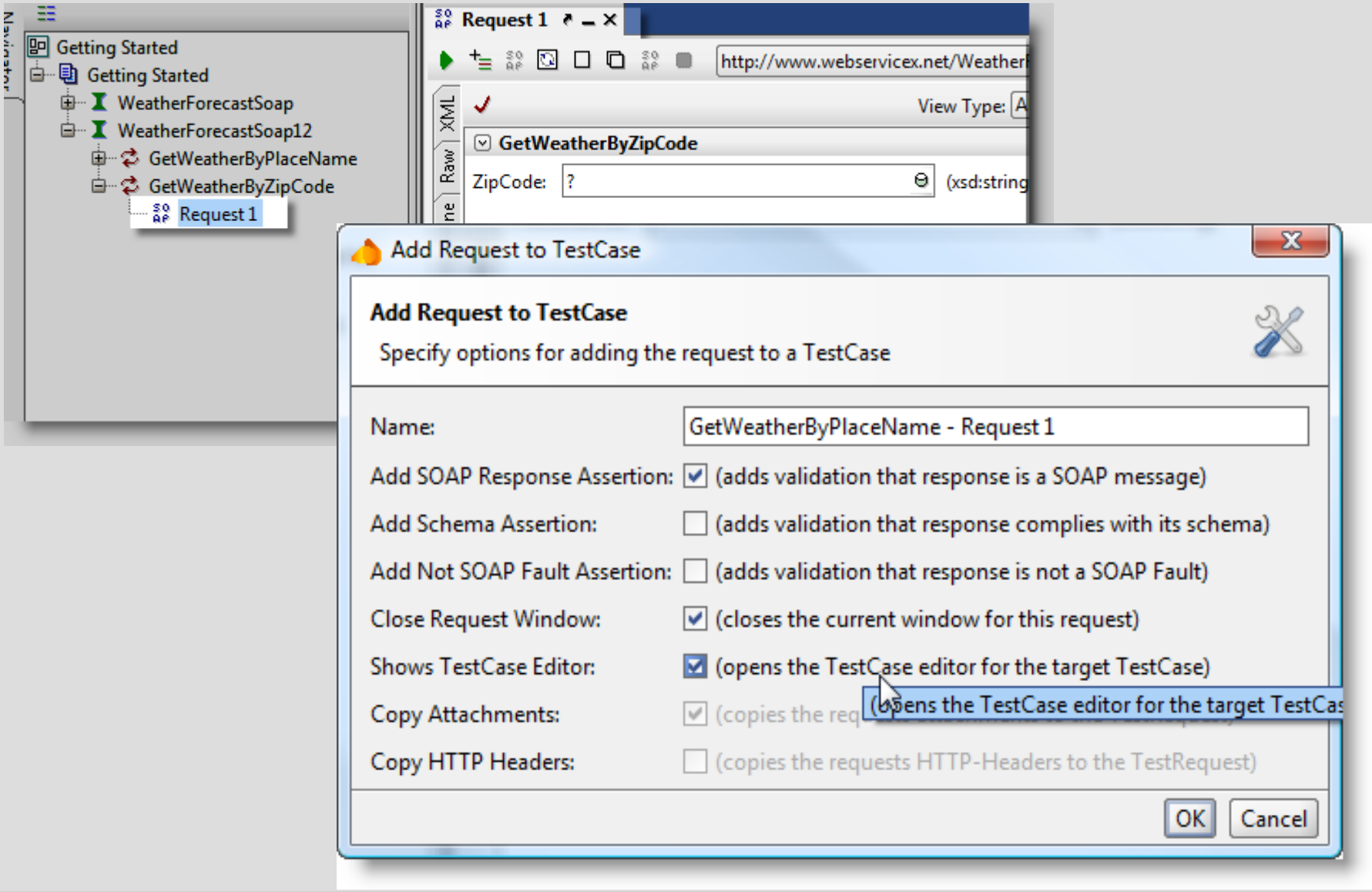
Testing Web Services

- In a broad sense, testing is the process of checking whether a piece of software does what it is supposed to do
- When dealing with WSDL-based Web Services, it implies:
  - Not only test whether a service “is listening”
  - Checking XSD compliance
  - Checking input-output correspondence (black-box testing)
- Indeed, the same issues that arise when testing traditional software also apply (e.g., coverage)

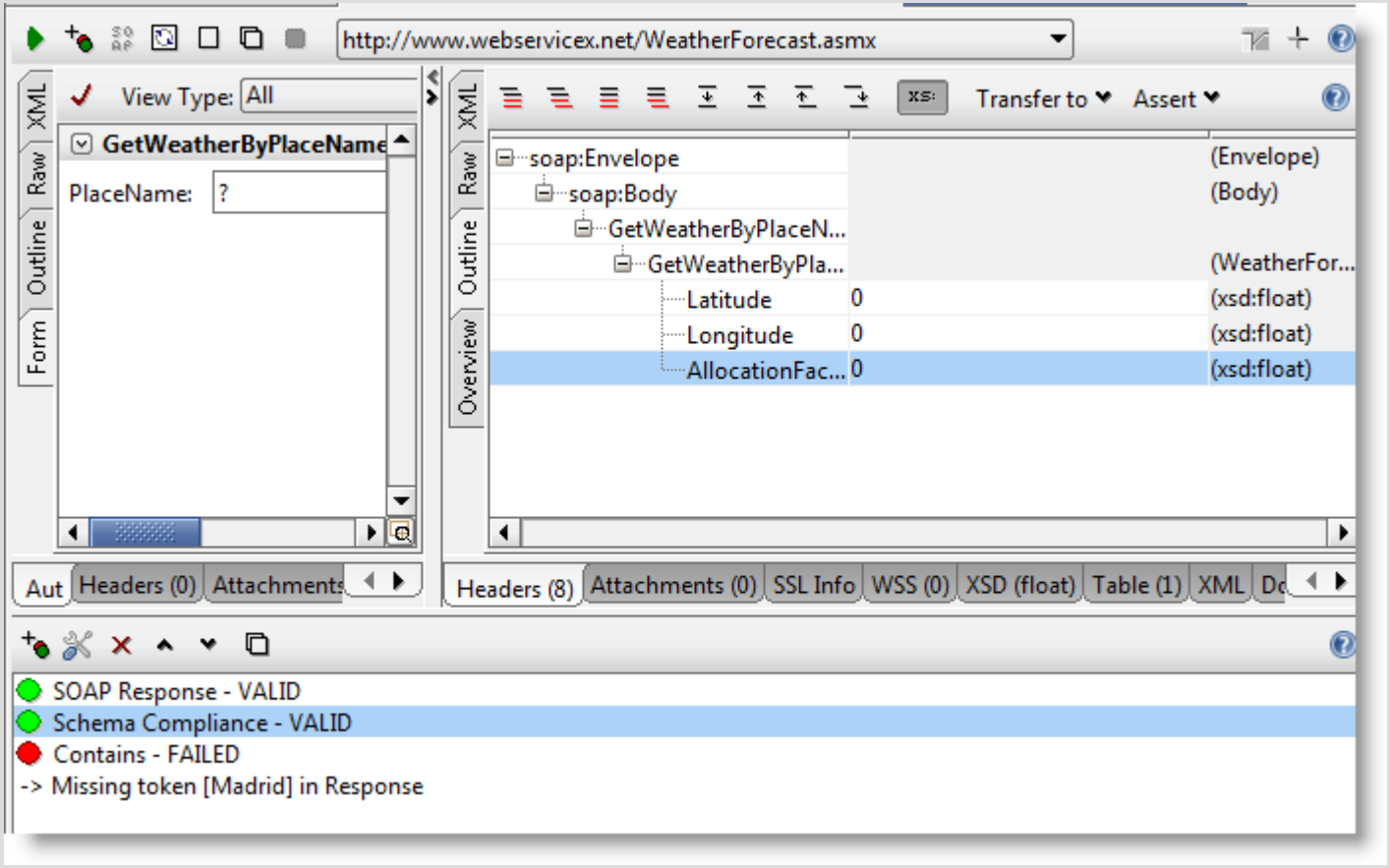
Testing Web Services: soapUI

- soapUI (http://www.soapui.org) is a comprehensive tool for testing Web Services
- It supports:
  - Functional testing supporting all the testing aspects mentioned earlier
  - Load testing (load testing itself, stress testing, soak testing, scalability testing)

Testing Web Services: soapUI (cont.)



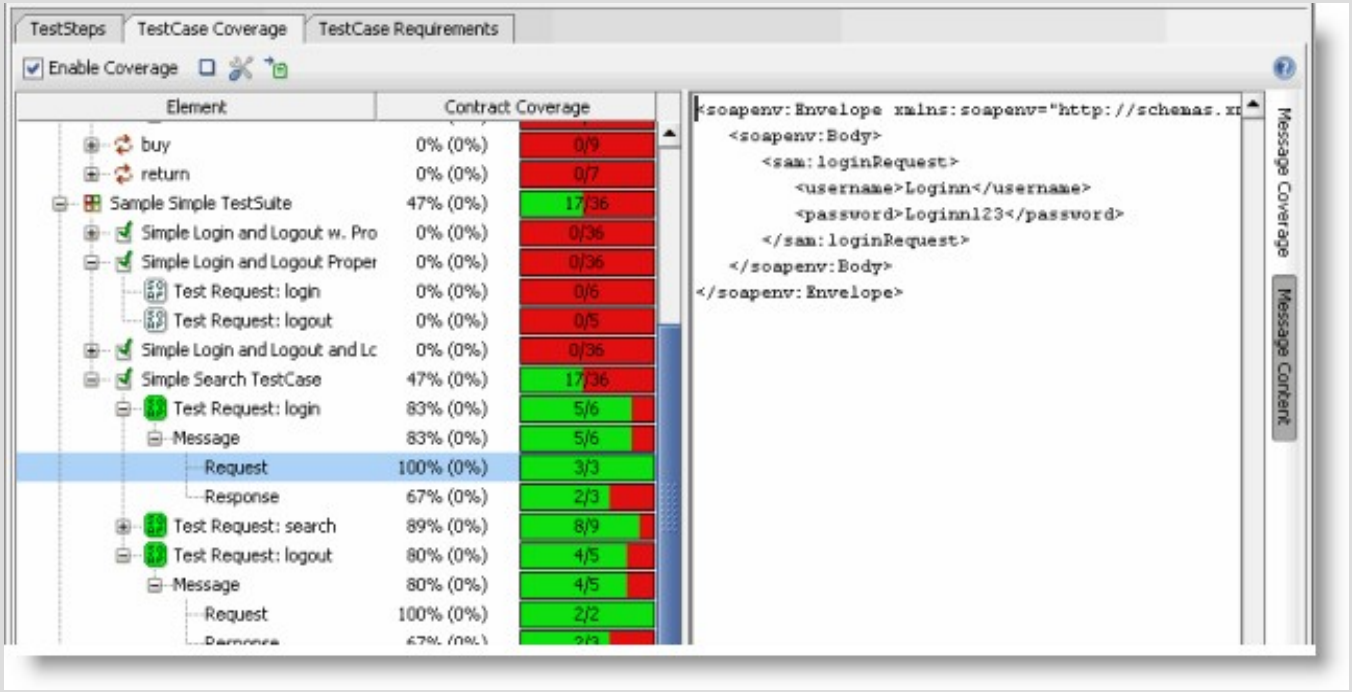
Testing Web Services: soapUI (cont.)



A lot of more “assertions” can be defined by using XML-based query languages (XPath and XQuery), WS-I, and so on...

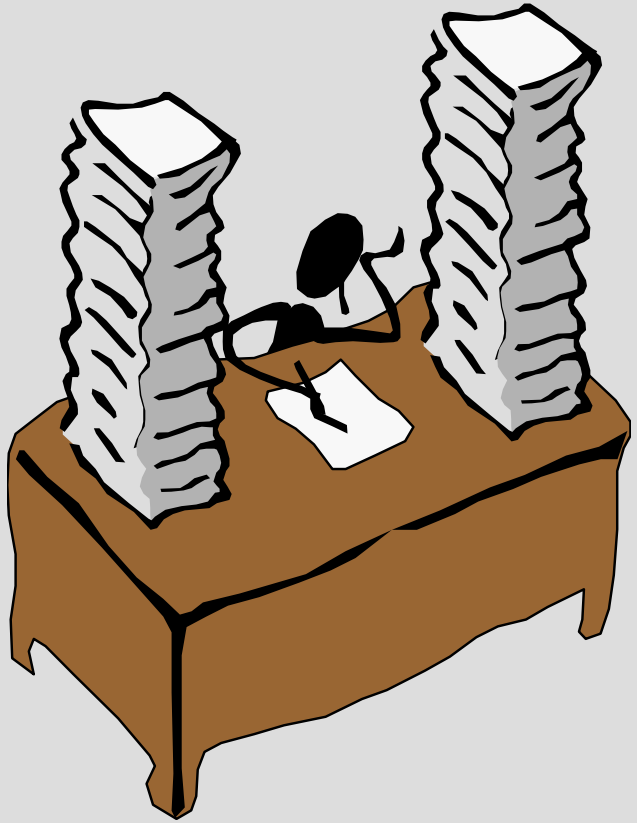
Testing Web Services: soapUI (cont.)

- In traditional Web development, coverage is used to measure how much of the actual code base that is being exercised during a given execution scenario
- soapUI computes coverage based on two separate aspects, i.e. “message coverage” and “assertion coverage”

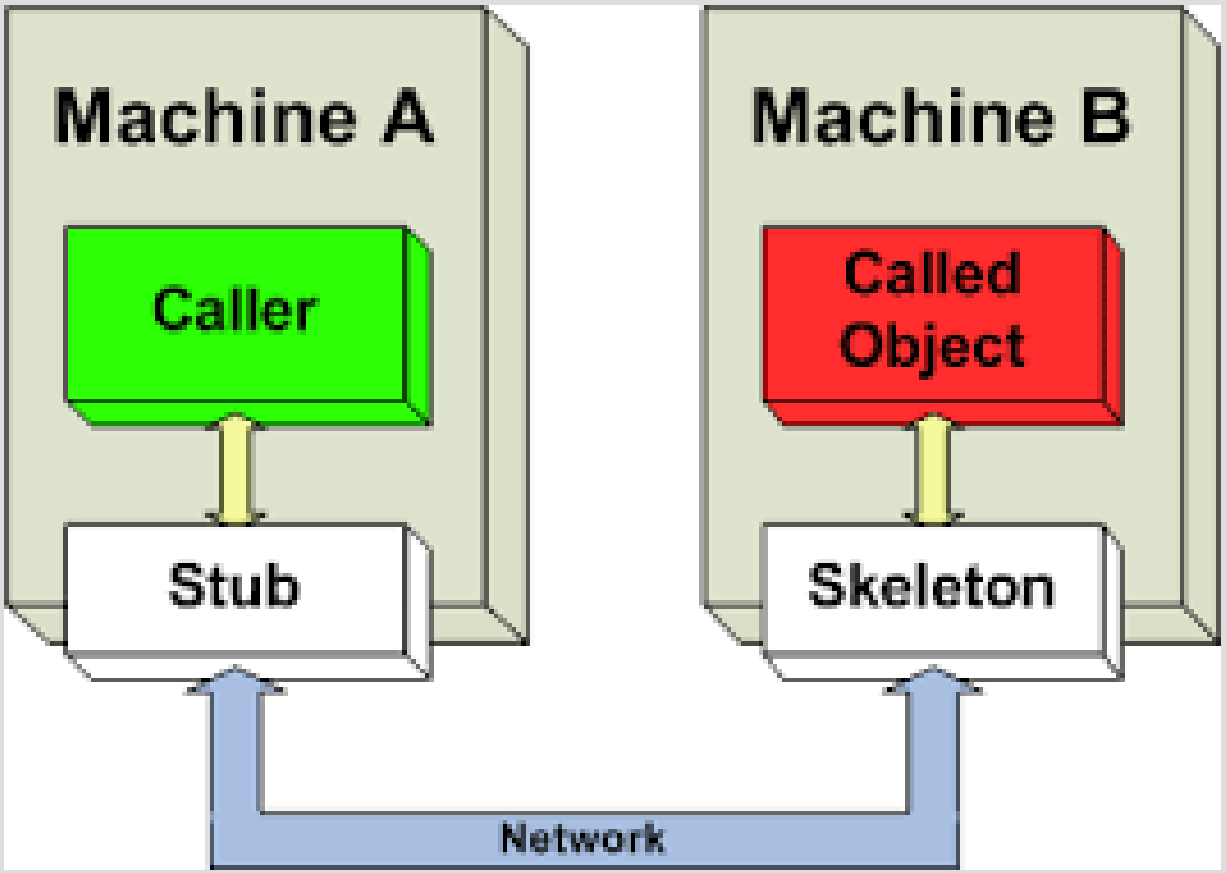


Talk outline

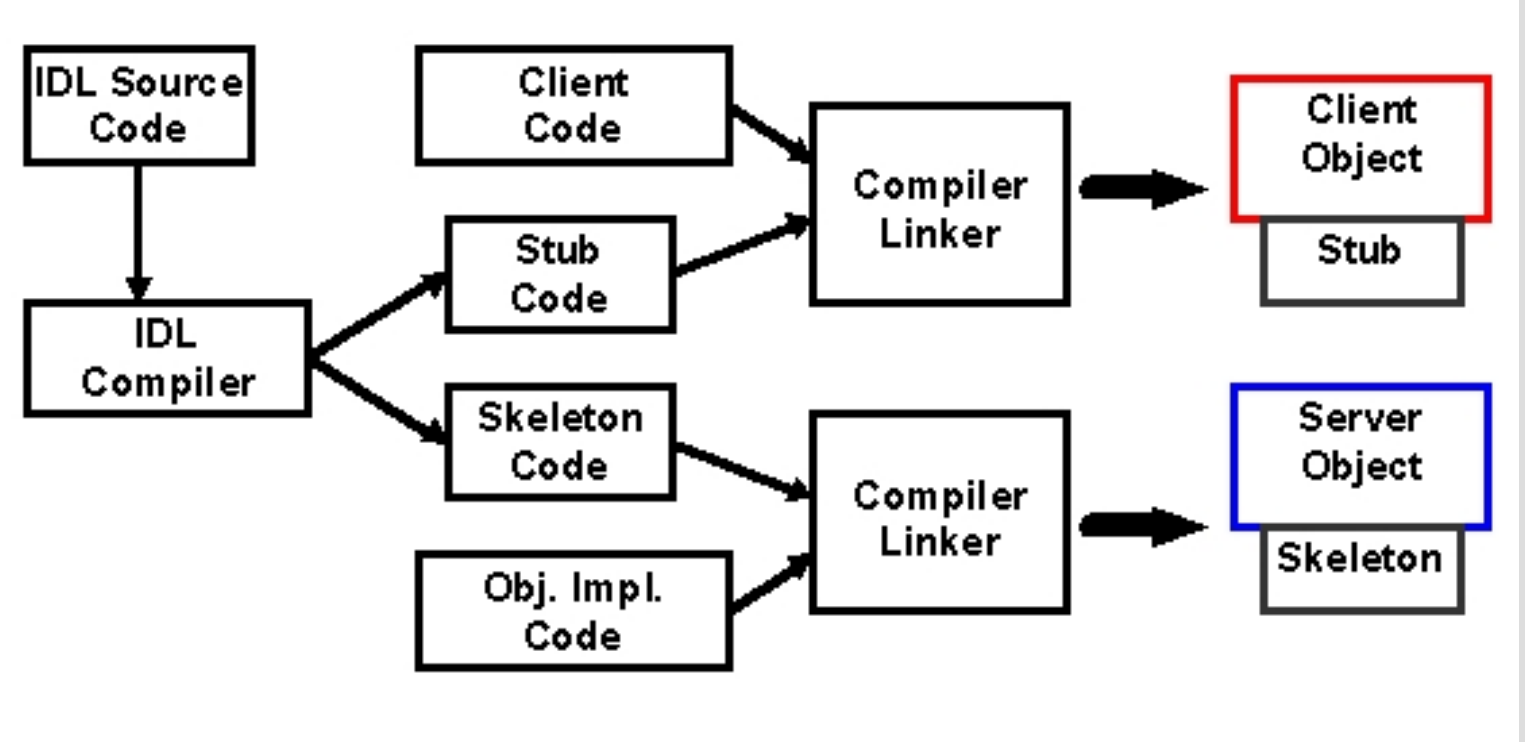
- Web Services
  - SOAP, WSDL (we will discuss WADL latter), UDDI
- Developing Web Services
- Testing Web Services
  - soapUI
- Consuming and invoking Web Services
- Advanced WS programming: DI and Spring



Stubs and skeletons



Stubs and skeletons (cont.)



Consuming Web Services: Axis 2 (http://axis.apache.org/axis2/java/core/)

```
import org.apache.axis2.SimpleServiceStub.EchoRequest;
import org.apache.axis2.SimpleServiceStub.EchoResponse;

public class TestClient {

    public static void main(String[] args) throws Exception {
        SimpleServiceStub stub = new SimpleServiceStub();

        //Create the request
        EchoRequest request = new SimpleServiceStub.EchoRequest();
        request.setParam0("Hello world");

        //Invoke the service
        EchoResponse response = stub.echo(request);

        System.out.println(response.get_return());
    }
}
```

```
WSDL2Java.sh -uri http://localhost:8080/axis2/services/SimpleService?wsdl
             -o /path/to/my/client/code/
```

Asynchronous Web Services

- So far, we have dealt with **synchronous** clients
- Many invocation frameworks support the notion of **asynchronous** Web Service call
- Recall the Notification WSDL operation pattern
- E.g., CXF supports:
  - *Polling approach*: The service returns a special response object (Future) that can be polled to check whether or not a response message has arrived
  - *Callback approach*: Developers call another special method that takes a reference to a callback object. The framework calls back on this object to pass it on the contents of the response message

Asynchronous Web Services: Java Futures

- From version 5.0 on, Java has support for special objects called *futures* provided by the *java.lang.concurrent* package

```
class interface ArchiveSearcher { String search(String target); }
public class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target) throws InterruptedException {
     Future<String> future = executor.submit(new Callable<String>() {
         public String call() { return searcher.search(target); }
     });
     displayOtherThings(); // do other things while searching
     try {  displayText(future.get()); // use future }
     catch (ExecutionException ex) { cleanup(); return; }
   }
 }
```

Asynchronous Web Services: Proxy generation

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" name="HelloWorld" ...>
   <wsdl:types>
      <schema targetNamespace="http://apache.org/hello_world_async_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
            elementFormDefault="qualified">
         <element name="greetMeSometime">
            <complexType>
               <sequence><element name="requestType" type="xsd:string"/></sequence>
            </complexType>
         </element>
         <element name="greetMeSometimeResponse">
            <complexType>
               <sequence><element name="responseType" type="xsd:string"/></sequence>
            </complexType>
         </element>
      </schema>
   </wsdl:types>
   <wsdl:message name="greetMeSometimeRequest">
      <wsdl:part name="in" element="x1:greetMeSometime"/>
   </wsdl:message>
   <wsdl:message name="greetMeSometimeResponse">
      <wsdl:part name="out" element="x1:greetMeSometimeResponse"/>
   </wsdl:message>
   <wsdl:portType name="GreeterAsync">
      <wsdl:operation name="greetMeSometime">
         <wsdl:input name="greetMeSometimeRequest" message="tns:greetMeSometimeRequest"/>
         <wsdl:output name="greetMeSometimeResponse" message="tns:greetMeSometimeResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   ...
</wsdl:definitions>
```

Asynchronous Web Services: Proxy generation (cont.)

After running wsdl2java with proper configuration, we obtain the following client-side interface code:

```
/* Generated by WSDLToJava Compiler. */
package org.apache.hello_world_async_soap_http;
...
import java.util.concurrent.Future;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
...
public interface GreeterAsync {

  public Future<?> greetMeSometimeAsync(
    java.lang.String requestType,
    AsyncHandler<org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse>
       asyncHandler);

  public Response<org.pache.hello_world_async_soap_http.types.GreetMeSometimeResponse>
                  greetMeSometimeAsync(java.lang.String requestType);

  public java.lang.String greetMeSometime(java.lang.String requestType);
}
```

Asynchronous Web Services: Polling-based invocation

- Non-blocking polling

```
Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;

if (greetMeSomeTimeResp.isDone()) {
  GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
}
```

- Blocking polling

```
Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;

GreetMeSometimeResponse reply = greetMeSomeTimeResp.get(60L,
  java.util.concurrent.TimeUnit.SECONDS);
```

Asynchronous Web Services: Callback-based invocation

```
package demo.hw.client;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
import
org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;

public class TestAsyncHandler
       implements AsyncHandler<GreetMeSometimeResponse> {

  private GreetMeSometimeResponse reply;

  public void handleResponse(Response<GreetMeSometimeResponse> response) {
    try {
      reply = response.get();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

}
```

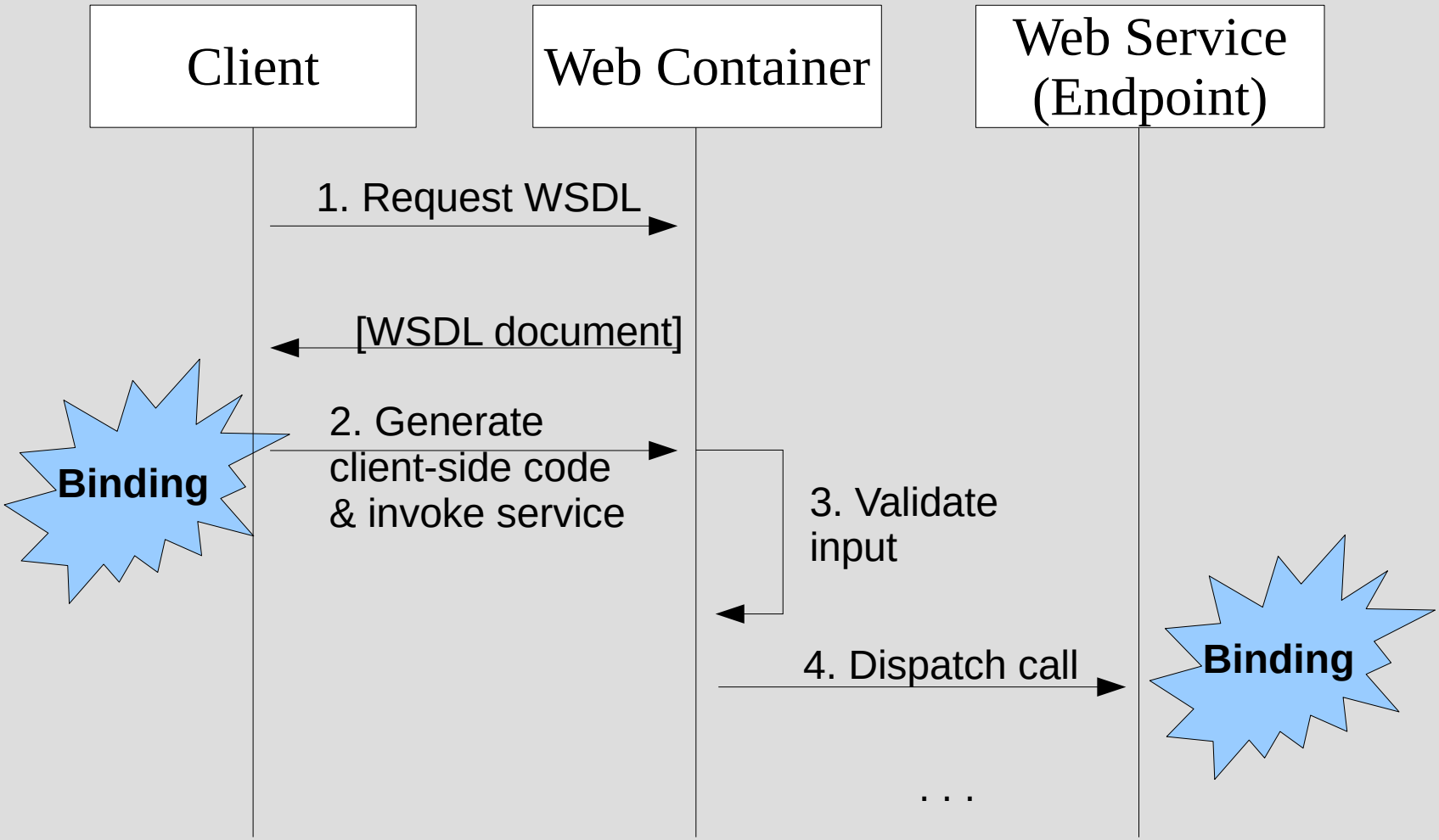
Our callback or handler just obtains and stores the result in a variable...

Asynchronous Web Services: Callback-based invocation (cont.)

```
public static void main(String args[]) throws Exception {
    ...
    TestAsyncHandler testAsyncHandler = new TestAsyncHandler();
    Future<?> response = port.greetMeSometimeAsync(
                                "Mr. X",
                                testAsyncHandler);
    while (!response.isDone()) {
      Thread.sleep(100);
    }
    resp = testAsyncHandler.getResponse();
    ...
    System.exit(0);
}
```

Note that results are only made accessible to the handler...

Consuming Web Services: XML data binding



JAXB versus DOM

- Two technologies to map XML to objects in memory
- Consider the following XML structure:

```
<customers>
  <customer>
    <name id=1>Customer 1</name>
    <age>30</age>
  </customer>
  <customer id=2>
    <name>Customer 2</name>
    <age>41</age>
  </customer>
  . . .
</customers>
```

JAXB versus DOM (cont.)

- DOM (data object model) solution:

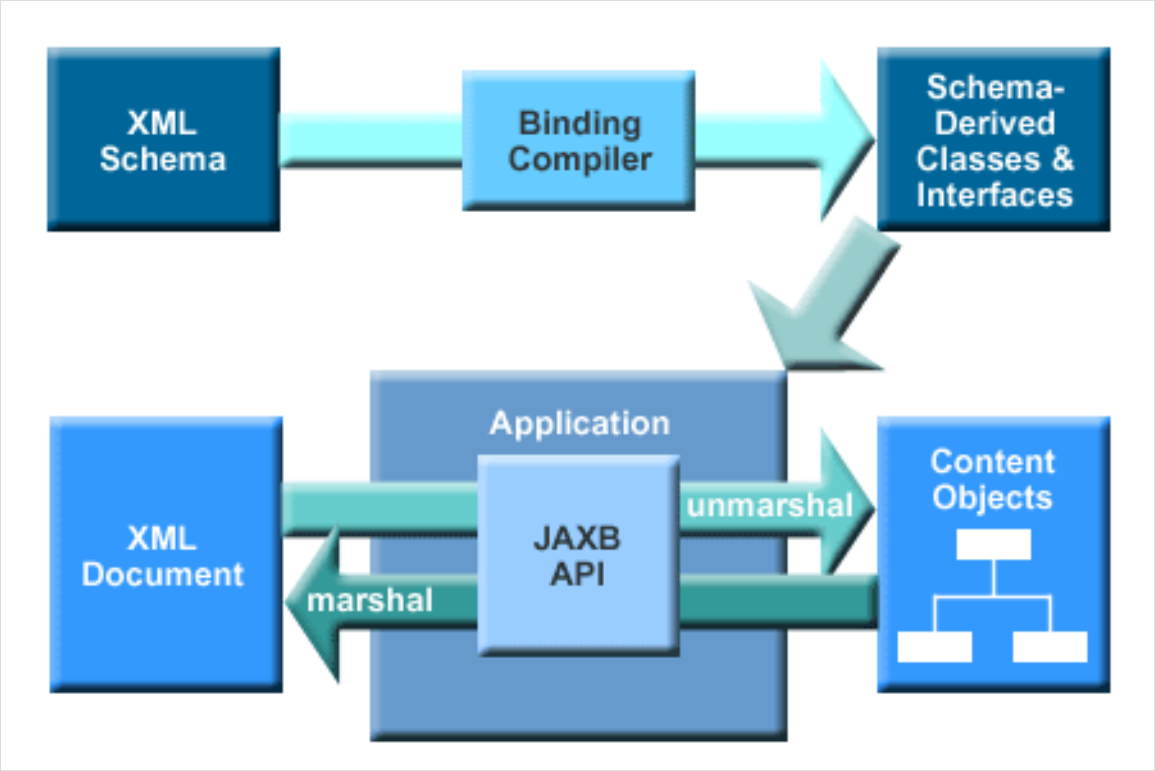
```
root.getFirstChild().getLastChild().getText()
```

Someone's age
(no type checking)

- JAXB (object binding model) solution:

```
doc.getCustomers[0].getAge()
```

JAXB: Overview



Using JAXB then requires:

1) Run a *binding compiler* on the schema file(s) to produce the appropriate Java class files
2) Compile the Java classes
3) Code is provided to *marshal/unmarshal* specific document instances
4) Optionally: Provide bindings

- Reference implementation: JAXBRI by Oracle
- Other alternatives exist! XMLBeans (oldest), JiBX (fastest), JDOM, Aegis (since XFire)...

JAXB: The binding compiler

```
<xsd:complexType name="AddressType">
  <xsd:sequence>
    <xsd:element name="Number" type="xs:unsignedInt"/>
    <xsd:element name="Street" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Compiling the schema

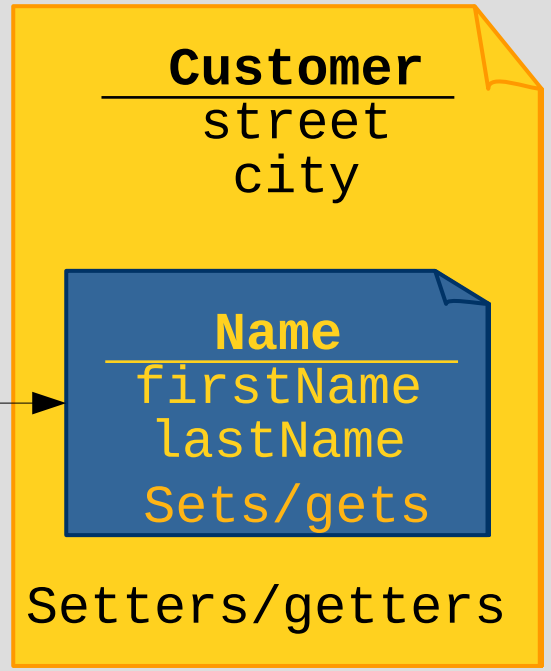
```
public interface AddressType {
    long getNumber();
    void setNumber(long value);

    String getStreet();
    void setStreet(String value);
}
```

Must be non-negative

Must be non-null

JAXB: Marshaling/Unmarshaling

```
<customer>
  <name>
    <first-name>Homer</first-name>
    <last-name>Simpson</last-name>
  </name>
  <street>Evergreen Ave.</street>
  <age>50</age>
  …
</customer>
```



- Complex type definitions are mapped to classes
- Child elements & attributes are mapped to fields
- May be able to adjust depth

JAXB: Limitations

```
<xsd:complexType name="ABType">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="A" type="xs:int"/>
      <xsd:element name="B" type="xs:string"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="B" type="xs:string"/>
      <xsd:element name="A" type="xs:int"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>
```

```
public interface ABType {
    int getA();    void setA(int value);
    String getB(); void setB(String value);
}
```

- Marshal(Unmarshal(xml)) ≠ xml (no roundtrip)
  (e.g. order is not always preserved; but usually roundtrip holds)
- @XmlType + propOrder

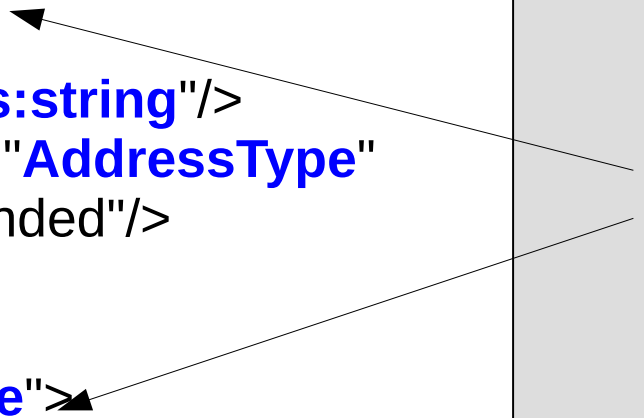
```
obj.setA(5);
obj.setB("a");
```

What happens when we marshal an object of this type?

JAXB: Example schema

Example.xsd

```
<xsd:element name="Person" type="PersonType"/>
<xsd:complexType name="PersonType">
  <xsd:sequence>
    <xsd:element name="Name" type="xs:string"/>
    <xsd:element name="Address" type="AddressType"
      minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="AddressType">
  <xsd:sequence>
    <xsd:element name="Number" type="xs:unsignedInt"/>
    <xsd:element name="Street" type="xs:string"/>
  </xsd:sequence>
</xsd:complexType>
```



JAXB: Generated Java interfaces

```
public interface AddressType {
    long getNumber();
    void setNumber(long value);

    String getStreet();
    void setStreet(String value);
}
```

Must be non-negative

Must be non-null

```
public interface PersonType {
    String getName();
    void setName(String value);

    /* List of AddressType */
    List<AddressType> getAddress();
}
```

Must be non-null

Must contain at least one item

JAXB: Marshaling/Unmarshaling (cont.)

XML  Java

```
<book>
  <title>Compilers</title>
  <author>Aho</author>
  <author>Sethi</author>
  <author>Ullman</author>
  <pages>796</pages>
</book>
```

```
ObjectFactory f =
      new ObjectFactory();
Book b = f.createBook();
b.setTitle("Compilers");
List a = b.getAuthor();
a.add("Aho");
a.add("Sethi");
a.add("Ullman");
b.setPages(796);
```

Nesting/flattening is specified through configuration

Programmatically using JAXB: Java API

- To load a specific Java data model, use:

```
JAXBContext context =
JAXBContext.newInstance("example");
```

Loads the "example" package, which is obtained through "*xjc -p example example.xsd*")

- To unmarshal an XML document, use:

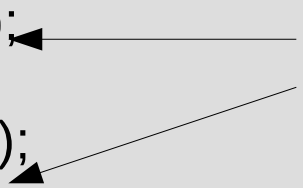
```
Unmarshaller unmarshaller = context.createUnmarshaller();
unmarshaller.setValidating(true);
PersonType person = (PersonType) unmarshaller.unmarshal(
                        new FileInputStream("example.xml") );
```

The root type

Programmatically using JAXB: Java API (cont.)

- You can access the loaded objects in the usual way:

```
// Read-only operations
System.out.println("Person name=" + person.getName() );
AddressType address = (AddressType)person.getAddress().get(0);
System.out.println("First Address: " + " Street=" + address.getStreet() +
                   " Number=" + address.getNumber() );

// Updates
person.setName("Homer J. Simpson");

List addressList = person.getAddress();
addressList.clear();

ObjectFactory objectFactory = new ObjectFactory();
AddressType newAddr = objectFactory.createAddressType();
anAddr.setStreet("Kwik-e-mart Ave.");
anAddr.setNumber(10000);
addressList.add(anAddr);
```



What if we validate at these two points?

Programmatically using JAXB: Java API (cont.)

- To validate objects on demand, use:

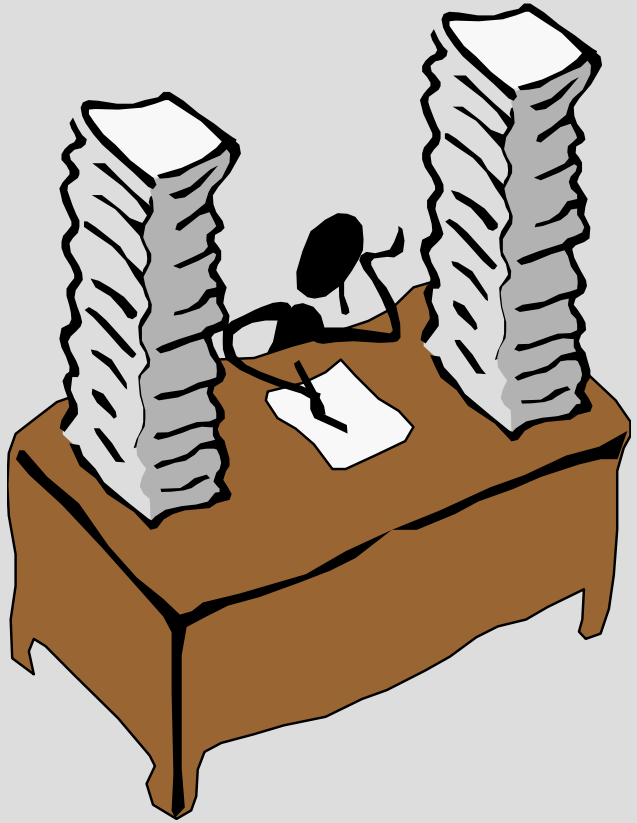
```
Validator validator = context.createValidator();
    // Validate restrictions on AddressType
validator.validate(anAddrObj);
    // Validate restrictions on PersonType
validator.validate(aPersonObj);
```

- To marshal objects, use:

```
Marshaller marshaller = context.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                                Boolean.TRUE);
marshaller.marshal(person, new FileOutputStream("example-output.xml"));
```

Talk outline

- Web Services
  - SOAP, WSDL (we will discuss WADL latter), UDDI
- Developing Web Services
- Testing Web Services
  - soapUI
- Consuming and invoking Web Services
- Advanced WS programming: DI and Spring



Spring (http://projects.spring.io/spring-framework)

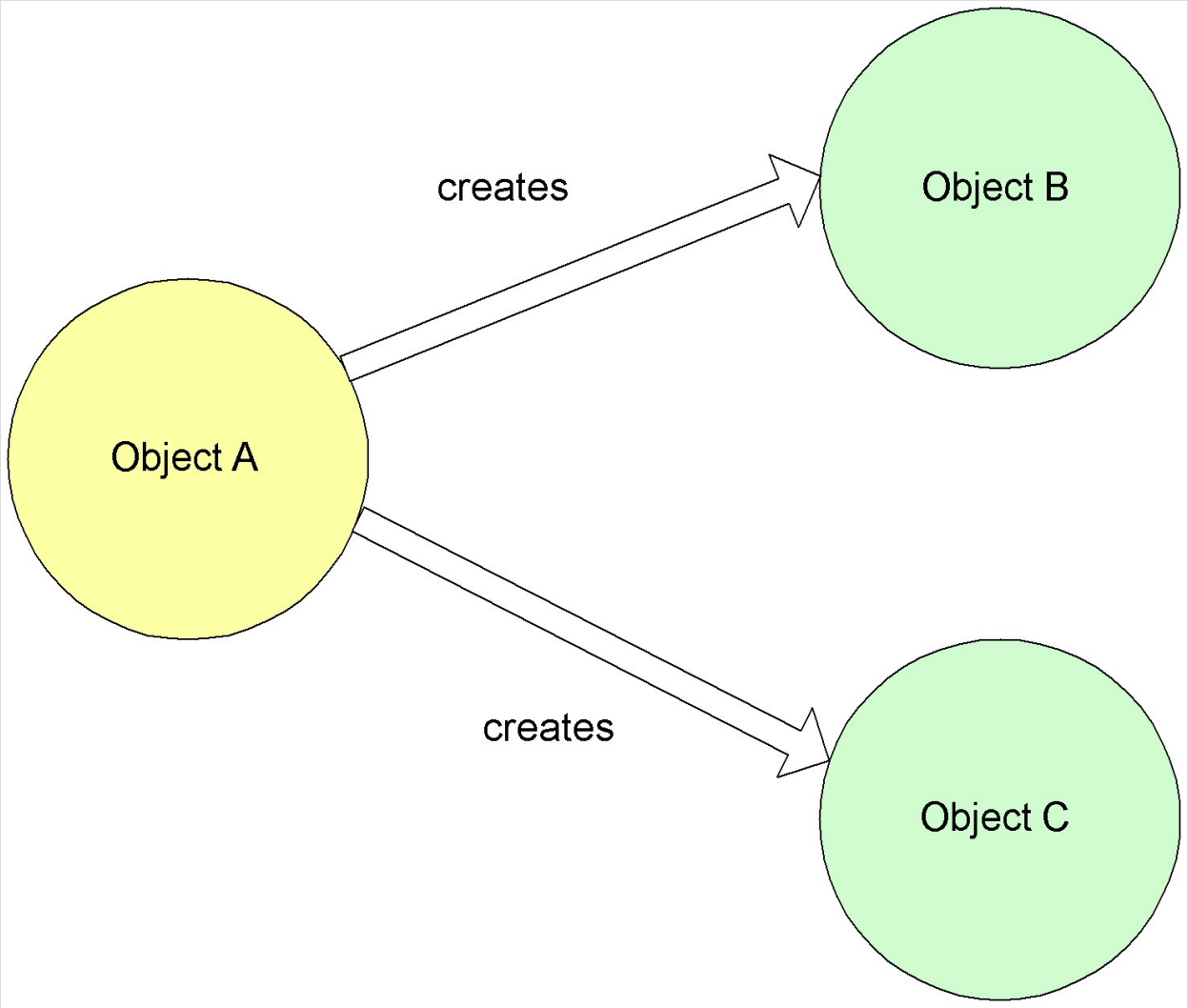
- Spring is a non-invasive and portable framework that allows you to introduce as much or as little API-related code as you want to your application
- Promotes decoupling and reusability
- Based on POJOs (Plain Old Java Objects)
- Most business objects in Spring applications do not depend on the Spring framework
- *Allows developers to focus more on reused* ***business logic*** *and less on* ***plumbing*** *problems*

Spring Web Services facilitates the use of Web Services by preserving the integrity of the application logic!

Spring: Inversion of control (IoC)

- Dependency injection (DI)
  - http://www.martinfowler.com/articles/injection.html
  - *Beans* define their dependencies through interfaces plus annotations (or XML files)
  - The container provides the injection at runtime
- Also known as the *Hollywood principle* - “don’t call us we will call you”
- Decouples object creators and locators from application logic
- Easy to maintain and reuse
- Testing is easier

Non-DI application: Object diagram

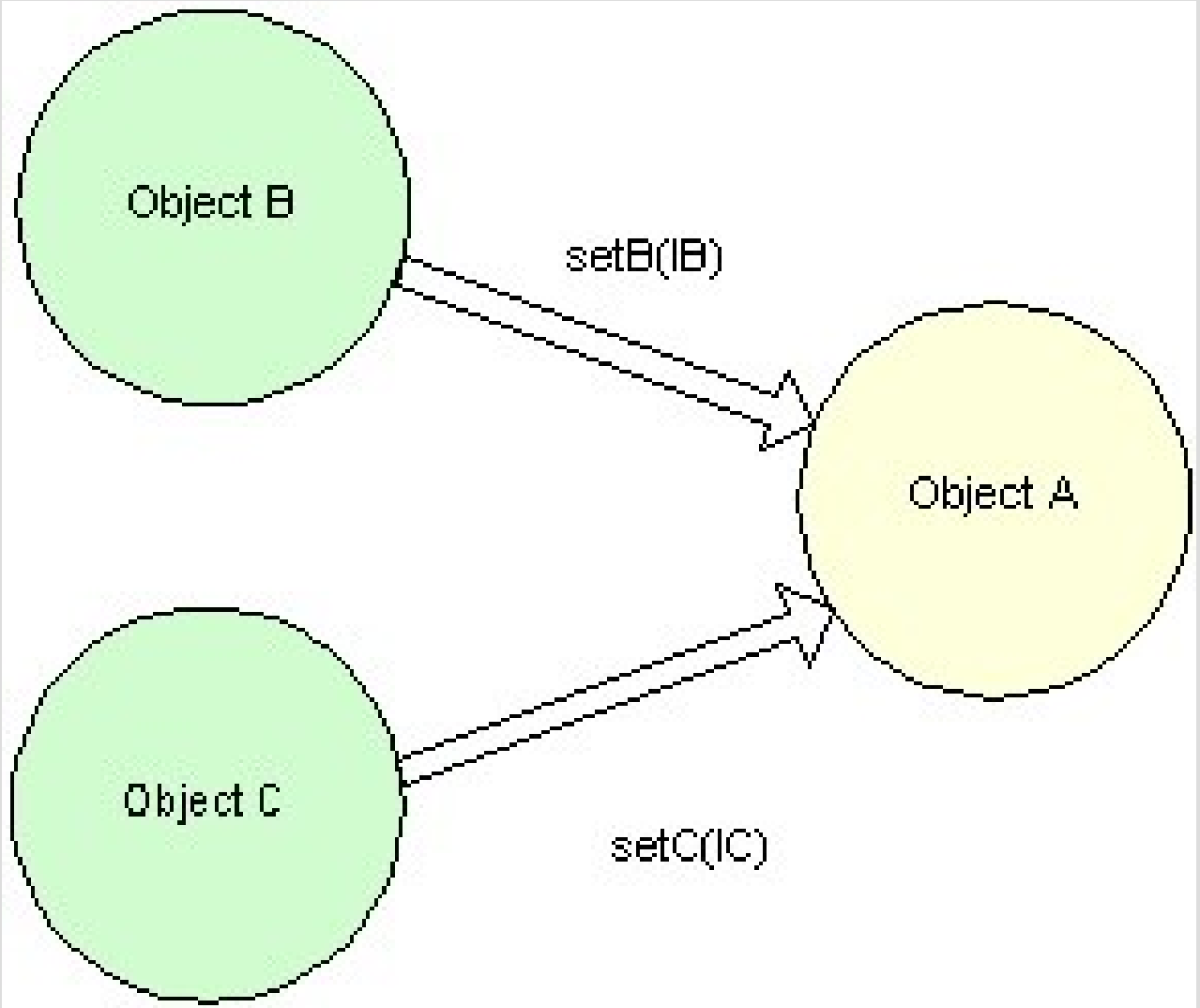


Non-DI object: Implementation

```
public class OrderServiceImpl implements IOrderService {
    public Order saveOrder(Order order) throws OrderException{
        try{
            // 1. Create a Session/Connection object
            // 2. Start a transaction
            // 3. Lookup and invoke one of the methods in a
            // DAO and pass the Session/Connection object.
            // 4. Commit transaction
        } catch(Exception e){
            // handle e, rollback transaction, //cleanup, // throw e
        } finally{
            //Release resources and handle more exceptions
    }
}
```

Two many code handling non-functional concerns!

DI-based application: Object diagram



DI-based application: Implementation

```
public class OrderSpringService implements IOrderService {
    IOrderDAO orderDAO;

    public Order saveOrder(Order order) throws OrderException{

        // perform some business logic…
        return orderDAO.saveNewOrder(order);
    }

    @Autowired
    public void setOrderDAO(IOrderDAO orderDAO) {
        this.orderDAO = orderDAO;
    }
}
```

The code looks better now...

DI-based application: More on autowiring

```
public class OrderSpringService implements IOrderService {
      @Autowired
      IOrderDAO orderDAO;

       public Order saveOrder(Order order) throws OrderException{
            ...
       }
}
```

Or alternatively:

```
public class OrderSpringService implements IOrderService {
      IOrderDAO orderDAO;

      @Autowired
       public OrderSpringService(IOrderDAO orderDAO) {
          this.orderDAO = orderDAO;
      }

       public Order saveOrder(Order order) throws OrderException{
          …
       }
}
```

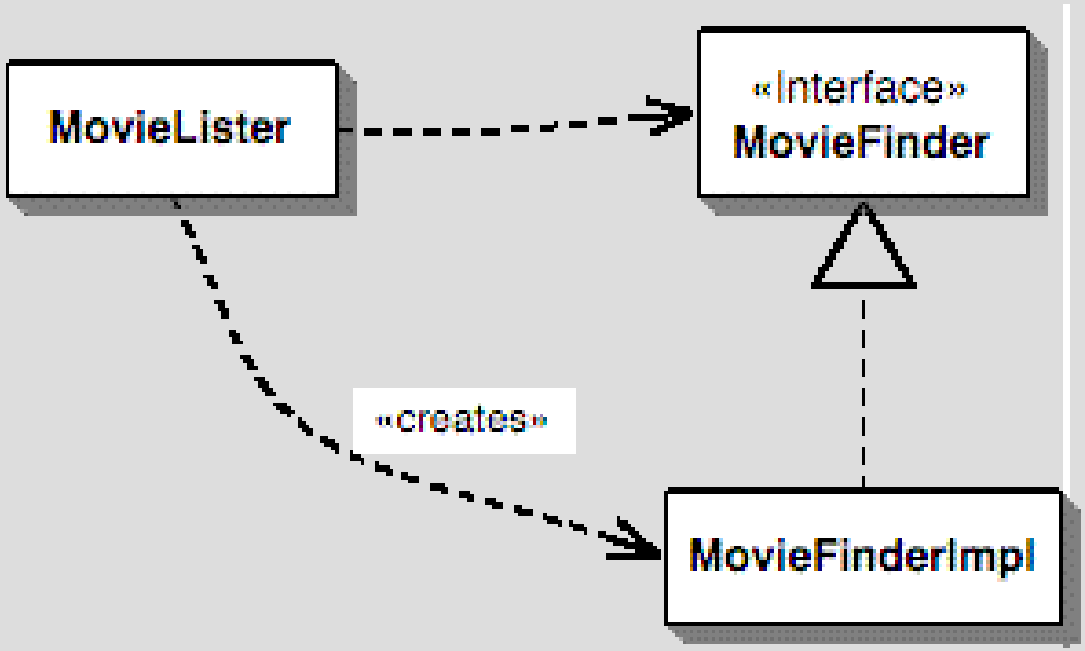
DI: A naïve example

```
class MovieLister {
    public Movie[] moviesDirectedBy(String arg) {
        List allMovies = finder.findAll();
        for (Iterator it = allMovies.iterator(); it.hasNext();) {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg))
                it.remove();
        }
        return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
    }
}

public interface MovieFinder {
    List findAll();
}
```

DI: A naïve example (cont.)

This is well-decoupled,
but...

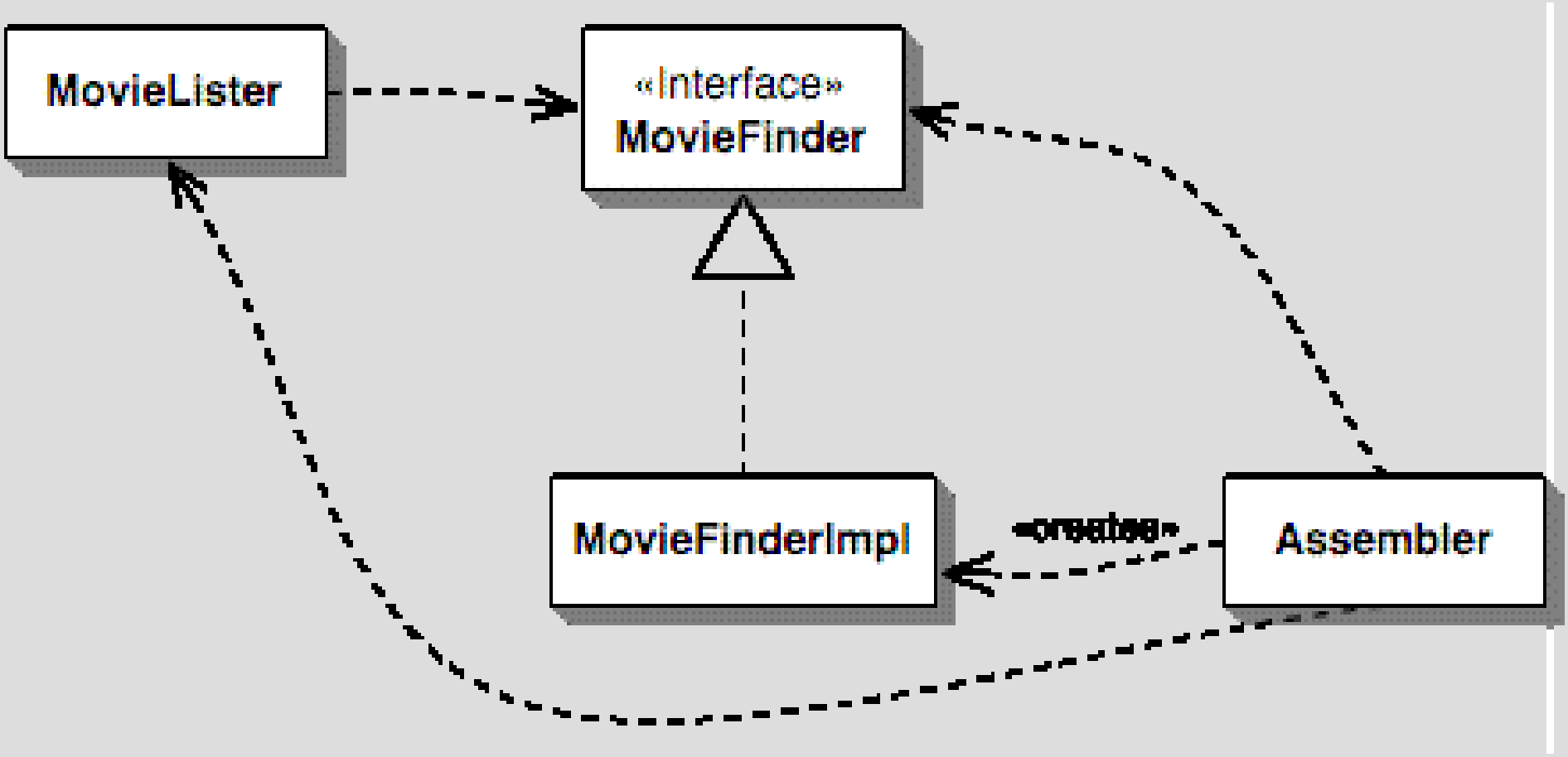
```
class MovieLister{
  private MovieFinder finder;
  public MovieLister() {
    finder = new CSVDelimitedMovieFinder("movies1.txt");
}
```



DI: A naïve example (cont.)

- The **MovieLister** class is dependent on both the **MovieFinder** interface and upon the implementation
- We would prefer it if it were only dependent on the interface:
  - but then how do we built an instance to work with?
  - we want **MovieLister** to work with any implementation
- Expanding this into a real system, we might have dozens of such cases

DI: A naïve example (cont.)



Finder injection

```
class MovieLister...
  private MovieFinder finder;
  @Autowired
  public void setFinder(MovieFinder finder) {
    this.finder = finder;
  }
   public Movie[] moviesDirectedBy(String arg) { … }
}

class CSVMovieFinder...
    public void setFilename(String filename) {
        this.filename = filename;
    }
}
```

Finder injection (cont.)

How do you tell Spring to bootstrap your bean-based app?

```
@Component("movieLister")
public class MovieLister {
    ...
}

@Component("csvMovieFinder")
public class CSVMovieFinder implements MovieFinder {
    ...
}
```

Finder injection: Autowiring disambiguation

Sometimes we have to tell Spring which bean to inject...

```
@Component("csvMovieFinder")
public class CSVMovieFinder implements MovieFinder { … }

@Component("sqlMovieFinder")
public class SQLMovieFinder implements MovieFinder { … }

@Component("movieLister")
public class MovieLister {
   @Autowired
   @Qualifier("csvMovieFinder")
   private MovieFinder finder;
   ...
}
```

Spring: Benefits for service-oriented computing

- You are not forced to import or extend any APIs when accessing services
- An **invasive** API takes over your code
  - Axis2 requires to use specific API classes
  - An example beyond SOC: Concurrent programming
- Invasive frameworks are more difficult to test
- Apps. can be build by gluing together a bunch of POJOs plus API components representing Web Services

Spring: Creating a SOAP Web Service XSD definition

```
<xs:schema ...>
    <xs:element name="getCountryRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="getCountryResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="country" type="tns:country"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="country">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="population" type="xs:int"/>
            <xs:element name="capital" type="xs:string"/>
            <xs:element name="currency" type="tns:currency"/>
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="currency">
        <xs:restriction base="xs:string">
            <xs:enumeration value="GBP"/>
            <xs:enumeration value="EUR"/>
            <xs:enumeration value="PLN"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Then, generate domain classes based on this XML schema: The right approach is do this automatically during build time using a maven or gradle plugin (**xjc**)

Spring: Creating a SOAP Web Service
Dummy repository

```
...

@Component
public class CountryRepository {
        private static final Map<String, Country> countries = new HashMap<>();

        @PostConstruct // What to do after DI and object init itself
        public void initData() {
                Country spain = new Country();
                spain.setName("Spain");
                spain.setCapital("Madrid");
                spain.setCurrency(Currency.EUR);
                spain.setPopulation(46704314);
                countries.put(spain.getName(), spain);

               Country uk = new Country();
                uk.setName("United Kingdom");
                uk.setCapital("London");
                uk.setCurrency(Currency.GBP);
                uk.setPopulation(63705000);
                countries.put(uk.getName(), uk);

                ...
        }
        public Country findCountry(String name) {
                Assert.notNull(name, "The country's name must not be null");
                return countries.get(name);
        }
}
```

Spring: Creating a SOAP Web Service
Creating the endpoint

```
...
import io.spring.guides.gs_producing_web_service.GetCountryRequest;
import io.spring.guides.gs_producing_web_service.GetCountryResponse;
```


```
@Endpoint
public class CountryEndpoint {
     private static final String NAMESPACE_URI = ...
     private CountryRepository countryRepository;
     @Autowired
     public CountryEndpoint(CountryRepository countryRepository) {
         this.countryRepository = countryRepository;
     }
     @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCountryRequest")
     @ResponsePayload
     public GetCountryResponse getCountry(@RequestPayload GetCountryRequest r) {
         GetCountryResponse response = new GetCountryResponse();
         response.setCountry(countryRepository.findCountry(r.getName()));
         return response;
     }
}
```

- *Endpoint* means using a special Servlet (MessageDispatcherServlet) + application context
- WSDL is automatically constructed upon deployment

Spring remoting: Interceptors

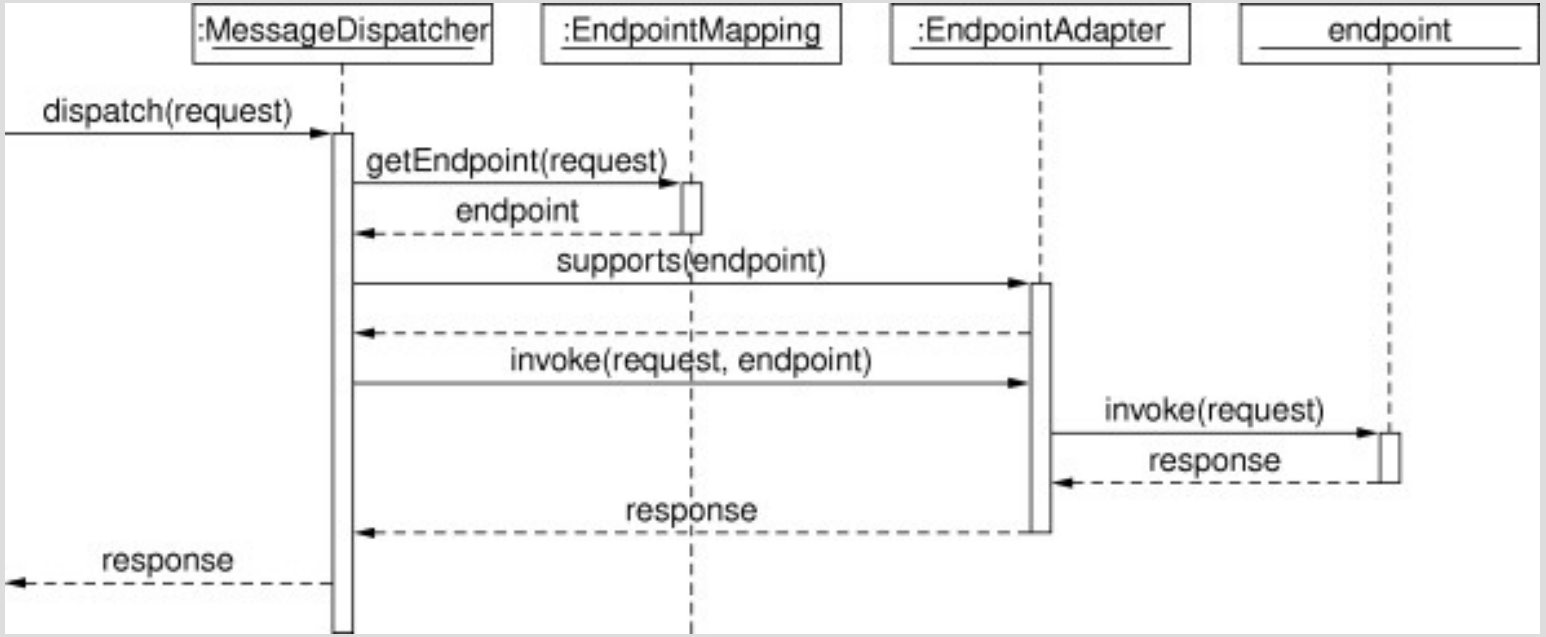
- Too simple right? But you can work closer to the wire...
- Spring provides **interceptors**
  - Non-invasive endpoint invocation chains
  - Useful for debugging, logging, security
  - Around 15 built-in interceptors

```
public class MyInterceptor implements EndpointInterceptor {
   public boolean handleRequest(MessageContext context, Object endpoint) {
      WebServiceMessage webServiceMessageRequest = context.getRequest();
      SoapMessage soapMessage = (SoapMessage) webServiceMessageRequest;
       try {
            SoapHeader soapHeader = soapMessage.getSoapHeader()
            // ...
       } catch (Exception e) {
           return true;
       }
       // Continue to the next interceptor/endpoint
       return true;
   }
}
```

You need to be familiarized with the SOAP API...

Spring remoting: Call chain

- An appropriate endpoint is searched; if found, preprocessors, postprocessors, and endpoints are executed
- An appropriate adapter is searched for the endpoint
- If no response is sent (e.g. due to block in the interceptor chain, or exceptions in the sequence) **exception resolvers** take over



- The MessageDispatcher has several properties for setting endpoint adapters, mappings, exception resolvers (no need by default)

Spring vs Spring *

- *Spring*
  - Provides a kernel for doing DI and IOC
  - Builds on these to provide modules (Spring JDBC, Spring MVC, Spring ORM, Spring JMS, Spring Test)
  - Focus on reducing boilerplate code (simple abstractions) and plumbing code; better unit testing; framework
  - Integration with third-party frameworks (e.g., Hibernate, iBATIS)
- *Spring MVC*
  - Abstractions (framework) to simplify Web application development
- *Spring Boot*
  - Auto-configuration through **starters** (dependency descriptors)
  - E.g. spring-boot-starter-web-services: SOAP Web Services
- *Spring Cloud*
  - Distributed chores (e.g., integration with AWS, load balancing, service discovery)
  - Next year! (Computación Paralela y Distribuida)

Questions?