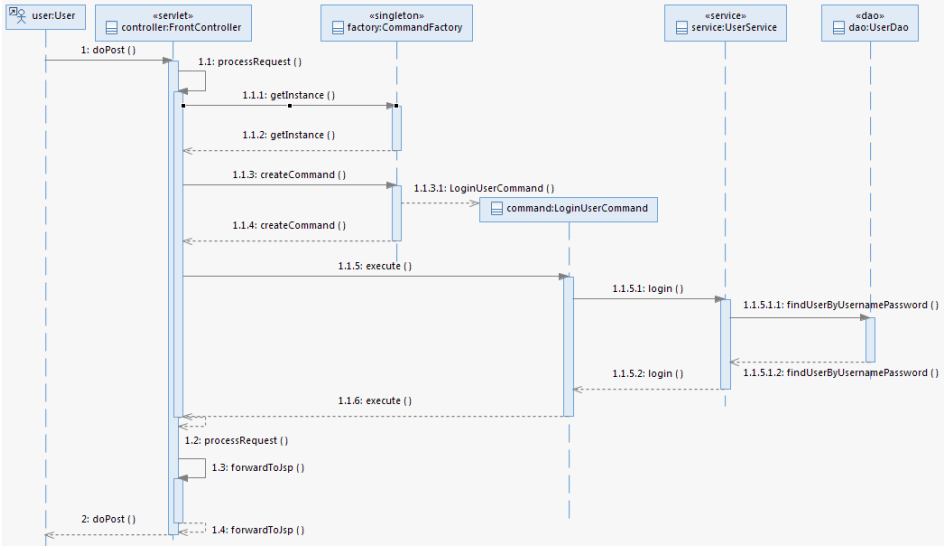
Software Engineering III

Example Patterns

Context Object Pattern

Application Example



Context Object Pattern

- **Intent:** Avoid using protocol-specific system information outside of its relevant context.
- The Command classes we used in our Web Application were **protocol dependent**.

```
public interface Command {
  String execute(HttpServletRequest request,
                 HttpServletResponse response);
}
```

Context Object Pattern

- This makes it **difficult to test** these classes outside of their context or to re-use them in another context.
- The **Context Object** pattern may be used to write these classes in a **protocol-independent** way.
- The simplest implementation of **Context Object** in this case involves using a Map (e.g. a HashMap) for storing the relevant data
- This can make it **easy to test** these new Command classes.

Context Object Pattern

```
package command ;
import java.util.Map;

public interface Command {
    String execute(Map<String, Object> map) ;
}
```

HttpServletRequest is replaced by Map

Examine the Map interface in the Java documentation.

Context Object Pattern

```
public class ListUsersCommand implements Command {

    public String execute(Map<String, Object> myMap) {

      String page = null;
      try {
            UserService userService = new UserService();
            List<Users> users = userService.ListUsers();
            if (users == null || users.isEmpty() ) {
                  myMap.put("message", "No users in list");
                  page = "message.jsp";
            } else {
                  myMap.put("users", users);
                  page = "viewUsers.jsp";
            }
    } catch (UserServiceException e) {
      myMap.put("message", "message.jsp");
      page = "message.jsp";
    }
    return page;
  }
}
```

HttpServletRequest is replaced by Map

request.setAttribute(/* etc. */)

is replaced by

myMap.put(/* etc. */)

Context Object Pattern

- As the *ListUsersCommand* class is ***no longer protocol-dependent***, it is a simple matter to test the class without starting the web server (e.g. Tomcat).

Context Object Pattern

```
public static void testListUsers() {

    Command command = new ListUsersCommand() ;
    Map<String, Object> myMap = new HashMap<String, Object>() ;

    String page = command.execute(myMap) ;
    System.out.println("page = " + page) ;

    List<User> users = (List<User>) myMap.get(" users ") ;
    if (users == null || users.isEmpty())
       System.out.println("No users ") ;
    else {
       for (User u : users) {
          System.out.println(u);
       }
    }

    String message = (String) myMap.get("message");
    System.out.println("message = " + message);
  }
```



```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    Map<String, Object> myMap = new HashMap<String, Object>() ;
    Map<String, String[]> parameterMap = request.getParameterMap();
    myMap.put("parameterMap", parameterMap);
    request.setAttribute("myMap", myMap);   // Now web pages can access the map through request
            ....
            ....
    HttpSession session = request.getSession();

    String commandStr = request.getParameter("action");
    try {
       CommandFactory factory = CommandFactory.getInstance();
       Command command = factory.createCommand(commandStr);
       page = command.execute(myMap);
       status = (String) myMap.get("status");
       if (status != null) {      // status has changed, e.g. in LoginCommand, so set in session
          session.setAttribute("status", status);
       }
    } catch (CommandCreationException e) {
       myMap.put("message", "Error: " + e.getMessage());
       page = "message.jsp";
    }
    gotoPage(page, request, response);
}
```

See note on next slide concerning this method

getParameterMap()

- This method returns a *java.util.Map* of the parameters of this request.
- Request parameters are extra information sent with the request.
- For HTTP servlets, parameters are contained in the query string or posted form data.

- Note: For the map returned from the method getParameterMap()

  the ***key*** is of type String
  and the ***value*** is of type String[].

'Wrapper' Patterns

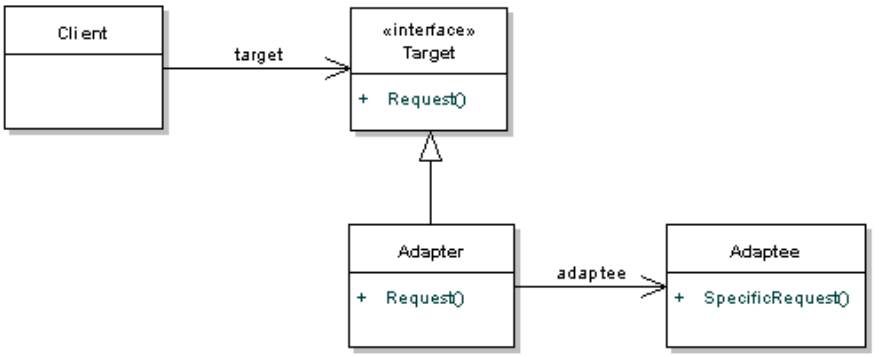
Wrapper Patterns

- Adapter
- Decorator
- Proxy
- Bridge

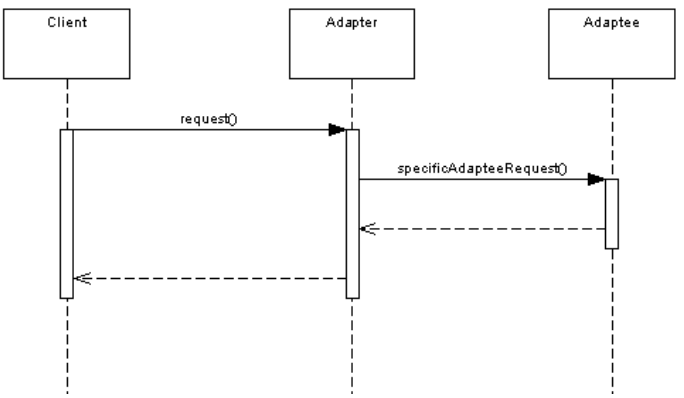
Adapter

- Intent
  - Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Provide an Adapter class which wraps the class that will ultimately perform the requested logic. The Adapter class provides a different interface to that provided by the wrapped class.

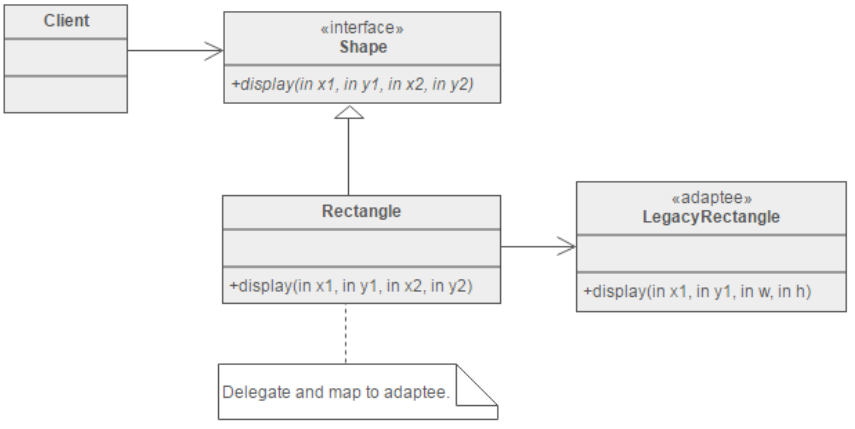
Adapter



Adapter



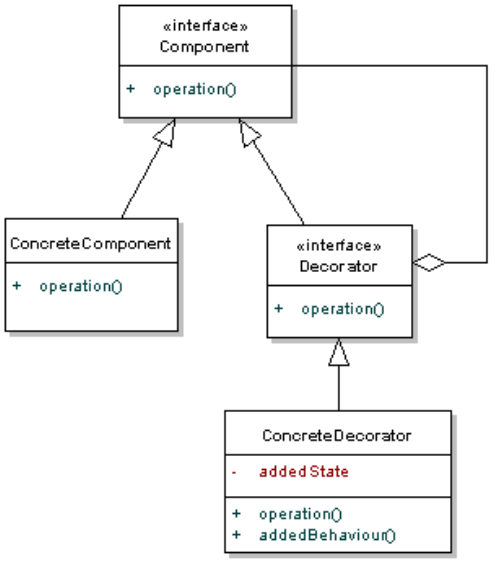
Adapter Example



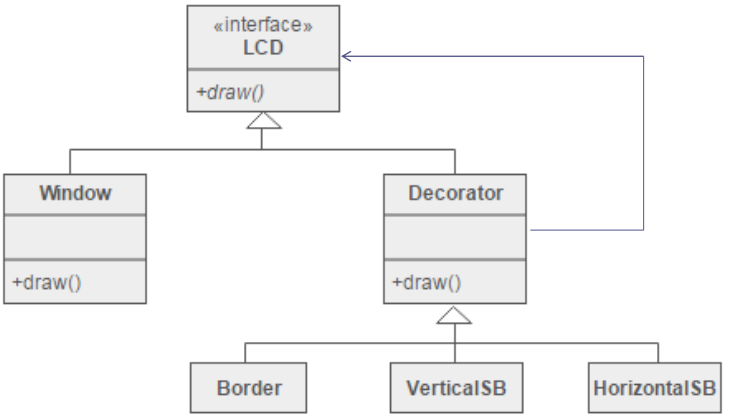
Decorator

- Intent
  - add additional responsibilities dynamically to an object
- Used when you want to add functionality to an object, but not by extending that object's type
- Note, with the *Adapter* pattern, the intent was not add additional responsibility (decorations) as it is here but just to provide a different interface

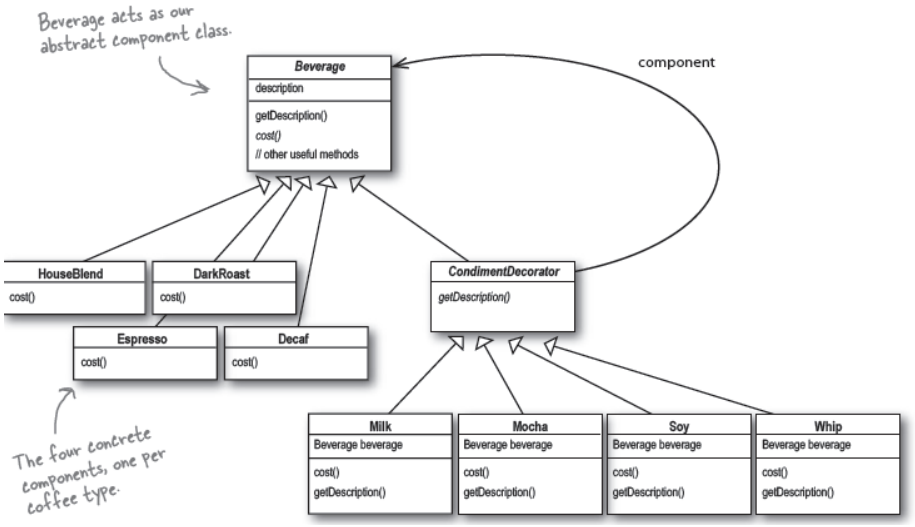
Decorator



Decorator



Decorator Example

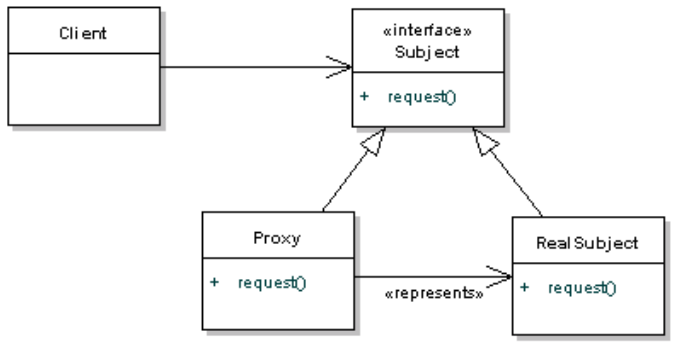


Proxy

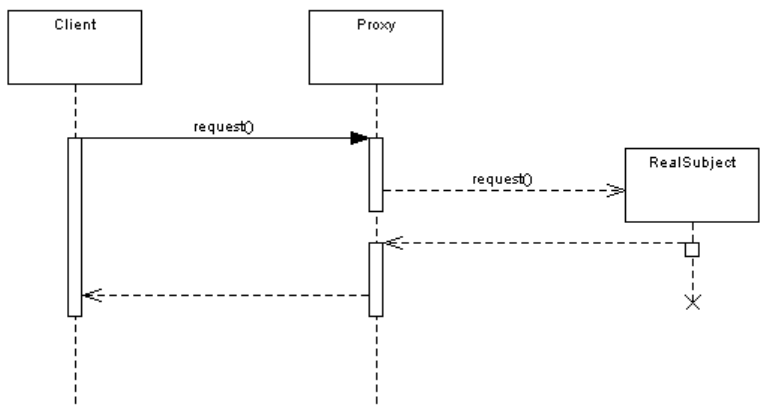
- Intent
  - Provide a *Placeholder* for an object to control references to it.

- lazy-instantiate an object
- hide call to a remote service
- control access to an object

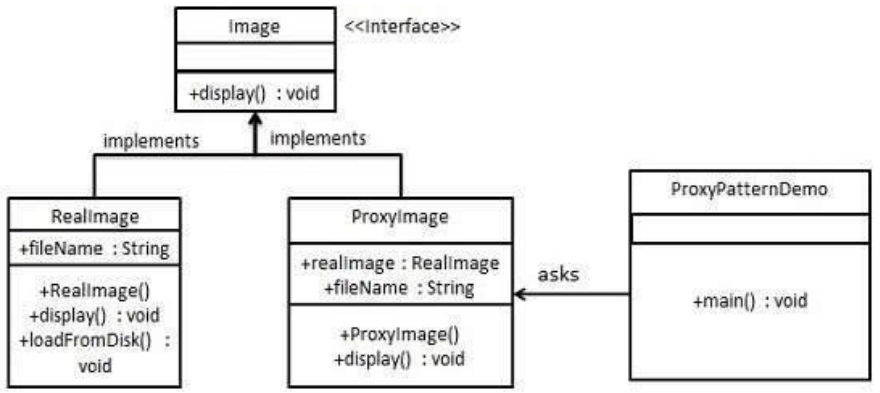
Proxy


Client
«interface»
Subject
+ request()
Proxy
+ request()
«represents»
Real Subject
+ request()

Proxy


Client
Proxy
request()
request()
RealSubject

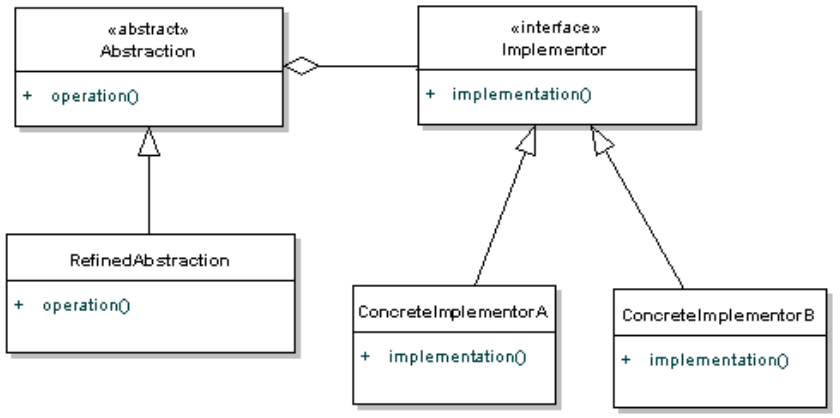
Proxy Example



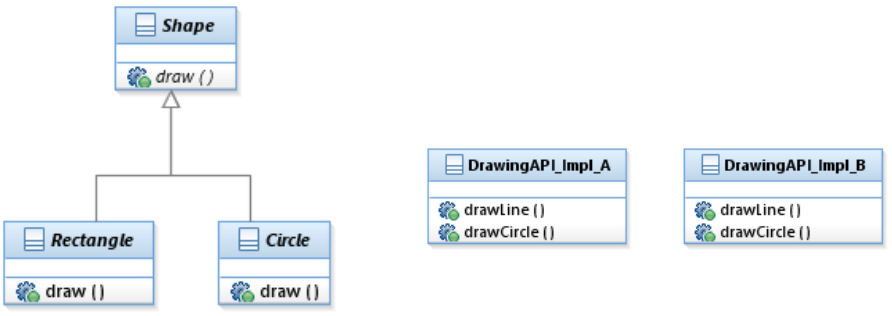
Bridge

- Intent
  - Decouple abstraction from implementation so that the two can vary independently
- Define both the abstract interface and the underlying implementation – can swap out different implementations

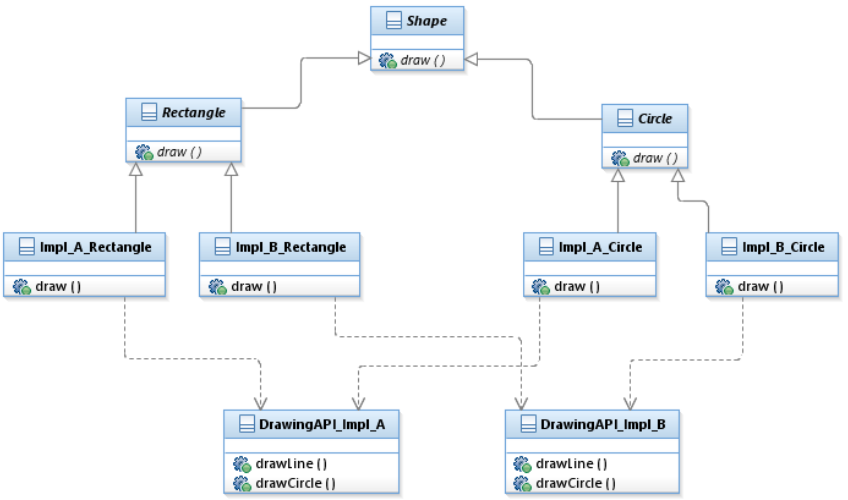
Bridge



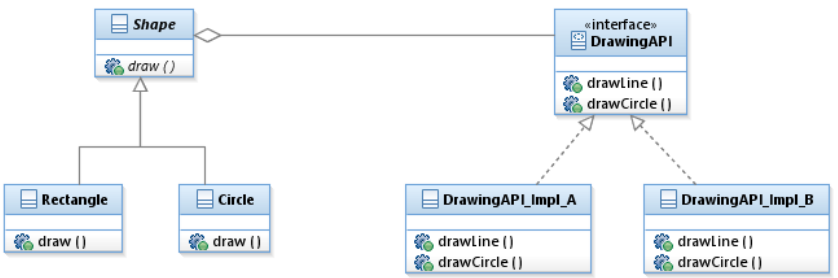
Bridge Example



Bridge Example – using Inheritance



Bridge Example – using Aggregation



REST

Architectural Style or Design Pattern

?

REST

Representational **S**tate **T**ransfer

- REST is an architectural style which is based on web-standards and the HTTP protocol.
- REST was first described by Roy Fielding in 2000.
- Seen as an alternative to SOAP

REST

- In a REST based architecture **everything is a resource**. A resource is accessed via a common interface **based on the HTTP** standard methods.
- In an REST architecture you typically have a **REST server** which provides access to the resources and a **REST client** which accesses and modify the REST resources. Every resource should support the HTTP common operations. **Resources are identified by global ID's**.
- REST allows that **resources have different representations**, e.g. **text, xml, json etc**. The rest client can ask for specific representation via the HTTP protocol (Content Negotiation).

HTTP Methods

- The HTTP standards methods which are typically used in REST are PUT, GET, POST, DELETE.
- **GET** defines a reading access of the resource without side-effects. The resource is never changed via a GET request, e.g. the request has no side effects.
- **PUT** creates a new resource.
- **DELETE** removes the resources.
- **POST** updates an existing resource or creates a new resource.

Resources

- Data *and* functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs)
- The resources are acted upon by using a set of simple, well-defined operations (the HTTP methods)
- E.g. GET
  - http://localhost:8080/RestExample/rest/hello
- E.g. POST
  - http://localhost:8080/RestExample/rest/hello

JAX-RS

- JAX-RS - The Java API for RESTful Web Services
- JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.
- *Jersey* (http://jersey.java.net/) is a reference implementation of the JAX-RS specification (JSR 311).

Implementing a REST Web Service

- All incoming requests from clients are received by a *servlet*.
- The *servlet* analyses the request and **selects the correct class and method to execute** based on the request.
- How does it know what class/method to execute?

Java Annotations

- Within our server side code, we use annotations to specify what code should be executed for specific client requests.

JAX-RS Hello World Example – No Annotation

```
public class Hello {

    public String sayPlainTextHello() {
            return "Hello Jersey";
    }

    public String sayXMLHello() {
            return "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";
    }

    public String sayHtmlHello() {
            return "<html> " + "<title>" + "Hello Jersey" + "</title>"
                                    + "<body><h1>" + "Hello Jersey" + "</body></h1>" +
    "</html> ";
    }

}
```


JAX-RS Hello World Example - Annotated

```
@Path("/hello")
public class Hello {

    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
            return "Hello Jersey";
    }

    // This method is called if XML is request
    @GET
    @Produces(MediaType.TEXT_XML)
    public String sayXMLHello() {
            return "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";
    }

    // This method is called if HTML is request
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String sayHtmlHello() {
            return "<html> " + "<title>" + "Hello Jersey" + "</title>"
                                    + "<body><h1>" + "Hello Jersey" + "</body></h1>" + "</html> ";
    }
}
```

Note:
POJO, no interface no extends

The class registers its methods for the HTTP GET request using the *@GET* annotation.

Using the *@Produces* annotation, it defines that it can deliver several MIME types, text, XML and HTML.

The browser requests per default the HTML MIME type.

JAX-RS annotations are runtime annotations

Web Service *web.xml Configuration File*

```
<display-name>RestExample</display-name>
 <servlet>
   <servlet-name>Jersey REST Service</servlet-name>
   <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>com.restexample</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
   <servlet-name>Jersey REST Service</servlet-name>
   <url-pattern>/rest/*</url-pattern>
 </servlet-mapping>
```

The Jersey servlet that will handle incoming client requests

Code packages where our web service code resides

The urls that will be handled by the Jersey servlet

REST Client – Browser (GET request)

http://localhost:8080/RestExample/rest/hello

- This name is derived from the "display-name" defined in the "web.xml" file, augmented with the servlet-mapping url-pattern and the "hello" @Path annotation from your class file. You should get the message

"Hello Jersey"

REST Client - Java

```
public static void main(String[] args) {

        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create(config);
        WebResource service = client.resource(getBaseURI());

        // Get plain text
        System.out.println(service.path("rest“).path("hello")
                                                .accept(MediaType.TEXT_PLAIN)
                                                .get(String.class));

        // Get XML
        System.out.println(service.path("rest“).path("hello")
                                                .accept(MediaType.TEXT_XML)
                                                .get(String.class));

        // The HTML
        System.out.println(service.path("rest“).path("hello")
                                                .accept(MediaType.TEXT_HTML)
                                                .get(String.class));
}
```


Receiving a POST request

```
@POST
@Produces(MediaType.TEXT_HTML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void postHello(  @FormParam("message") String message,
                        @Context HttpServletResponse servletResponse
                      ) throws IOException {

   System.out.println("*******************************");
   System.out.println("Incoming parameter is: " + message);
   System.out.println("This is where you can write your POST code...");
   System.out.println("*******************************");
}
```

Sending a POST request - HTML

```
<!DOCTYPE html>
<html>
 <head>
    <title>Form to POST a resource</title>
 </head>
<body>

   <form action="RestExample/rest/hello" method="POST">
          <input name="message" />
          <input type="submit" value="Submit" />
   </form>

</body>
</html>
```


More Annotations

@PATH(your_path)	Sets the path to base URL + /your_path. The base URL is based on your application name, the servlet and the URL pattern from the web.xml" configuration file.
@POST	Indicates that the following method will answer to a HTTP POST request
@GET	Indicates that the following method will answer to a HTTP GET request
@PUT	Indicates that the following method will answer to a HTTP PUT request
@DELETE	Indicates that the following method will answer to a HTTP DELETE request
@Produces(MediaType.TEXT_PLAIN [, more-types])	@Produces defines which MIME type is delivered by a method annotated with @GET. In the example text ("text/plain") is produced. Other examples would be "application/xml" or "application/json".
@Consumes(type [, more-types])	@Consumes defines which MIME type is consumed by this method.
@PathParam	Used to inject values from the URL into a method parameter. This way you inject for example the ID of a resource into the method to get the correct object.

Passing Parameters – in true REST style

http://localhost:8080/RestExample/rest/hello/tom

not

http://localhost:8080/RestExample/rest/hello?name=tom

Although you will see this type of usage

Passing Parameters – in true REST style

```
@Path("/hello")
public class Hello {

  @Path("{name}")
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String sayPlainTextHello(@PathParam("name")
  String name) {
      return "Hello " + name;
  }
```

Benefits of REST

- Familiar Simple Architecture
- Modularity
- OO Design
- Abstraction