How to build a plugin for Servoy
– A step by step tutorial brought to you by Servoy Stuff

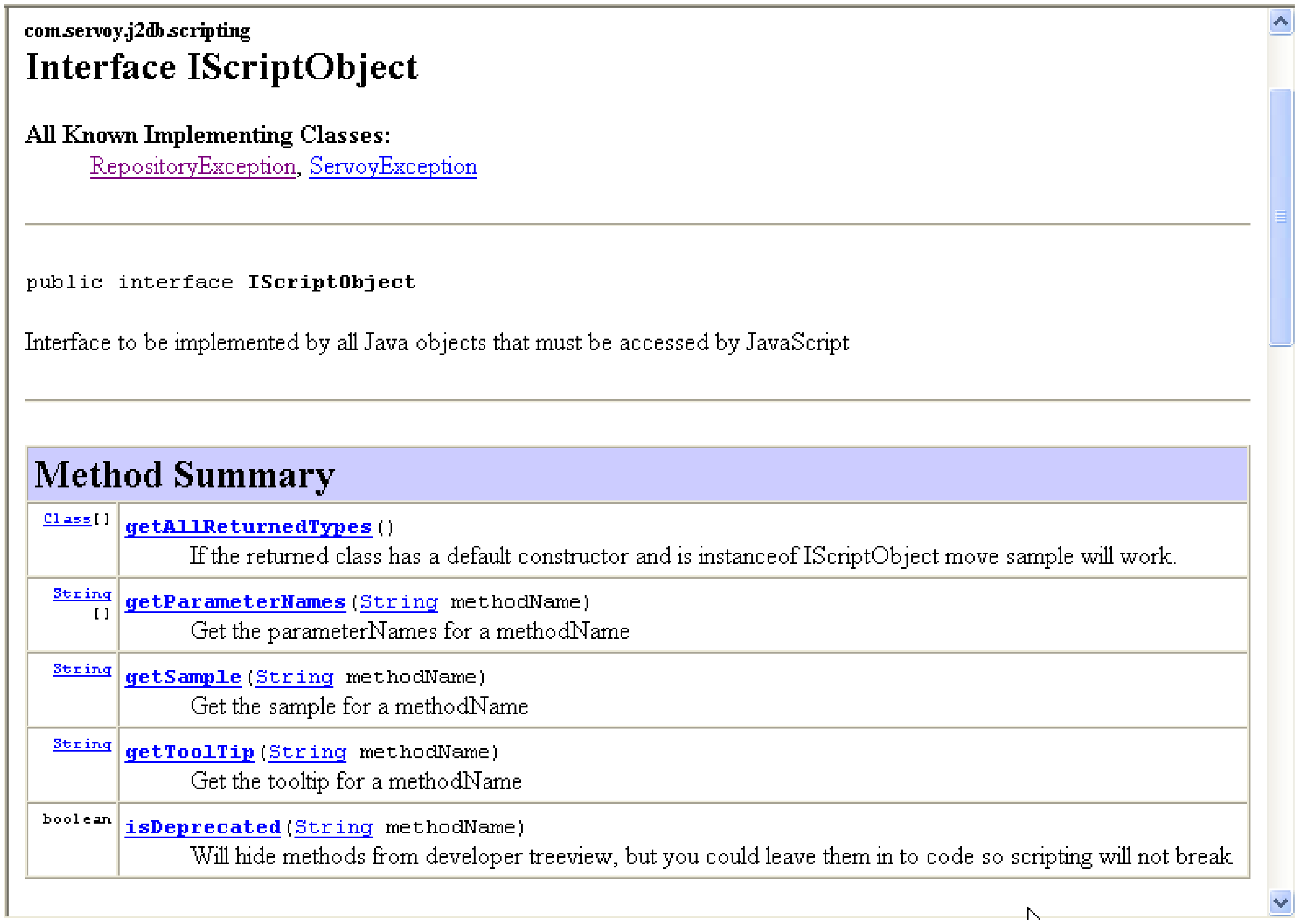
PART 3

J. The IScriptObject interface

In the first part of this tutorial we went through the tedious details of configuring our Servoy/Eclipse environment, and set-up our project "WhoisPlugin", now it's time to actually build the plugin.
Then in the second part, we implemented the IScriptPlugin using (and describing as we went along) the Eclipse IDE, we learned to rely on the API and to use the interface in an efficient way.

Now is (finally!) time to implement our second main class, the "WhoisPluginProvider" class for which we already have a stub nicely prepared for us by Eclipse. As you remember, all we have to do is to follow the "TODO" task trail and implement each method one after the other. Or so it seems…

But first, we need to take a good look at the IScriptObject interface that our WhoisPluginProvider class is going to implement, so let's browse to the public API and click on the relevant link to get the interface definition:



com.servoy.j2db.scripting

Interface IScriptObject

All Known Implementing Classes:
RepositoryException, ServoyException

`public interface IScriptObject`

Interface to be implemented by all Java objects that must be accessed by JavaScript

Method Summary	
`Class[]`	`getAllReturnedTypes()` If the returned class has a default constructor and is instanceof IScriptObject move sample will work.
`String[]`	`getParameterNames(String methodName)` Get the parameterNames for a methodName
`String`	`getSample(String methodName)` Get the sample for a methodName
`String`	`getToolTip(String methodName)` Get the tooltip for a methodName
`boolean`	`isDeprecated(String methodName)` Will hide methods from developer treeview, but you could leave them in to code so scripting will not break

We're in luck because this interface doesn't extend any other, so we only have to implement the methods above.
Now, let's look at them in details:

- getAllReturnedTypes() should return a Class array ("Class" is a java class that defines the type of class). I particularly like the comment here which is ambiguous and quite misleading:

"If the returned class has a default constructor and is instanceof IScriptObject move sample will work."
What the hell are they talking about?

In fact, what they really mean is that you can return here an array of all the types of objects that the scripting methods of this object will return. This is useful if the returned objects need to be accessed by the scripting environment too (Servoy).
Just imagine that you write a scripting method that, when invoked by the client using the plugin, would return another object and that this object itself has useful methods that you want to expose to the client.
For an example of this, look at the http plugin for example, and see in the Servoy's Solution Explorer how you can deploy it in the tree to reveal 2 "inner" objects: "Poster" and "Cookie", now if you look at the sources of this plugin (they are in the jar!), you will see that the getAllReturnedTypes() method of the HttpProvider class returns an array of Poster and Cookie classes:

```
	public Class[] getAllReturnedTypes() {
		return new Class[] { Poster.class, Cookie.class };
	}
```

Note the construct that looks a lot like JavaScript: you create an array by filling it with values (only that they are enclosed by parenthesis), and note too how you can access the Class object of any class.
Now one interesting thing to add is that even if the misleading comment above tells you that the returned classes need to be instances of (*instanceof* is the operator that can test what an object's class, superclasses or interfaces are), this is not true at all. You can try it yourself, and return a String.class in your Class[] array, and you will see that it will appear in the Explorer node under your plugin, exposing all its methods to innocent JavaScripters ;-)

This is what I would call the "**naked object technique**" - if you ever need to access a java object directly, you can do a plugin that will return an object of this type, and you will find that all its methods will be exposed! Now the IScriptObject interface in this case is actually acting as some kind of opaque underwear to hide the facts of life in java to the JavaScript children ;-)
Now this is true with the latest (as of today) version of Servoy v4.1.3, I cannot guarantee that it will still be the case in the next major releases, but right now it can be convenient, something to keep in mind anyway.

Back to our methods, the next one is:

- getParameterNames(String methodName) should return an array of Strings (String[]): this is where you will start implementing the clever build-in help system of Servoy. See that when you point your mouse over a method in the Solution Explorer, there is a tooltip, and when you click on it, the method will appear in the (annoyingly not resizable) footer of the Eclipse interface. What you return here will help construct the "signature" of your JavaScript method, these are the parameters that you will see in the first line of the tooltip, and the first line of the footer enclosed in the bracket of the JavaScript method.

Now since the client (Servoy) is giving you when it calls this method which method name it is, you will have to test the methodName and return an array depending on it. Look at the HttpProvider class and you will see that there is a few methods and that the implementation of the method getParameterNames() is testing the methodName with a series of if…else if construct. And look at the getMediaData test for example and you will see that the method returns

```
new String[] { "url", "[http_clientname]" };
```

Note the [] inside the string for the second parameter and you will be able to deduce that these strings will be inserted verbatim inside the tooltip

Next one is fairly easy to understand:

- getSample(String methodName) should return a String. You have guessed that given the method name that Servoy will give you, it expects in return a String for the "move sample" function of the build-in help system. So use some if…else if to return a String with a sample usage for the relevant method, as simple as that.
- getTooltip(String methodName) should also return a String, and this time it is used by the build-in help system to construct the second line of the tooltip (first line is deduced by the name of the method itself and the parameterNames we saw earlier). You implement it with the same if… else if construct, except that you can return a shorter String here.
- isDeprecated(String methodName) should return a boolean: this time the comment is explicit enough, just think that if you ever need to update your plugin and change your methods but still want to keep compatibility with the previous versions, it might be good to return true to the methods that should be used anymore. As the comment says, they will be hidden in the Solution Explorer plugin tree, even if they are still there, to avoid breaking old code.

That's it. Yes that's it.
But wait! If we implement these methods right now it doesn't mean that our plugin is finished, is it? Well, the truth is that it might compile ok, but it will not be doing anything useful since we didn't code any custom method. The users will see your plugin, but that's it. Your plugin will do nothing at all!

Because in fact before coding these methods you really need to think about what your plugin is going to do. You need to define what methods will be useful to your users, what parameters you are going to allow, and basically what you plugin is doing for a living.

K. Coding the plugin main behaviour

We said in the introduction that the requirements for our plugin were to query a whois server to query a domain and return the information about it.

Now, I'm not going to go too deep in the details of that, just a few hints because this is the kind of code that you will find in many java IO tutorials. You can find information on the internet about how to access a server using a Socket object, how to read and write using InputStream and OutputStream or Reader and Writer objects.

Basically to communicate to a server using any protocol, we need to create a Socket. If we look at the definition of a Socket in the java API, it states that "A socket is an endpoint for communication between two machines." – just what we need.

Now to create a Socket, there are lots of constructors, but basically the one we need is the one were we can specify a remote server (our whois server) and a port (the default port used by whois servers –default being 43). So it will be something like that:

```
Socket socket = new Socket(server, port);
```

Then we also need to define a timeout, because we don't want to wait forever, right?
Our user might prove very impatient if for any reason the network is not responsive enough. So we setup a timeout like this (the unit is milliseconds):

```
        socket.setSoTimeout(timeout);
```

Now to be able to communicate on a socket, java use the IO API, and the base interfaces to read/write on IO endpoint (whatever they are, sockets, files, etc.) are InputStream/OutputStream and Reader/Writer – just look in the java API for these. So let's create a Reader to read from like this:

```
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

It's always good practice to encapsulate your streams into a buffered implementation (after all you don't know how big the stuff to read will be, you wouldn't want to load a huge amount of data in memory, now a buffered implementation (of InputStream/OutputStream or Reader/Writer) just guarantee that if needed it will use some caching to deal with it.

Now that we can read from our socket using our Reader, we also need to be able to write onto it (to send the domainName query), so we create an OutputStream like this:

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

This time we don't really need a buffer since there is going to be a very little amount of data to send (a simple String).

Now we are all set, let's call the whois server with the domainName query:

```
out.writeBytes(domainName +" \r\n");
```

We terminate with a carriage return to tell the distant server that our name ends here.

It's time to get the response, we do it like that:

```
      String str1 = null;
      StringBuffer buffer = new StringBuffer();
      while ((str1 = in.readLine()) != null) {
            buffer.append(str1);
            buffer.append("\r\n");
      }
```

We define a string to receive each line, and a StringBuffer to concatenate the lines received (StringBuffer is a way to optimize String creation in java, you will find many examples of its use, it is an essential base class for String manipulation, see the java API for more info).
Now we can iterate for each line received and fill our StringBuffer. Once finished, the readLine() method will return null, so we can exit our loop.

To wrap up, we need to clean our mess by closing everything we opened (in the reverse order to what they were opened), so we write:

```
      // close our stream and reader
      out.close();
      in.close();
      // close the socket
      socket.close();
```

And finally we will return the result of our query like that:

```
      return buffer.toString();
```

No need to test for null here, we know our buffer isn't since we created it ourselves a few lines earlier!

To be on the safe side, though we will need to wrap all of this code into a try/catch block, to trap all the nasty Exceptions that can happen when dealing with the network – you know that one day or another they will!

So our whole method code is going to be:

```
try
{
	// create the socket
	Socket socket = new Socket(server, port);
	socket.setSoTimeout(timeout);

	// create a reader to get the response from the server
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

	// create an output stream to send our query to the server
	DataOutputStream out = new DataOutputStream(socket.getOutputStream());
	// call the service with the domainName supplied
	// and terminate with carriage return)
	out.writeBytes(domainName +" \r\n");

	// read the response from the server
	String str1 = null;
	StringBuffer buffer = new StringBuffer();
	while ((str1 = in.readLine()) != null) {
		buffer.append(str1);
		buffer.append("\r\n");
	}

	// close our stream and reader
	out.close();
	in.close();
	// close the socket
	socket.close();

	// return the result as String
	return buffer.toString();

} catch (IOException ioEx) {
	return ioEx.getLocalizedMessage();

} catch (Exception ex) {
	return ex.getLocalizedMessage();
}
```

There is 2 set of catch blocks because you might want to do different things if the error is an IOException (network problem) or any other Exception – here it does the same thing return the message of the exception to the caller of our method

That's it. That's our implementation of a method to query the whois server. Now those who are still awake might have noticed that I haven't used any String for the "server" and domainName nor did I put any int for the port and timeout parameters.

This is because our method "signature" is going to be:

```
public String js_query(String domainName, String server, int port, int timeout)
```

There is more than meets the eye to that method signature, and especially its name.
First, notice that it is "public", meaning that any java object with a reference to an object of this class will be able to call it. After all, we want the Servoy JavaScript environment to be able to call it, so that's fine!
And of course it returns a String, hopefully the response from the whois server or an Exception message if something went wrong.

But the name? You would think that you can name you method anything you want as long as it is public.
To a certain extent that is true, AS LONG AS IT STARTS WITH THE MAGIC "js_" PREFIX.

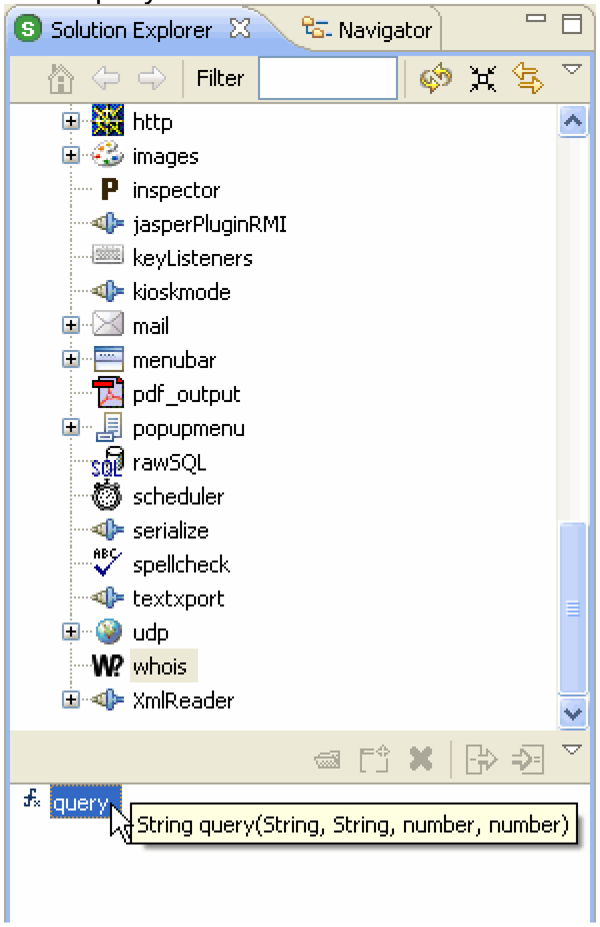
In fact, any public method of an IScriptObject that starts with this "js_" prefix will immediately be available to the Servoy JavaScript environment.
For those of you who don't buy magic, I would explain that in java there is a way to discover what's in any class (what properties, what constructors and methods and with what arguments). It is one of the most powerful feature of the language and it's hidden in an API called "reflection" – see http://java.sun.com/docs/books/tutorial/reflect/index.html for an in-depth explanation of what Reflection is, how powerful it is and how you should use it wisely if you want to avoid performance problems.

Basically, even if I don't know the exact way Servoy has implemented it, I know that there is a fair chance that it's using the Reflection API on the IScriptObjects found to know what kind of methods to display in the Solution plugin tree and to give you access to them in the scripting environment.

Now if you want to have some fun, you can try deploying your whois plugin as-is, with just this one method and deploy it as a jar (see further on) into the /plugins folder of Servoy. If you do that, even without having implemented any of the IScriptObject interface (other than the empty stubs that Eclipse did for you, you would find that you plugin is totally operational, already!), to prove it, I have made a version of the plugin with only this method and a few aliases method (see further on) and packaged it into a plugin.
You can find it here:
http://servoy-stuff.net/tutorials/utils/t01/whois-v00.jar

If you use Internet Explorer and it insists on saying that the download is a zip file, all you need to do is to rename it after the download from whois-v0.zip to whois-v0.jar again – I guarantee you IT IS a jar file!

Put it into your "/plugins" folder and restart Servoy developer, you will see that our little plugin is here already, and more important our js_query method has been recognized and appear in the tree under the label "query":



String query(String, String, number, number)

Not only does it appear, but it is perfectly operational!
If you create a form with a simple method like this for example:

```
function itMagicallyWorks()
{
      application.output(plugins.whois.query("servoy.com",
                                              "whois.networksolutions.com",
                                              43,
                                              30000));
}
```

And call your method in any developer's client (smart or web), it will output the response from the server in your console panel, a big stream of information starting wit something like that:

```
NOTICE AND TERMS OF USE: You are not authorized to access or query our WHOIS
database through the use of high-volume, automated, electronic processes. The
Data in Network Solutions' WHOIS database is provided by Network Solutions for
information
purposes only, and to assist persons in obtaining information about or related
…
```

Proving that we just made a perfectly functional Servoy plugin, smart and web client compatible!

So what's left would you ask? Well…

L. Smoothing the edges

Our plugin right now is a bit rough around the edge, for a few reasons:

1. Right now we only have one method, and we need to supply every arguments when some of them should really be optional (like port and timeout, even the whois server should have a default value). The only really mandatory argument of our method should be the domainName.
2. Our plugin doesn't use any of the nice build-in help system, to help users understand the syntax to use and the order of the arguments (you noticed that the tooltip is only displaying "String query(String, String, number, number) – we will need to tell the users what these Strings and numbers really are.

1. Overloading our js_query method:
To address the first problem, you need to understand a fundamental difference between Java and JavaScript. To make it short, let's say that Java is a very strict language and JavaScript is very loose.

You can call a JavaScript function with any number of arguments. You can't do that in Java (unless you use some special constructs introduced with java 1.5 – know as "**varargs**" – look for it on the internet).
But varargs have some constraints too:

- Method with varargs is called only when no other method signature matches the invocation.
- Only one varargs per method can be declared.
- Varargs should be the last argument (set) for a method. So
  public void varargTest(int i, String ... args) is valid, but
  public void varargTest(String ... args, int i) is not.
- Zero or more arguments can be passed for varargs unlike an array where at least a null has to be provided.

I prefer not to use varargs in Java (you can call me old school) if I can avoid them. And anyway you cannot use varargs for a mix of object types like we have: in our method signature we have String and int. Meaning that we would need either to have two varargs (which is a no-no in Java - see above), or to have a signature like *public String js_query(Object... arguments)* which would force us to do some casting to retrieve usable objects, a waste of time and a lot of code in perspective, I don't think it's worth it.

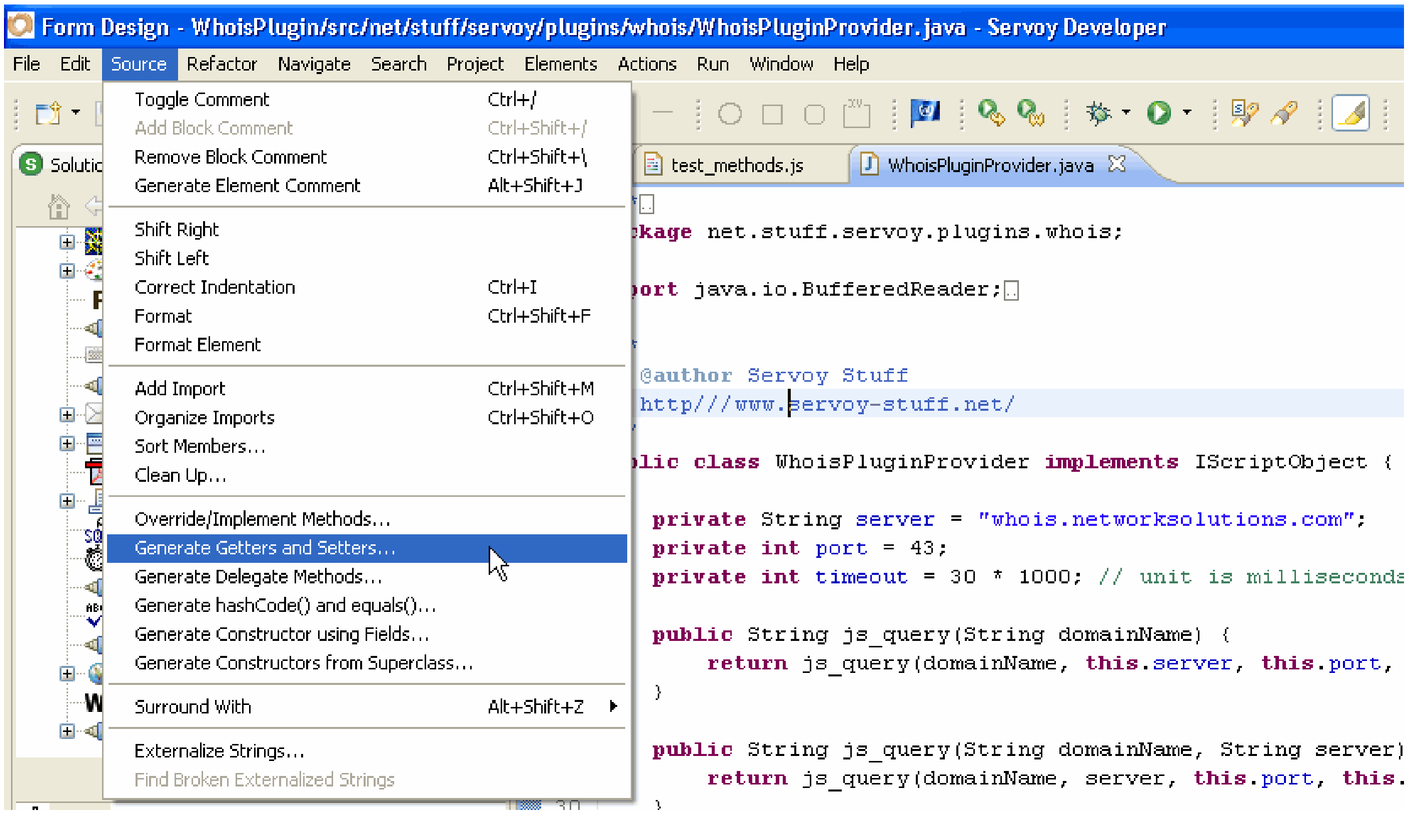
There is another way to deal with "optional" parameters, and this is **overloading** our method.
Basically we will do some alias method of method with less parameters, and these alias methods are going to call our "core" method providing it with always the same set of parameters coming from default values.
Let's see how.

First we need to provide the "default" values for all the arguments that are going to be optional. So let's define some private variables at the top of the class:

```
private String server = "whois.networksolutions.com";
private int port = 43;
private int timeout = 30 * 1000; // unit is milliseconds
```

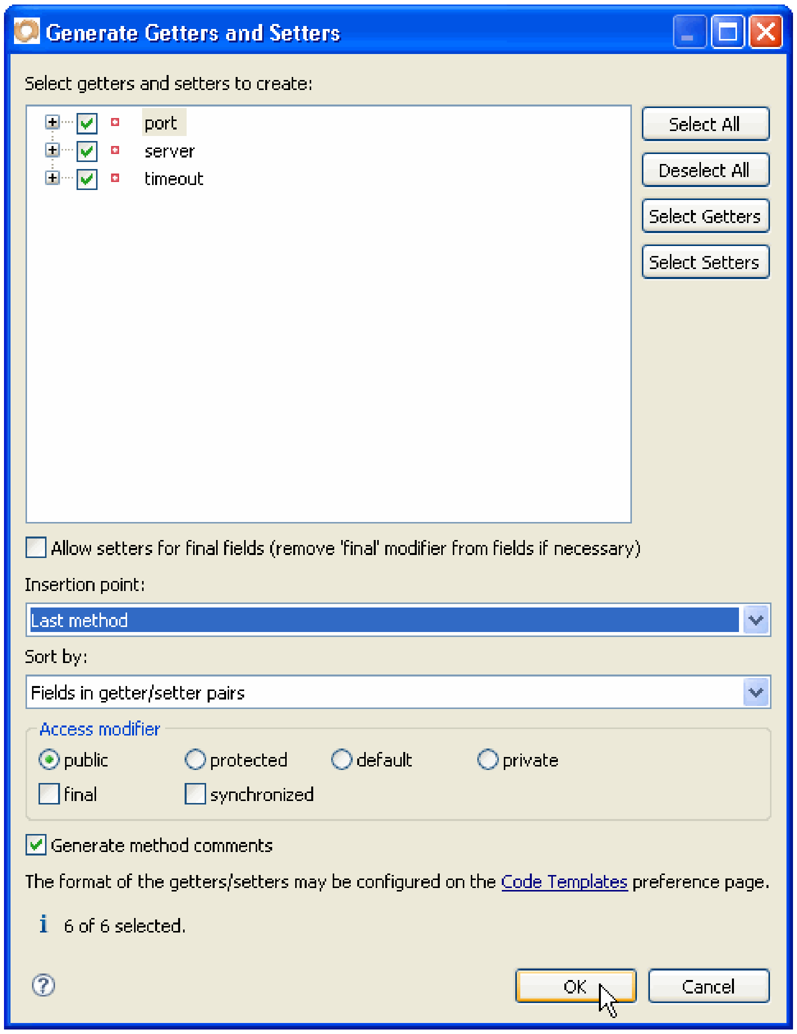
Note that it could be nice to create some accessors (get and set methods) to these properties, in case the user would want to set up its own default values, so let's add the getters and setters. In Eclipse there is an easy way to do it.

Once you have typed your private variables in the class, save it and go to the menu “Source > Generate Getters and Setters...”


Form Design - WhoisPlugin/src/net/stuff/servoy/plugins/whois/WhoisPluginProvider.java - Servoy Developer
File Edit Source Refactor Navigate Search Project Elements Actions Run Window Help
Toggle Comment Ctrl+/
Add Block Comment Ctrl+Shift+/
Remove Block Comment Ctrl+Shift+\
Generate Element Comment Alt+Shift+J
Shift Right
Shift Left
Correct Indentation Ctrl+I
Format Ctrl+Shift+F
Format Element
Add Import Ctrl+Shift+M
Organize Imports Ctrl+Shift+O
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Surround With Alt+Shift+Z
Externalize Strings...
Find Broken Externalized Strings
test_methods.js
WhoisPluginProvider.java

```
kage net.stuff.servoy.plugins.whois;

ort java.io.BufferedReader;

 @author Servoy Stuff
 http///www.servoy-stuff.net/

lic class WhoisPluginProvider implements IScriptObject {

 private String server = "whois.networksolutions.com";
 private int port = 43;
 private int timeout = 30 * 1000; // unit is milliseconds

 public String js_query(String domainName) {
     return js_query(domainName, this.server, this.port,
 }

 public String js_query(String domainName, String server)
     return js_query(domainName, server, this.port, this.
```

The following dialog will show you all the private properties (variables) that Eclipse discovered in your class. Check all the properties for which you want Eclipse to generate the accessors, choose a place to put them (I usually put them at the end of the class, this are not really interesting/essential code as such), and click OK.


Generate Getters and Setters
Select getters and setters to create:
Select All
Deselect All
Select Getters
Select Setters
OK
Cancel

[✓] port
[✓] server
[✓] timeout

[ ] Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: Last method

Sort by: Fields in getter/setter pairs

Access modifier
[● public] ○ protected ○ default ○ private
[ ] final [ ] synchronized

[✓] Generate method comments

The format of the getters/setters may be configured on the Code Templates preference page.

6 of 6 selected.

You will see that Eclipse has generated a set of methods like this:

```
	/**
	 * @return the server
	 */
	public String getServer() {
		return this.server;
	}

	/**
	 * @param server the server to set
	 */
	public void setServer(String server) {
		this.server = server;
	}
```

This for each property you have selected.

I will not explain why this is better than having your properties simply set as “public”, it goes with the whole idea of **encapsulation**, and if you’ve never heard of it, you should look for it on the internet... Anyway for now, suffice to say that this is useful, this is cleaner, this is secure, and in the future this will help you manage your code.

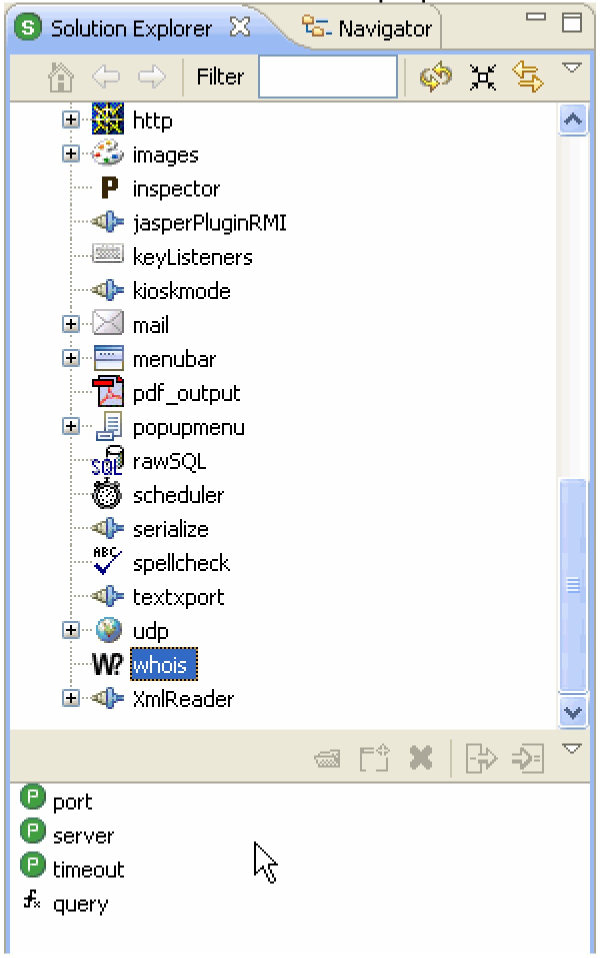
So now that we have getters and setters for all the properties that might be changed, what if we made so that users can access them from the scripting interface?
You should have guessed by now that to be available by the interface, all you need to do is provide methods starting by the "js_" prefix, so let's do that. Rename all the getXXX and setXXX that Eclipse just created for you in js_getXXX and js_setXXX.

Since we have a pair of getter and setter for each variable, Servoy will understand that this is really properties and it will allow users from the scripting environment to use this kind of syntax:

```
      plugins.whois.server = "whois.internic.net";
```

I have made a jar of this intermediate version for you to test it:
http://servoy-stuff.net/tutorials/utils/t01/whois-v01.jar

If you put it in your /plugins (get rid of the previous one first, of course!), when restarting Servoy, you should be able to see 3 new properties in the Solution explorer's plugin node, like that:



As you can see, these are marked with a little "P" icon. So, now we know how to make "Properties" for our plugins, let's get back to smoothing the edges of our plugin, this time by adding the capabilities to deal with optional parameters.

I already said that, unless you are using varargs (which will not do in our case) you cannot use optional arguments in java methods. So how are we going to allow for users to call our method with only one, 2 or 3 arguments when our core method needs 4?

The answer to that is called operator “**overloading**”. Basically, the principle of overloading is that for one method with a certain set of parameters, you write some other methods with another set of parameters but with the SAME NAME.

Remember that our core method (the one really doing the job) was using this “signature”:

```
public String js_query(String domainName, String server, int port, int timeout) {…}
```

Now we are going to define these 3 methods:

```
public String js_query(String domainName) {…}
public String js_query(String domainName, String server) {…}
public String js_query(String domainName, String server, int port) {…}
```

As long as the set of parameters (arguments) of our method is different from any other, the compiler will be happy (because he will be able to recognize it as different from any other), and will not complain that “a method with the same name already exists”.

But are you going to copy the content of our core method into each of these new methods, changing the content slightly so as to use your default variables? Of course not, all you have to do is to call the “core” method from within the “alias” or “overloaded” methods, like so:

```
	public String js_query(String domainName) {
		return js_query(domainName, this.server, this.port, this.timeout);
	}

	public String js_query(String domainName, String server) {
		return js_query(domainName, server, this.port, this.timeout);
	}

	public String js_query(String domainName, String server, int port) {
		return js_query(domainName, server, port, this.timeout);
	}
```

As you can see above, each method is really just passing the call to our “core” method, providing our already set “default” value by using the inner variables of our class (the `this` keyword here is really optional, but it is just a way of making clear – for you, later - that the variables are the inner variables of our class, the compiler knows that already).

So with this set of alias methods we are now able to use these kinds of calls from the scripting environment:

```
plugins.whois.query("servoy.com");
plugins.whois.query("servoy.com", "whois.networksolutions.com");
plugins.whois.query("servoy.com", "whois.networksolutions.com", 43);
plugins.whois.query("servoy.com", "whois.networksolutions.com", 43, 30000);
```

And the calls will do just the same thing!
You can try it if you don’t believe me with the previous whois-v01.jar that I contained the **overloaded** methods already.

2. Implementing the method for the build-in help system:

Last thing we need to do is to implement the standard “IScriptObject” interface methods.

Remember them?
First one is:
```
	public Class[] getAllReturnedTypes()
```

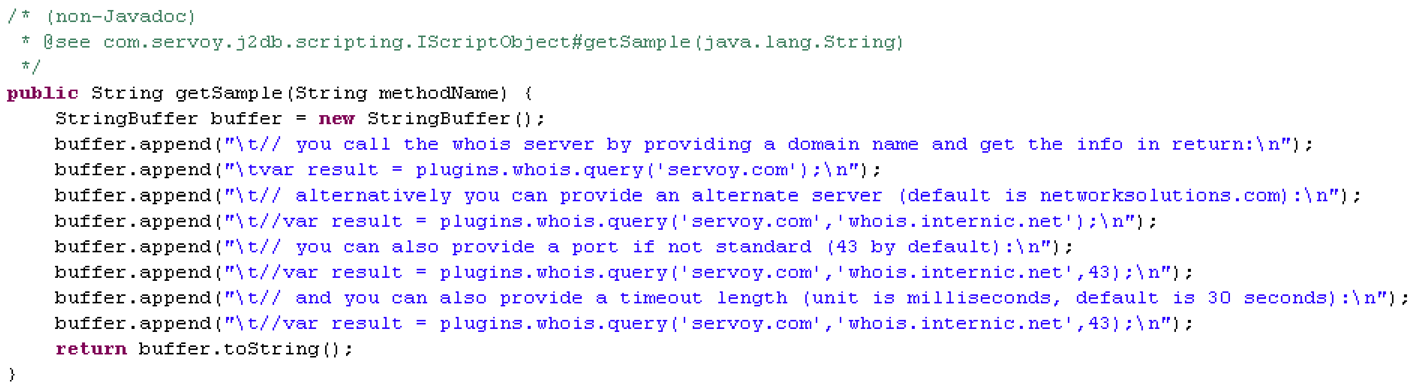
Here we can return null (get rid of the `// TODO Auto-generated method stub` to indicate that we finished implementing the method).

That was easy! Next one is:
```
	public String[] getParameterNames(String methodName) {
		return new String[] {"domainName", "[server]", "[port]", "[timeout] "};
	}
```

You remember the construct? We create a static String array, and give back the name of all the parameters of our method. Note that to indicate that some parameters are optional, we enclose the parameter name String with square brackets, and this is how it will appear in the signature tooltip of our method in Servoy.

OK, next is:
```
	public String getSample(String methodName)
```

This time we are going to send back a useful sample usage of our method (this is the one which will be pushed into the scripting editor when using the “move sample” command of Servoy.


```
/* (non-Javadoc)
 * @see com.servoy.j2db.scripting.IScriptObject#getSample(java.lang.String)
 */
public String getSample(String methodName) {
    StringBuffer buffer = new StringBuffer();
    buffer.append("\t// you call the whois server by providing a domain name and get the info in return:\n");
    buffer.append("\tvar result = plugins.whois.query('servoy.com');\n");
    buffer.append("\t// alternatively you can provide an alternate server (default is networksolutions.com):\n");
    buffer.append("\t//var result = plugins.whois.query('servoy.com','whois.internic.net');\n");
    buffer.append("\t// you can also provide a port if not standard (43 by default):\n");
    buffer.append("\t//var result = plugins.whois.query('servoy.com','whois.internic.net',43);\n");
    buffer.append("\t// and you can also provide a timeout length (unit is milliseconds, default is 30 seconds):\n");
    buffer.append("\t//var result = plugins.whois.query('servoy.com','whois.internic.net',43);\n");
    return buffer.toString();
}
```

Note that I used a StringBuffer (always good practice when you need to concatenate Strings) and returned the string with the toString() method.
Note also that I use **`\t`** before each line, so that the inserted code lines up cleanly in the JavaScript editor.
And finally each line is finished by a newline character **`\n`**.

You will be able to download the finished java file here:
http://servoy-stuff.net/tutorials/utils/t01/WhoisPluginProvider.java

Next one is:
```
public String getToolTip(String methodName) {
return "Calls a whois server to retrieve information about the domain name you provide";
}
```
That will do, but in any case, feel free to put something more significant if you think it’s not clear enough.

And the final method, easily enough, is already done for us, just get rid of the // TODO comment to mark it as done:
```
	public boolean isDeprecated(String methodName) {
		return false;
	}
```

This is because our method is not deprecated yet, we just wrote it!
So we just answer “false” to Servoy’s inquiry about the freshness of our code here. ;-)

We can go to our “Tasks” view to confirm that it is empty. If not, go to the tasks by double-clicking on it and verify that you got rid of the // TODO comments and implemented the code correctly.

This time all the code needed for our little plugin to function nicely in the Servoy environment is done.
Now all we need to do is to wrap our plugin into a nice little jar and put it in the /plugins folder, and build a little test solution to go with it…

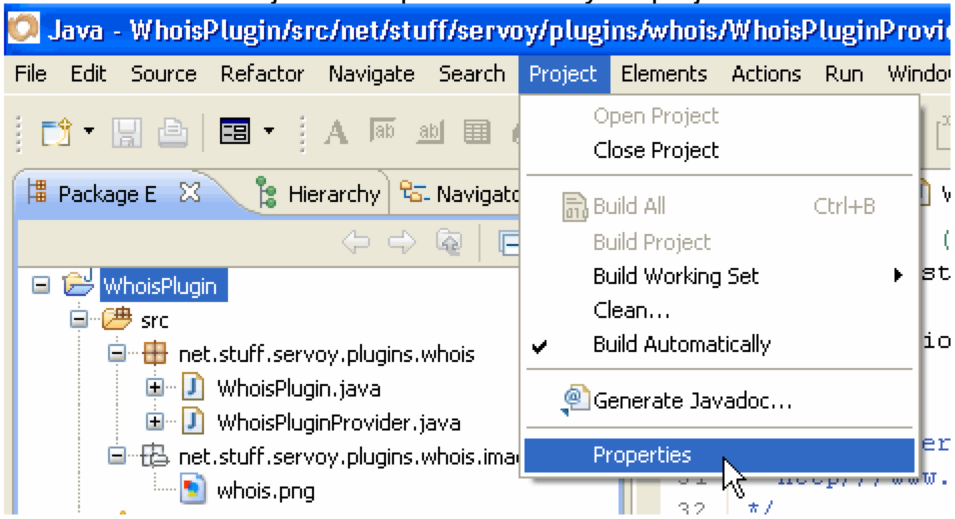
M. Building for a specific target

Right now, we didn’t ask ourselves whether we wanted our plugin to work for Servoy 3.5.x and above or Servoy 4.1.x
You might think that it’s equal, providing that we used only compatible interfaces in the Servoy API.

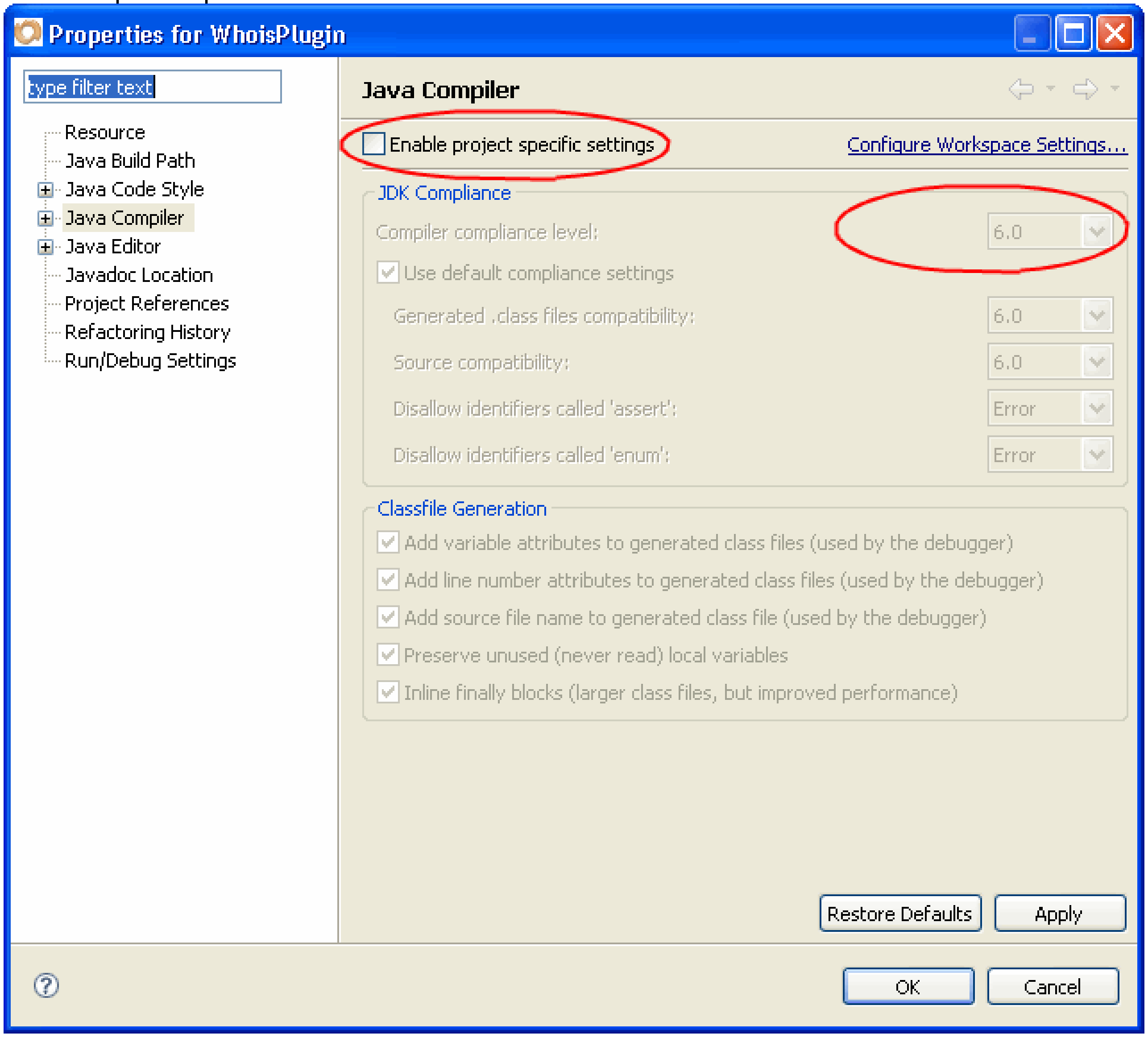
This is not strictly true. Servoy 3.5 is built “against” Java 1.4.x and above while Servoy 4.1 is built “against” Java 1.5.x and above. Now you should know that java changed a lot between 1.4 and 1.5, so you need to set up your target JRE for one version or the other. And since the current JRE is actually 1.6.x it is essential that you choose a target for your compilation, otherwise your source class might not work and your plugin will fail with a “java.lang.UnsupportedClassVersion – Error: Bad version number in class file”.

So how do you target a specific JRE for your project within the Eclipse IDE?

Go to the menu “Project > Properties” while your project is still selected in the Package Explorer:



You get the properties dialog specific to your project, now click on the “Java Compiler Node” and check the “JDK compliance panel”:



Properties for WhoisPlugin

type filter text

- Resource
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project References
- Refactoring History
- Run/Debug Settings

Java Compiler

☐ Enable project specific settings

Configure Workspace Settings...

JDK Compliance

Compiler compliance level: 6.0

✓ Use default compliance settings

Generated .class files compatibility: 6.0

Source compatibility: 6.0

Disallow identifiers called 'assert': Error

Disallow identifiers called 'enum': Error

Classfile Generation

✓ Add variable attributes to generated class files (used by the debugger)

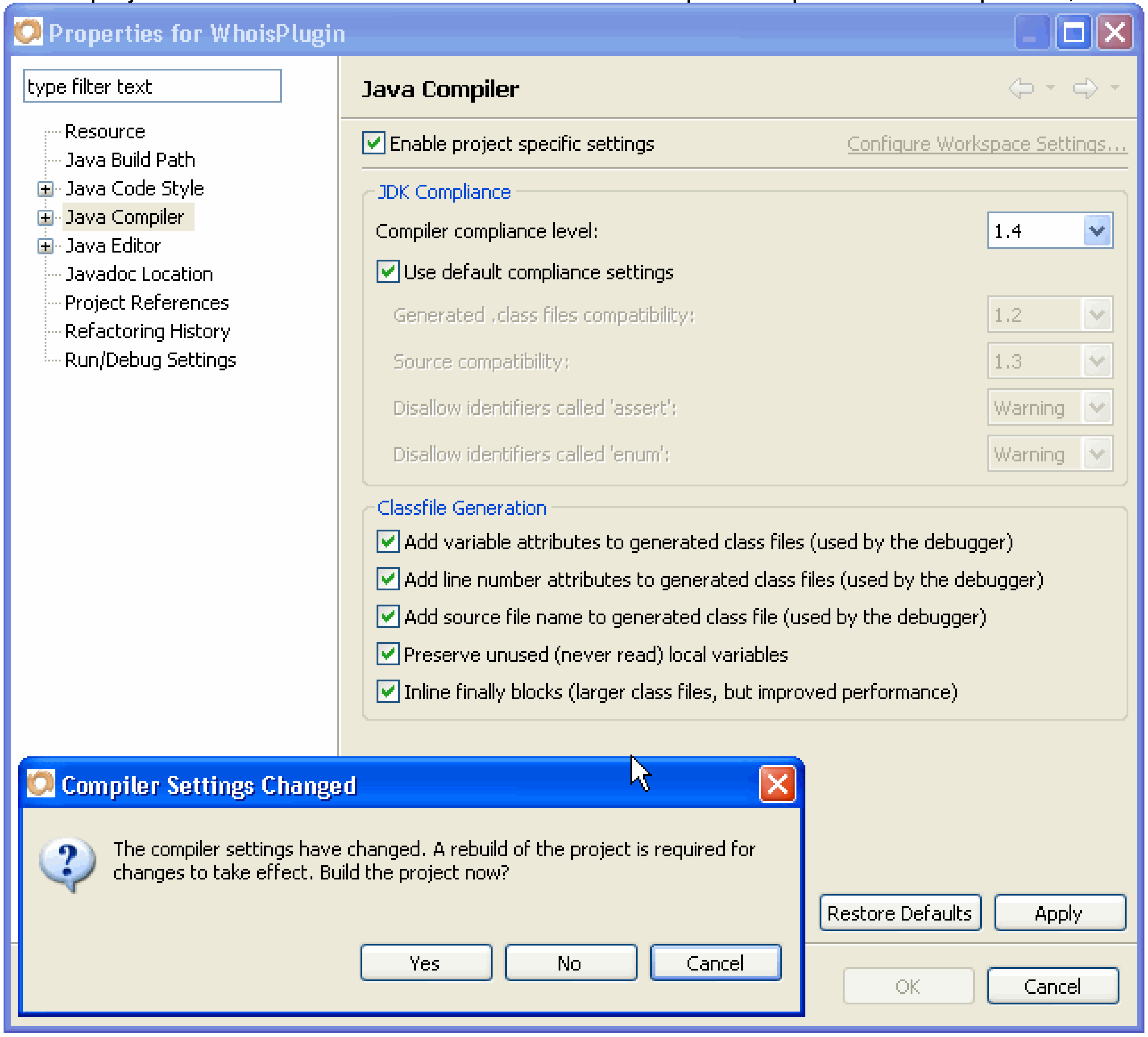
✓ Add line number attributes to generated class files (used by the debugger)

✓ Add source file name to generated class file (used by the debugger)

✓ Preserve unused (never read) local variables

✓ Inline finally blocks (larger class files, but improved performance)

If it says “6.0” that’s bad, we need to check the “Enable project specific settings” to allow a specific target for our project. Let’s check it and choose “1.4” in the “Compiler compliance level dropdown”, then click “OK”:



☑ Enable project specific settings

Compiler compliance level: 1.4

☑ Use default compliance settings

Generated .class files compatibility: 1.2

Source compatibility: 1.3

Disallow identifiers called 'assert': Warning

Disallow identifiers called 'enum': Warning

☑ Add variable attributes to generated class files (used by the debugger)

☑ Add line number attributes to generated class files (used by the debugger)

☑ Add source file name to generated class file (used by the debugger)

☑ Preserve unused (never read) local variables

☑ Inline finally blocks (larger class files, but improved performance)

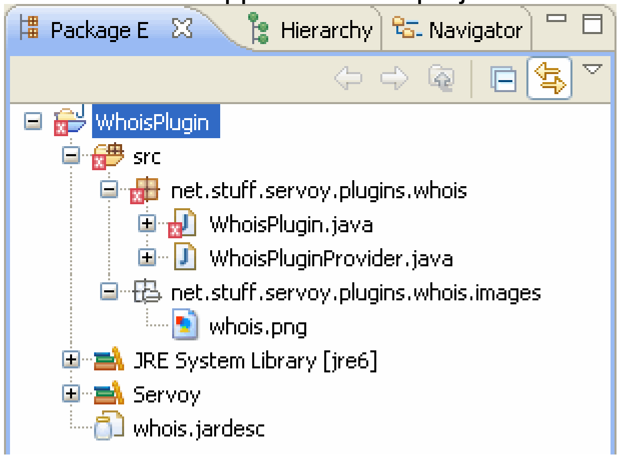
Compiler Settings Changed

The compiler settings have changed. A rebuild of the project is required for changes to take effect. Build the project now?

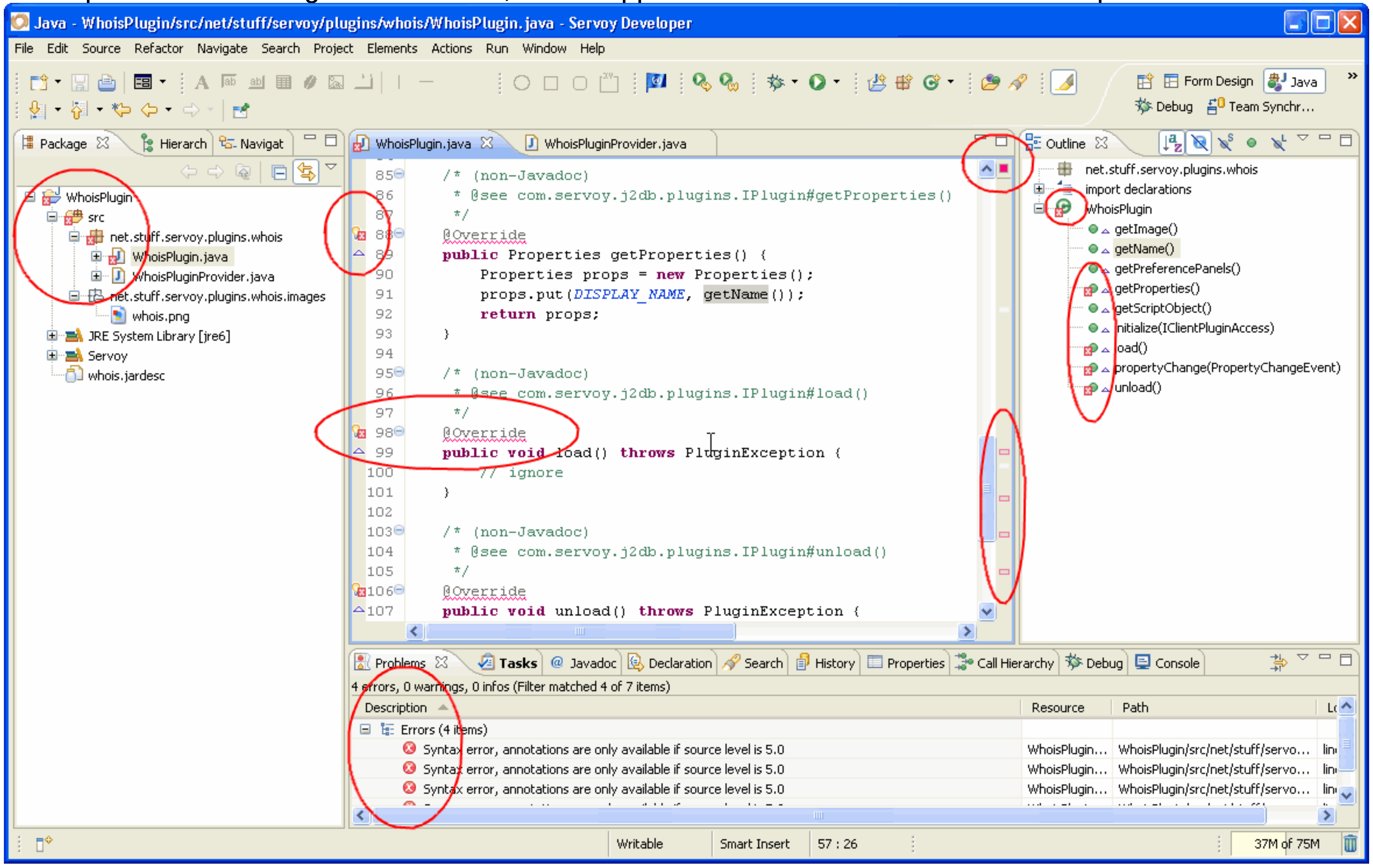
Yes No Cancel

You will see that a dialog informs you that Eclipse noticed that you wanted to change the target for your project, so it says that it needs to rebuild, answer “**Yes**” to that prompt.

But now what happened to our project? When we look at it in the Package explorer view, it displays like this:



Meaning that there are errors in the WhoisPlugin code! Why?
Let's open our WhoisPlugin in the editor, what happened? There are errors all over the place:



```java
    /* (non-Javadoc)
     * @see com.servoy.j2db.plugins.IPlugin#getProperties()
     */
    @Override
    public Properties getProperties() {
        Properties props = new Properties();
        props.put(DISPLAY_NAME, getName());
        return props;
    }

    /* (non-Javadoc)
     * @see com.servoy.j2db.plugins.IPlugin#load()
     */
    @Override
    public void load() throws PluginException {
        // ignore
    }

    /* (non-Javadoc)
     * @see com.servoy.j2db.plugins.IPlugin#unload()
     */
    @Override
    public void unload() throws PluginException {
```

4 errors, 0 warnings, 0 infos (Filter matched 4 of 7 items)

Description	Resource	Path
Errors (4 items)		
Syntax error, annotations are only available if source level is 5.0	WhoisPlugin...	WhoisPlugin/src/net/stuff/servo...
Syntax error, annotations are only available if source level is 5.0	WhoisPlugin...	WhoisPlugin/src/net/stuff/servo...
Syntax error, annotations are only available if source level is 5.0	WhoisPlugin...	WhoisPlugin/src/net/stuff/servo...

The truth is that in the WhoisPlugin the code that was generated by Eclipse in the first place was done according to the level of compliance which was 6.0. And Java since 1.5 had added some new options to the language, namely (in our case) what is known as "annotations". You will see in the "problems" view of Eclipse that the errors are all related to the fact that "annotations are only available if source level is 5.0", since we changed our level to 1.4 annotations are no longer valid.

So basically all we need to do is to get rid of them: they are these lines (currently with a squiggly red underline) like that:

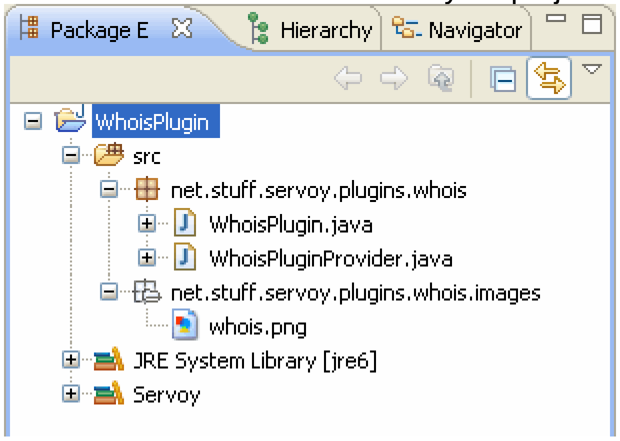
```
@Override
```

Go ahead! Delete all these pesky annotations for good. Once you are done, your class will compile and you will no longer have errors anywhere!

Now what exactly are these annotations used for?
I'm not going to give you a tutorial on that, you can always look on the internet to find more, but in our case "*@Override*" was just a way to say to the compiler that these methods were related to some methods coming from a superclass (in fact overriding a method from an ancestor of our interface IClientPlugin).
And there are times when these little things come useful, mainly when you change the code of your superclass (without knowing that there are sub classes that have overridden them), the class where the code was overridden will not be able to compile anymore, giving you the opportunity to fix your code before running it, which is always better than having your code break at runtime and have your users cry for help in front of a stack trace of full of exceptions!

N. Packaging the plugin

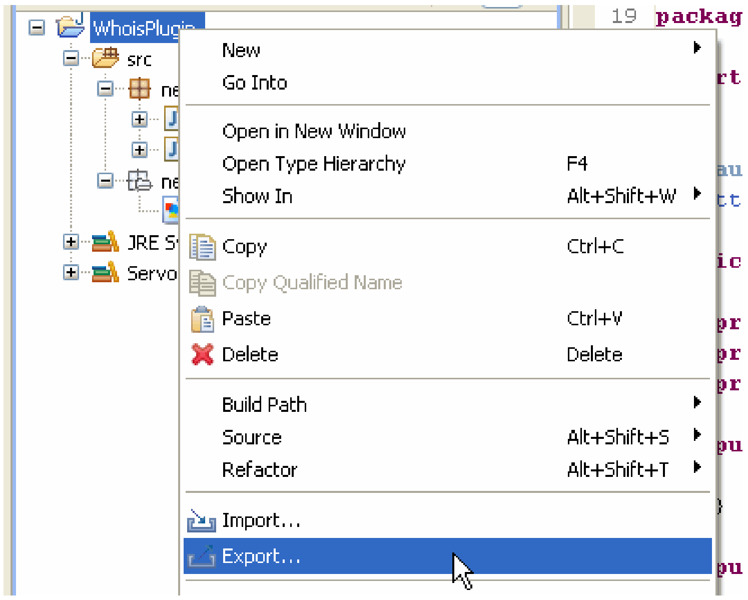
This below should be the state of your project now as seen in the java "Package" explorer of Eclipse:



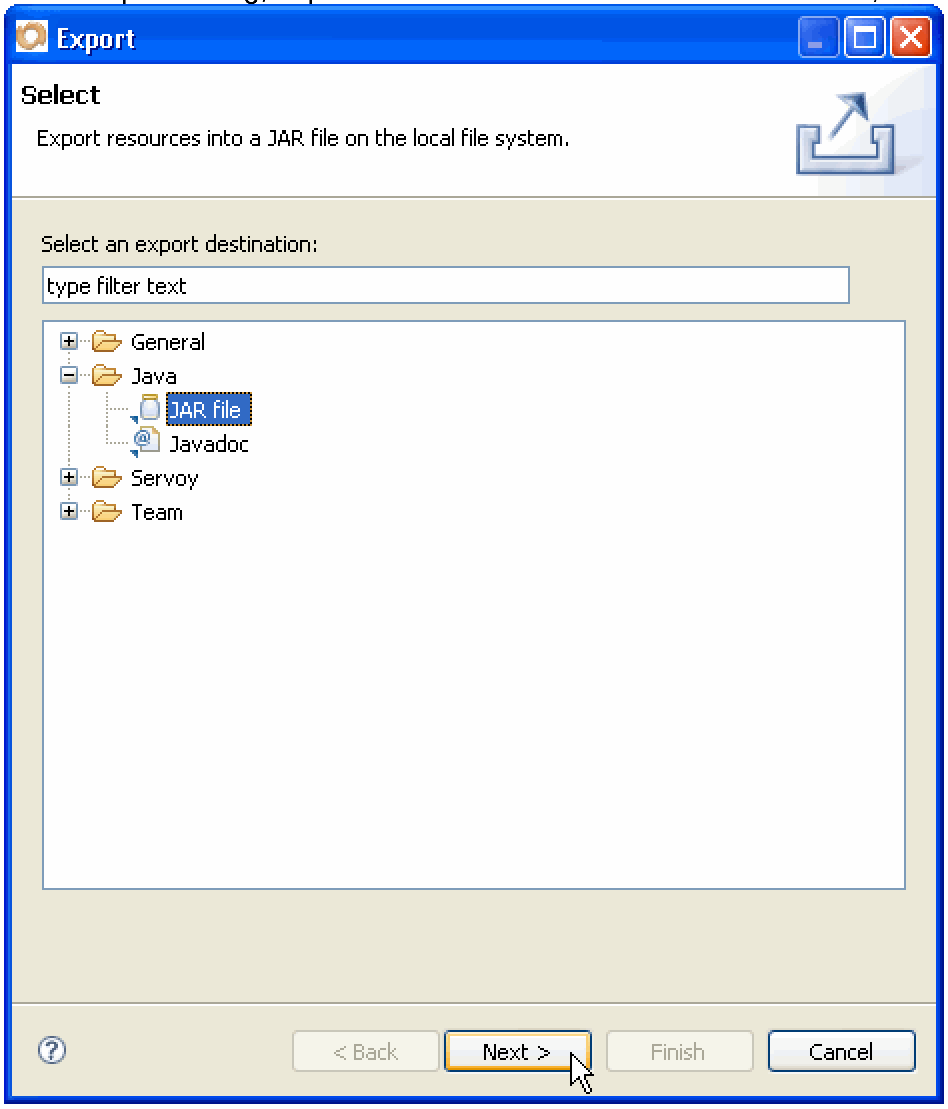
```
WhoisPlugin
  src
    net.stuff.servoy.plugins.whois
      WhoisPlugin.java
      WhoisPluginProvider.java
    net.stuff.servoy.plugins.whois.images
      whois.png
  JRE System Library [jre6]
  Servoy
```

If you still have some warnings (little x on red background attached to our .java files) check your code!
If in doubt, download the 2 classes I have put on the Servoy Stuff site.

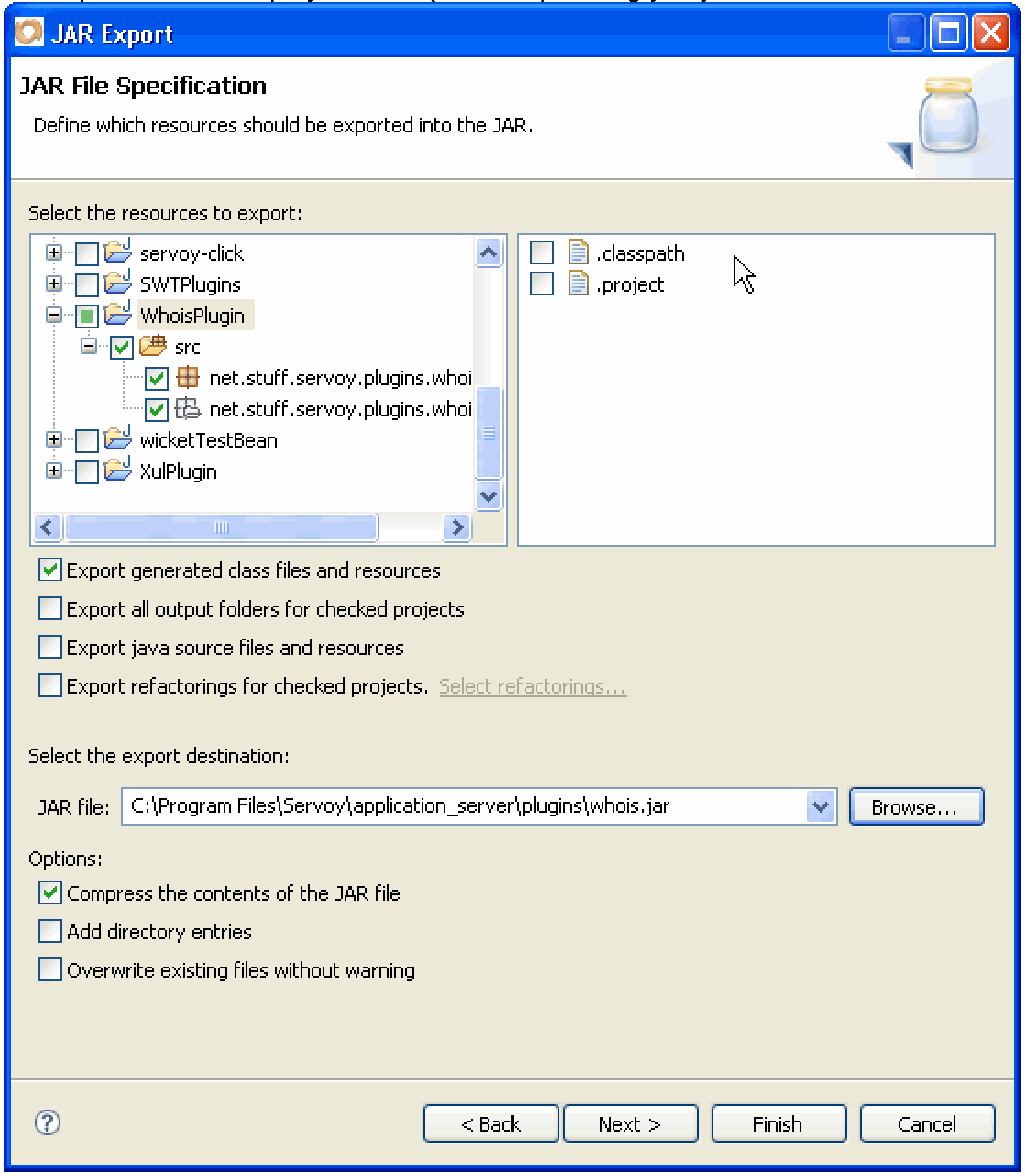
To create a jar file for our plugin, right-click on the project folder in the Package Explorer and choose "Export...":



In the export dialog, expand the “Java” node and choose “JAR file”, then click next:



In the JAR Export dialog, the project should already be selected, (if not, select it :), and you can deselect the ".classpath" and the ".project" files (to avoid polluting you jars with unwanted files only used by Eclipse)



JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- [ ] servoy-click
- [ ] SWTPlugins
- [■] WhoisPlugin
  - [✓] src
    - [✓] net.stuff.servoy.plugins.whoi
    - [✓] net.stuff.servoy.plugins.whoi
- [ ] wicketTestBean
- [ ] XulPlugin

- [ ] .classpath
- [ ] .project

- [✓] Export generated class files and resources
- [ ] Export all output folders for checked projects
- [ ] Export java source files and resources
- [ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\Program Files\Servoy\application_server\plugins\whois.jar

Options:

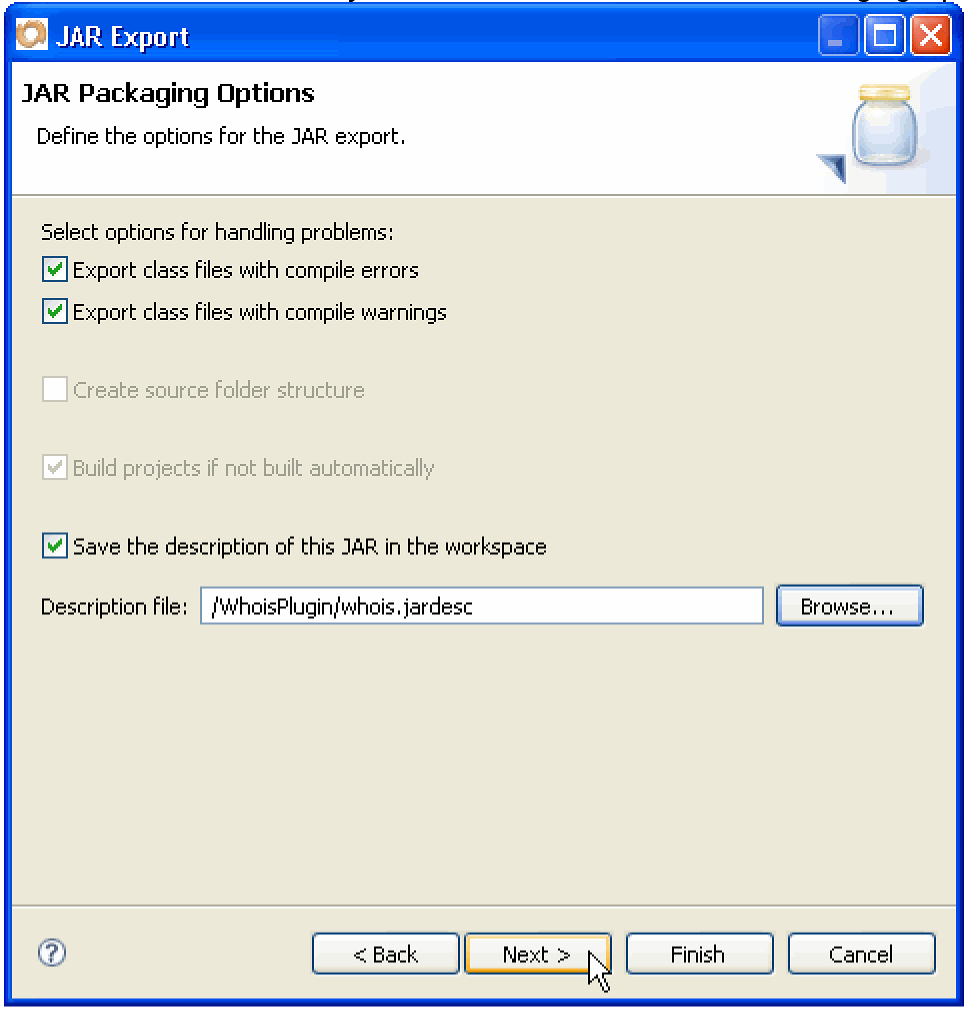
- [✓] Compress the contents of the JAR file
- [ ] Add directory entries
- [ ] Overwrite existing files without warning

Check the "Export generated class files and resources", leave the other one unchecked (see Eclipse doc for more information about these options)

You can choose to export the jar directly into your /$servoy_install_dir$/application_server/plugins directory, just use the browse button to navigate to it, and give it a name (let's use "whois.jar") – you can choose to locate it somewhere else. Personally, I like to put it directly there, so that if I want to test it, all I have to do is restart Servoy developer.
In any case it won't break your Servoy install, so I consider it safe to copy it directly there, it's a nice shortcut.

Finally, check the "Compress the contents of the JAR file" to make it lighter, and leave the other options unchecked (again for more details, see the Eclipse doc).

Click the “Next >” button, you are now faced with the “JAR Packaging Options” dialog:



JAR Packaging Options

Define the options for the JAR export.

Select options for handling problems:

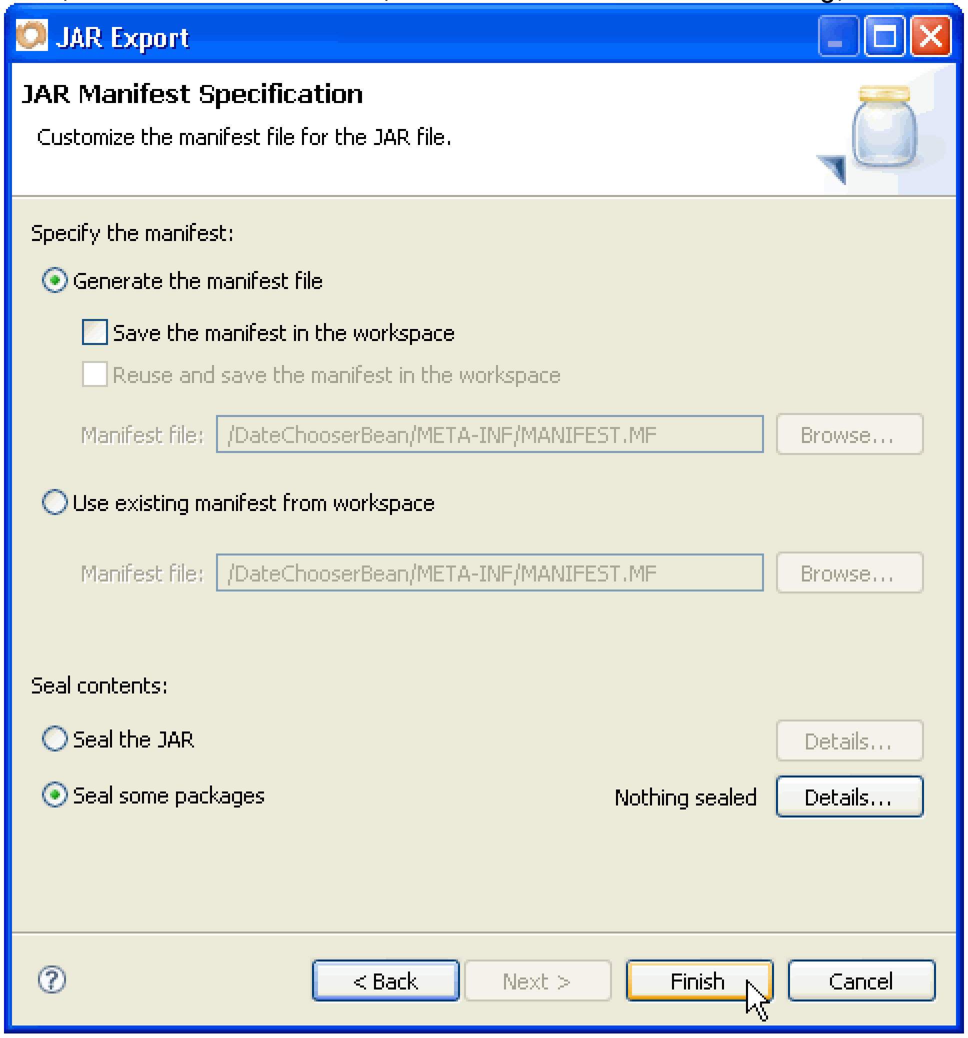
- [✓] Export class files with compile errors
- [✓] Export class files with compile warnings
- [ ] Create source folder structure
- [✓] Build projects if not built automatically
- [✓] Save the description of this JAR in the workspace

Description file: /WhoisPlugin/whois.jardesc

Leave the 2 “Export class files...” checked, and check the “Save the description of this JAR in the workspace”. You will need to use the browse button to navigate to your “WhoisPlugin” project, and give a name to your definition, I chose “whois” and Eclipse automatically added the “jardesc” extension.

NOTE: What is nice about this option is that the next time you will want to deploy your jar (with modified sources for examples), all you will have to do is right-click on the file “whois.jardesc” in the Package explorer and choose “Create JAR” in the menu. A great time saver that will dispense you to go through all the Export dialogs each time you change something in your plugin.

Now, when we click “Next >”, we are faced with the final Dialog, the JAR Manifest dialog:


JAR Export

JAR Manifest Specification

Customize the manifest file for the JAR file.

Specify the manifest:

(●) Generate the manifest file

[ ] Save the manifest in the workspace

[ ] Reuse and save the manifest in the workspace

Manifest file: /DateChooserBean/META-INF/MANIFEST.MF

( ) Use existing manifest from workspace

Manifest file: /DateChooserBean/META-INF/MANIFEST.MF

Seal contents:

( ) Seal the JAR

(●) Seal some packages — Nothing sealed

We can leave it like it is above. Eclipse will generate a default “MANIFEST.MF” file into a folder “META-INF” which is supposed to contain information about the compiler used, the java version, the version of the jar, etc... It can also tell java which class contains a main() method (for standalone programs), and in case of Java Beans, it can tell which class to look for beans.

In our case, since we are building a plugin and not a bean, we can use the default and let Eclipse deal with that.

Now click “Finish” on our dialog, our whois.jar should have been copied right into the Servoy plugins/ folder and all you need to start using your plugin is to restart Servoy!

That’s it. You’ve done it! Congratulations! Let’s just wrap this tutorial by building a small test solution.

O. Testing the plugin

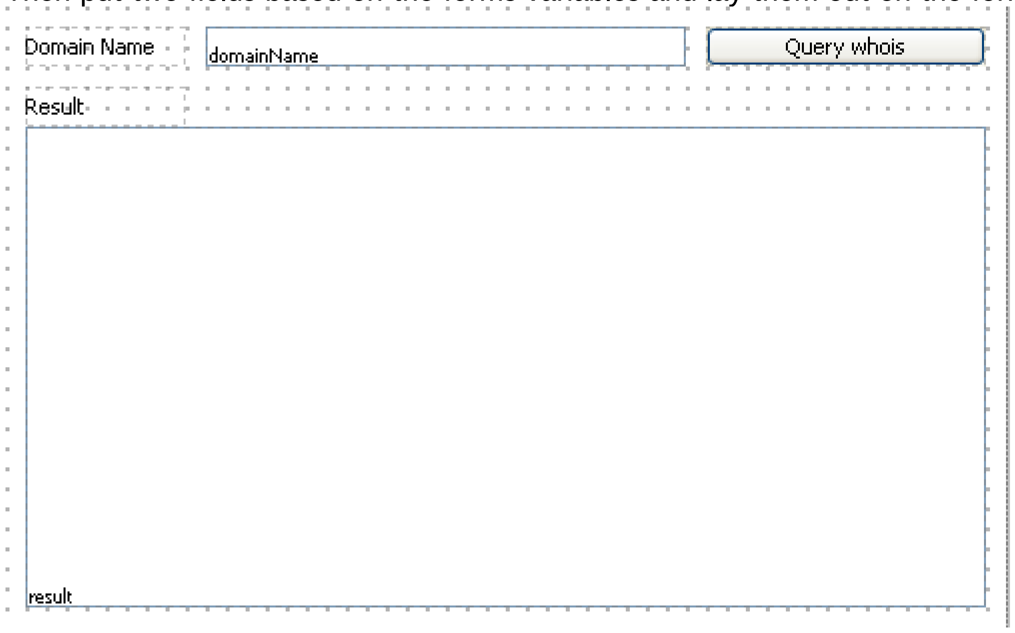
Back in Servoy, this time using it “regularly”, let’s create a sample solution, I have chosen to name it “WhoisTest”.

Let’s add a “test” form, you can build it on any table you want, we are not going to use the table anyway!

In the script editor, let’s add 2 variables:
```
var domainName = "";
```
and
```
var result = "";
```

and a simple function:
```
function ACTION_queryWhois()
{
	if (domainName != null && domainName.length > 0) {
		result = plugins.whois.query(domainName, "whois.internic.net");
	}
}
```

Then put two fields based on the forms variables and lay them out on the form, this is how mine looks like:

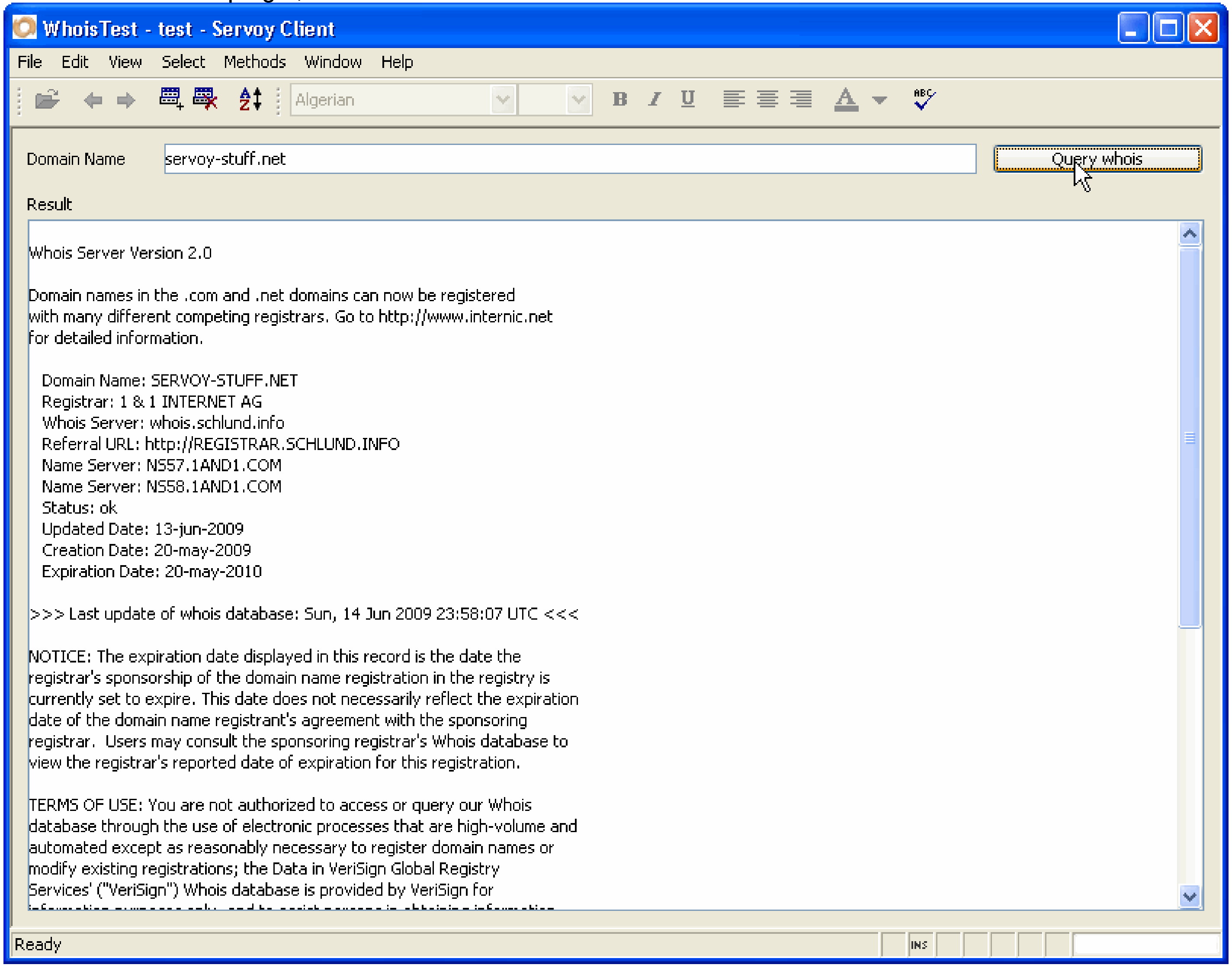


Add a button and attach the “ACTION_queryWhois” to the onAction event.
Set the “result” field to a bigger size and choose “TEXT_AREA” as a displayType.
Optionally, set the navigator of the form to “-none-“.

I have put the sample solution here:
http://servoy-stuff.net/tutorials/utils/t01/WhoisTest.servoy

Download (rename to WhoisTest.servoy if IE insisted it is a zip file), and import via the servoy-admin “Solutions” page.

It’s time to test our plugin, first in the smart client:

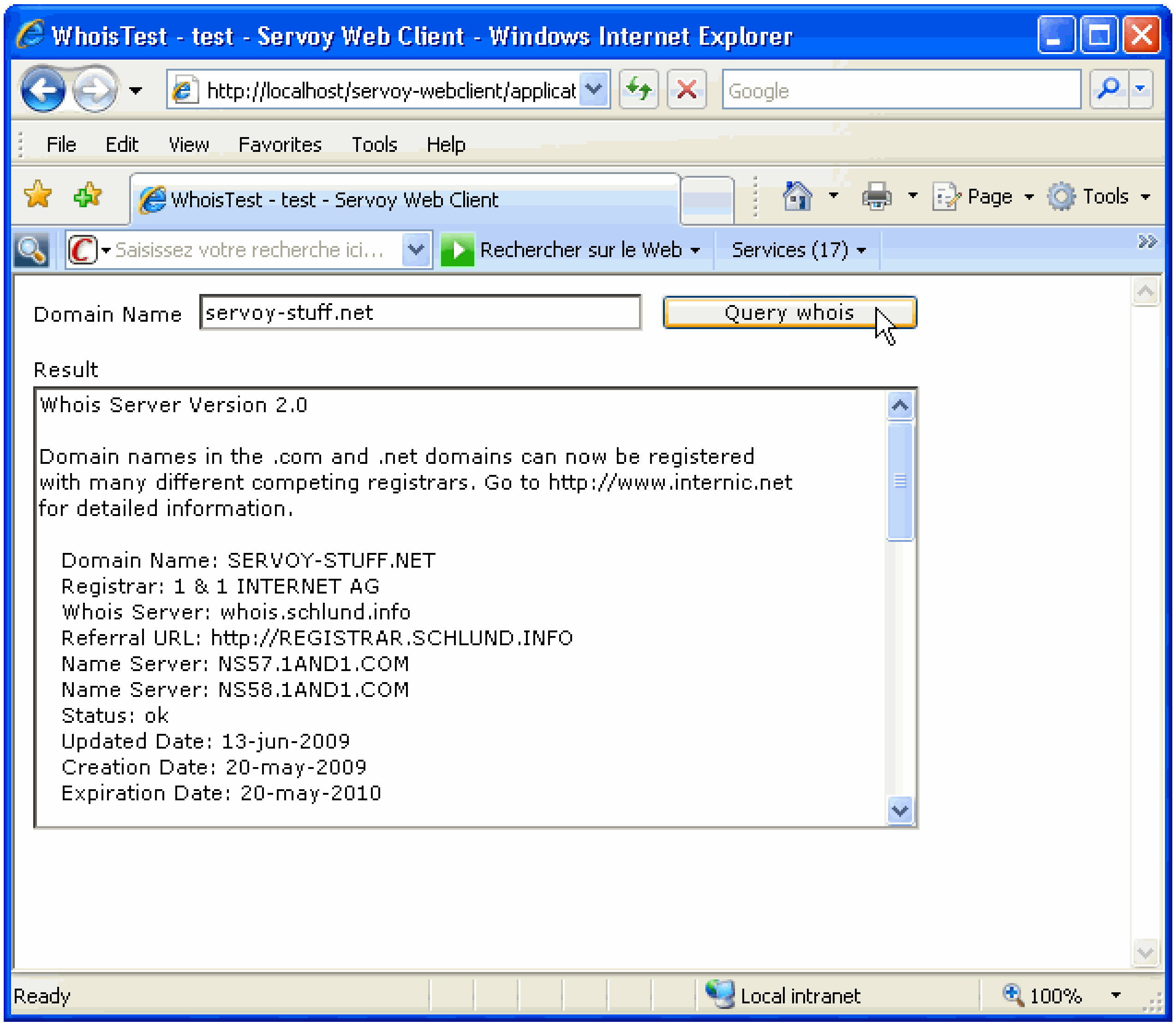

WhoisTest - test - Servoy Client
File Edit View Select Methods Window Help
Algerian
Domain Name
Query whois
Result
Ready
INS

servoy-stuff.net

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: SERVOY-STUFF.NET
   Registrar: 1 & 1 INTERNET AG
   Whois Server: whois.schlund.info
   Referral URL: http://REGISTRAR.SCHLUND.INFO
   Name Server: NS57.1AND1.COM
   Name Server: NS58.1AND1.COM
   Status: ok
   Updated Date: 13-jun-2009
   Creation Date: 20-may-2009
   Expiration Date: 20-may-2010

>>> Last update of whois database: Sun, 14 Jun 2009 23:58:07 UTC <<<

NOTICE: The expiration date displayed in this record is the date the
registrar's sponsorship of the domain name registration in the registry is
currently set to expire. This date does not necessarily reflect the expiration
date of the domain name registrant's agreement with the sponsoring
registrar.  Users may consult the sponsoring registrar's Whois database to
view the registrar's reported date of expiration for this registration.

TERMS OF USE: You are not authorized to access or query our Whois
database through the use of electronic processes that are high-volume and
automated except as reasonably necessary to register domain names or
modify existing registrations; the Data in VeriSign Global Registry
Services' ("VeriSign") Whois database is provided by VeriSign for
```

Working great!

And in the web client:


WhoisTest - test - Servoy Web Client - Windows Internet Explorer
http://localhost/servoy-webclient/applicat
File Edit View Favorites Tools Help
WhoisTest - test - Servoy Web Client
Domain Name
servoy-stuff.net
Query whois
Result
Ready
Local intranet
100%

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: SERVOY-STUFF.NET
   Registrar: 1 & 1 INTERNET AG
   Whois Server: whois.schlund.info
   Referral URL: http://REGISTRAR.SCHLUND.INFO
   Name Server: NS57.1AND1.COM
   Name Server: NS58.1AND1.COM
   Status: ok
   Updated Date: 13-jun-2009
   Creation Date: 20-may-2009
   Expiration Date: 20-may-2010
```

Success!

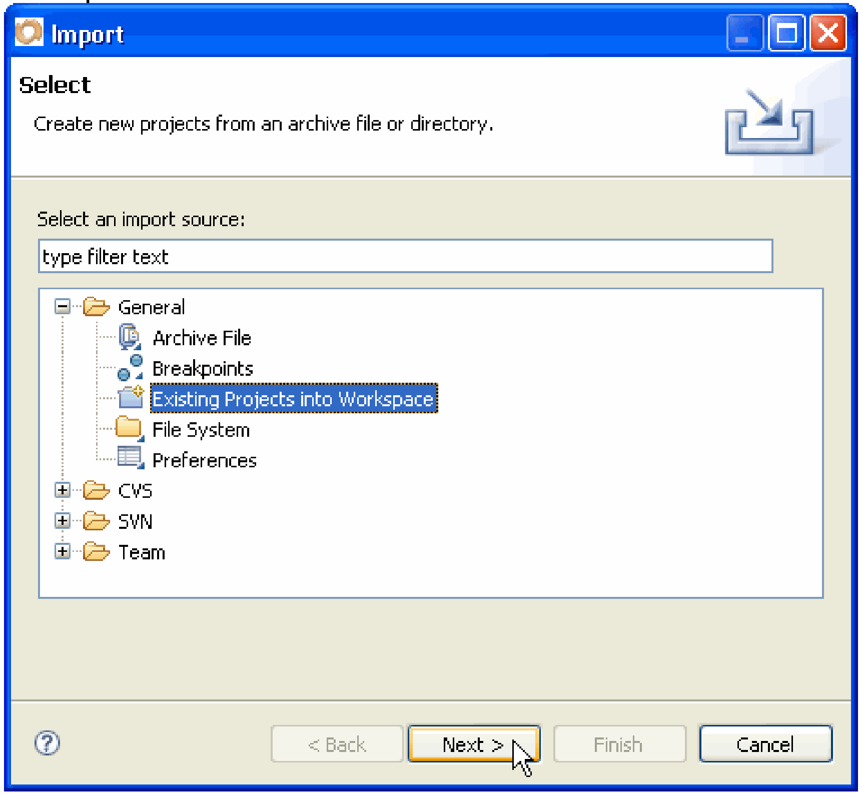
I hope you enjoyed following along this walkthrough tutorial, that you learned a few things here and there, and that now you feel more curious about this "dark side" of Servoy, that is not so dark really ;-)

I also hope that now you feel ready to create some new and exciting plugins for Servoy that you will hopefully release as Open Source, so that others will be able to use them and contribute to them for the better of the Servoy community.

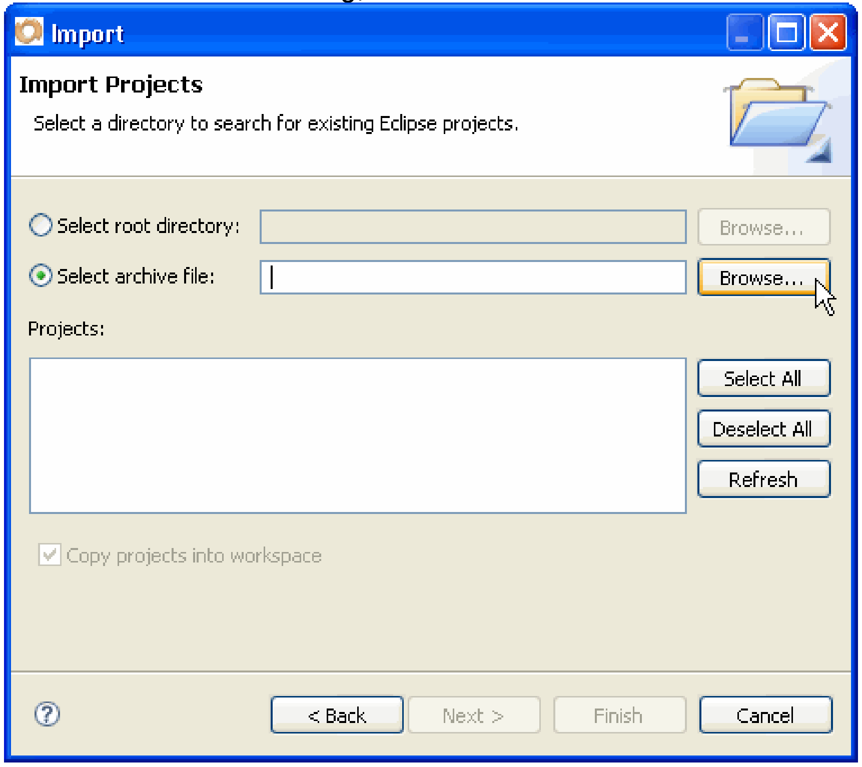
Send them to me and I will place them on the Servoy Stuff site if you want, with due recognition of your contribution. Feel also free to send me your comments/questions/suggestions about these tutorial or other Java/Servoy topics, on the Servoy forum of by using the contact form of the Servoy stuff website. http://www.servoy-stuff.net/contact.php you can also post comments on the blog, it is a free space for everyone to give their free opinions : http://www.servoy-stuff.net/blog

Patrick Talbot
Servoy Stuff
2009-06-14

PS: I will share the whole project as well as the finished jar in the "downloads/plugins" section of the Servoy Stuff site, (you can simply use the “File > Import...” menu and choose “Import existing project into Workspace”:



And then in the next dialog, check “Select archive file” and browse to the downloaded zip project:



Etc...