ADE Guide

Matthias Scheutz, Human Robot Interaction Lab

August 18, 2011

Contents

1 What is ADE?

ADE (Architecture Development Environment for Virtual and Robotic Agent) is a software framework for developing and running distributed robotics applications. ADEs core contains infrastructure for facilitating the discovery, communication, and maintenance of software components, but may run on different host computers, but can be viewed as a single entity through ADE. ADE also contains a sizable collection of existing software components, known as Servers, for support of robotic tasks (e.g., vision and laser servers, action managers, natural language processing components, simulation environments, and so forth).

The following document will attempt to provide an overview of ADE. Documentation for particular servers is best found by browsing through the JavaDocs of the corresponding server, and/or glancing at the ADE wiki, but this document should provide a good first start.

2 ADE Overview and Terminology

Each ADE component is known as a Server. Servers perform whatever task they were designed for (for example, fetching laser readings from an URG laser unit), and are responsible for defining their communication protocols, such as methods for obtaining the server data. Servers can obtain references to other servers (more on that below) in order to exchange information or requests (e.g., an Action Manager server may fetch information from the Vision Server, in order to send a motion request to a robots Motion Server).

One special ADE component is the ADE Registry. The Registry initiates the discovery and reference-fetching of servers, supports server recovery (in case of failure), provides a centralized GUI view into the entire ADE system, enforces security, and provides means for saving and opening entire ADE configurations just to name a few functions. Thus, each running ADE configuration must have at least one (typically just one) ADE Registry.

ADE makes an important distinction between the Interface and the Implementation of a server. The interface defines the communication protocol for the server: only methods defined in the interface will be exposed to other servers. The implementation, on the other hand, provides the actual code that runs the server, along with a fulfillment of the communication protocols that had been defined in the interface. By convention, interfaces typically have the name ”Server” in them, e.g., VisionServer, while Implementations typically mirror the interface name and must end in Impl (e.g., VisionServerImpl).

3 Installing ADE

The core of ADE is written in Java, targeted at Java 1.6. As such, ADE should be platform-agnostic though many of the supporting packages are not. ADE is best supported by Linux, (or UNIX derivatives, e.g., Mac OS X), and offers some limited support for Windows (in the case of Windows, a simulation environment like Cygwin would work best). Certain components of ADE are written in C++ and/or other languages (for example, the Vision server).

Most classpaths and references in ADE are relative to the original ADE directory. Thus, when installing ADE, you should only need to adapt the paths and classpaths once. When installing supporting packages

(e.g., Festival, Sphinx), placing the packages in the same top-level ADE directory should automatically resolve a lot of the references for you. One thing to note is that ADE servers use themselves as their credentials when making calls; so, if you're having trouble registering or obtaining a reference to some other server, one thing to check is the classpaths of the respective servers *and* the registry.

4 Compiling ADE

The officially supported approach for compiling ADE is via a shell script, mkade, located in the top-level ADE directory. The script will automatically fill in compilation details and find necessary references. Running ./mkade as is will attempt to compile all of ADE (including rarely-used servers and servers that may be out of date), so ./mkade small may be a better starting option. The script can also be used to compile a particular component of ADE (e.g., ./mkade vision). For a description of other options, run ./mkade help.

ADE can also be configured to integrate with a development environment, such as Eclipse or NetBeans. This can be especially helpful when navigating code or writing new servers. Some servers will easily compile and run right out of ADE, but servers with external references and packages might still be best compiled externally with ./mkade.

5 Running ADE

As mentioned in the ADE Overview section, each running ADE configuration must have at least one (usually just one) running Registry. The Registry can be used to start an entire ADE configuration (more on that in a sub-section below), but, especially when debugging, it is often easiest to start one server at a time, one in each terminal window.

5.1 Running the ADE Registry

ADE provides a helper script, runaderegistry, that will attempt to piece together a java command line that starts out the ADE Registry. The script can simply be used as follows: "./runaderegistry".

The first time that the script is run, the script also creates a "host" file, under tmp/your_username.host, which defines your local machine as a host for the ADE configuration. Later, if your ADE configuration spans across multiple machines (as is often the case when running a motion and vision server on the robot's laptop, and the registry, action manager, and natural language processing on a separate desktop), you will need to adjust the host file to include additional computers as well (see the host files definition a few sub-sections below). An alternative to adding hosts to a hosts file is to pass the host IP addresses on the command line using the "-o ¡hosts¿" parameter, where ¡hosts¿ is a space-delimited list of the addresses that the registry should accept as legitimate.

When running the registry with a "-g" or "–GUI" command-line option, the ADE Registry provides an comprehensive visualization of the entire ADE System, including the ability to see the visualizations of other servers, launch new servers, shut down existing servers, see server information, and save and open configuration files. It can be very helpful for getting started with ADE.

To view a description of options and examples for running the Registry, try "./runaderegistry -help".

5.2 Running ADE Servers

As with the registry script, ADE provides a helper script for running individual ADE Servers. For some simple servers, the script does little more than invoke Java on the specified server type; for others, the script fills in required jar files and java VM parameters. Thus, the script is very convenient for running servers from the command-line, though it is also possible to run ADE servers without it (especially if running servers directly out of an IDE, like Eclipse).

To use the "runadeserver" script, open a terminal window and type "./runadeserver", followed by either the path or the canonical java name of the server (e.g., "`com/adesim/SimPioneerServer`", or "`com.adesim.SimPioneerServer`"). You can also pass in any command-line options to the server simply by appending them after the server name.

Some command-line options are common across all ADE components. For example, for servers that provide a visualization option (ADE Registry, the ADESim simulation environment, Vision Server, Laser Server, and others), "-g" or "–GUI" will display the server's visualization windows. Other command-line options are server-specific (for example, specifying which device to use in the Vision Server). To view the list of all available command-line options for a server, both generic and specific, add a "-help" option to the server's commandline or to the "runadeserver" call.

5.3 Using ADE with an IDE

With fairly minimal effort, it is possible to configure ADE to work with an IDE, such as Eclipse or NetBeans. Working within an IDE carries the advantages of auto-completes, easier navigation, variable inspection during debugging, and other advantages. (NOTE: running servers through an IDE only makes sense for servers that you are working on; for something like the ADERegistry, it is still easiest to just launch it from the commandline).

Use the following steps as a quick guide, and see more details on the "ADE + Eclipse" page of the HRI-lab wiki.

- *Most importantly*: In order for ADE to work with an IDE the project folder must be used as a root for both sources and class files. By default, some IDEs such as Eclipse will want a separate "src" and "bin" directory - in the case of ADE, do not let them!
- Next, configure the build path (Eclipse: right-click on the ade project, "Build Path" $\Rightarrow$ "Configure Build Path" $\Rightarrow$ "Libraries"). Add any jars within the "core" folder of the top-level ADE directory. Depending on the server you are working with, you may also need to add other jars from the "jars" folder.
- To run/debug servers from within the IDE, open a Run Configuration dialog. For the main class, type in "ade.ADEServerImpl" (*regardless of the server that you are running*). In the arguments tab, specify any command-line options (such as "–GUI"), *along* with the magic ADE + IDE parameters: "-l 127.0.0.1 -r 127.0.0.1" (this specifies that the server is located on IP address 127.0.0.1, as is the

local registry). Finally, under the Java Virtual Machine arguments, add *"-Dcomponent=*" followed by the canonical name of the server that you want to run, such as "`-Dcomponent=com.adesim.ADESimEnvironmentServer`".

5.4 Config and host files

5.4.1 Config files

If you want the registry to start your servers for you, you need to tell it something about the servers you need as well as the host machines available on which to run them. These are specified in files that are (by convention) placed in the config directory. Listed below are the possible syntax elements, but keep in mind that many of them are optional, so the typical config and host files are not so complicated. At a minimum, the 'host,' 'type,' and 'startdirectory' items should be listed, and 'serverargs" and 'userclasspath' are typically needed, as well.

Format and valid options:

```
STARTSERVER
host            IP address or hostname
port            the RMI port (default=1099)
type            server class (e.g., com.test.TestServer)
name            assigned name (optional)
groups          the groups this server belongs to (optional)
conn            number of connections supported (default=10)
tmout           reaper period
acc             access rights
startdirectory  fully qualified directory
userclasspath   server-specific classpath (space delimited)
javavmargs      server-specific java vm arguments (space delimited,
                no leading dash)
serverargs      server-specific command-line arguments
restarts        number of recovery attempts (default=0)
onlyonhosts     IPs on which server may execute (space delimited)
devices         devices required (space delimited; host must
                supply listed devices)
configfile      configuration file (optional)
credentials     additional permissions
debug           debugging level
ENDSERVER

You may also add "pause" commands between servers.
    "pause 1000" = 1 second.
```

Once you have compiled a config file, run it by starting the ADE registry with a -f (–file) option, followed by a file name. For example:

```
./runaderegistry -f config/my_test.config
```

5.4.2 Host files

Hosts are specified similarly in ".hosts" files, with information about what capabilities each has. As mentioned in the "Running the ADE Registry" section above, when you first run the ./runaderegistry script, the script will automatically generate a host file, placed under tmp/your_username.host. If your ADE configuration later spans across multiple machines, adjust the host file to include additional computers as well.

All entries except the IP address are optional and will be filled with default values.

Format and valid options:

```
STARTHOST
ip            IP address or hostname
os            operating system name
cpus          number of CPUs (default=1)
cpumhz        CPU speeds (default=850)
memmb         available memory (default=256MB)
adehome       ADE home directory
tempdir       temporary directory (e.g., /tmp)
javadir       Java home directory
javabin       Java executable
javacompiler Java compiler
shell         command-line shell (e.g., /bin/sh)
shellargs     shell arguments (e.g., -c)
ping          ping command (e.g., /bin/ping)
pingargs      ping arguments (e.g., -c 2)
rsh           remote shell command (e.g., /bin/rsh)
rshargs       remote shell arguments (e.g., -X -A -n)
rcp           remote copy command (e.g., /bin/rcp)
rcpargs       remote copy arguments (e.g., -X -A -n)
ssh           ssh command (e.g., /bin/ssh)
sshargs       ssh arguments (e.g., -X)
scp           remote secure copy command (e.g., /bin/rcp)
scpargs       remote secure copy arguments (e.g., -A -n)
sshlogin      ssh login username
ps            process information command (e.g., /bin/ps)
psargs        process information command (e.g., -f)
devices       interfaces supported on host
ENDHOST
```

6 Writing your own ADE Server

In addition to using the pre-built ADE Servers, you may want to write your own. Writing ADE Servers is fairly straightforward, and essentially involves sub-classing the core ADEServer and ADEServerImp classes for your own server, and utilizing a few of ADE's APIs and conventions. In return, you get "free" server-to-server communication, recovery, GUI framework, and other benefits.

Before you attempt to write you own server, be sure that you can run ADE (e.g., start ADERegistry and some other server, for example `com.adesim.ADESimEnvironmentServer`). You may also want to

configure ADE to work with an IDE, such as Eclipse, as doing so will greatly simplify your task of matching interfaces and auto-completing methods.

A template Server interface and Implementation can be found under the com/template package. The example server is very thoroughly documented, and shows exactly where to put in the necessary code, how to broadcast methods and make remote calls, and how to compile and run the server.

6.1 Some basics of the general ADE server-writing philosophy

- All "services" (Laser, Vision, GPS, etc) are viewed as Servers.
- Each server has an *Interface* and an *Implementation* file.
  - By convention, the interfaces have the word "Server" in them. For example, LRFServer (laser range finder server), or ADESimEnvironmentServer. Note the capitalization.
  - Also by enforced convention, the implementation files need to be named the same as the interface, and must end with the word "Impl".
- All server interfaces must inherit from ade.ADEServer, and all server implementations must inherit from ade.ADEServerImpl
- The interface contains method calls that other servers are allowed to make to THIS server. For example, a laser server might expose a method to get all of the distances, or to return how many laser beams the device supports.
- Servers collaborate via the following mechanism:
  - Typically at startup, servers will request references to some other servers that they want to collaborate with (ie: ActionManager might request reference for a Laser server).
  - Subsequently, servers will call method names and arguments on references to servers.

6.2 Hello World, bare minimum

The following example shows a primitive Hello World. It assumes (as is the case with most servers) that the server is located in its own package, such as `com.helloworld`.

6.2.1 The Interface

In the newly created package, create the Server interface (i.e.: `HelloWorldServer`). On the interface declaration line, extend ADEServer. Your interface should now look like:

```
package com.helloworld;
import ade.ADEServer;
public interface HelloWorldServer extends ADEServer {
}
```

Since, at bare minimum, the server does not have to talk to any other servers, the empty interface is complete. Note that the interface is always required, even if it supports no methods (though that's hardly ever the case).

6.2.2 The Implementation

Within the same package, create a new class, `HelloWorldServerImpl.java`. The new class must extend *`ade.ADEServerImpl`*, and implement the newly-created interface. Do not forget *both* the extension and the implementation (otherwise, Java or you IDE may suggest that you must override several dozen methods, whereas in reality you don't!) Thus, so far, you should have

```
public class HelloWorldServerImpl
     extends ADEServerImpl implements HelloWorldServer
}
```

Even if you do not need to do anything in the constructor, a constructor declaration (with "`throwsRemoteException`" is still required. Beyond that, there are a dozen or so required methods (you can find them, with their explanations, in `com/template/TemplateServerImpl.java`). Their functions are also are outlined below:

Always used methods:

- The constructor: Required, even if just for a "super();" call. Use the constructor to establish connections to devices, request other servers, initialize data structures, etc.
- updateServer: This method is called every X milliseconds. Most servers will use this as a "tick" command to do whatever they are designed to do (e.g., get readings, send out action commands, etc).
- localServicesReady: If the server must wait for some event (a connection to a device or another server, for example) before it can be deemed operational, this is where the logic for replying "true" or "false" would go.
- localshutdown: Performs actions (if any) upon being asked to shutdown (such as when a user presses ctrl-c in the terminal). Servers will typically use this to cleanly close any log files and connections to devices.
- additionalUsageInfo: Returns a helpful string with the server's usage information (what additional parameters it accepts, etc). This information is displayed whenever the server is run with a -help flag, or an invalid command-line argument (see "ADE Server Parameters" section).
- parseadditionalargs: Parses additional arguments, and returns false if encounters an argument that the server does not know what to do with.(see "ADE Server Parameters" section).

Less-often-used (though still required to be present) methods:

- updateFromLog: used if the server supports playing back from a log file.

- clientConnectReact, clientDownReact, serverDownReact, serverConnectReact: For servers that need to keep track of connections to or from themselves, these methods provide callbacks for connection and disconnection alerts. If these events are not immediately relevant to your server, you can leave them blank or, in the case of non-void functions, return false.
- localrequestShutdown: adds a credentials check to see if the Object requesting the shutdown is allowed to shut down this server.

At bare minimum, the HelloWorld server will simply output "Hello World" to the terminal every cycle (by default, 10x/sec). Here is the complete code listing:

```
package com.helloworld;

import java.rmi.RemoteException;
import ade.ADEServerImpl;

public class HelloWorldServerImpl
        extends ADEServerImpl implements HelloWorldServer {
    private static final long serialVersionUID = 1L;

    public HelloWorldServerImpl() throws RemoteException {
        super();
    }

    protected void updateServer() {
        System.out.println("Hello, world!");
    }

    protected boolean localServicesReady() {
        return true;
    }

    protected String additionalUsageInfo() {
        return null;
    }

    protected boolean parseadditionalargs(String[] args) {
        return true;
    }

    protected void localshutdown() { /* N/A */ }

    protected void updateFromLog(String logEntry) { /* N/A */ }

    protected void clientConnectReact(String user) { /* N/A */ }

    protected boolean clientDownReact(String user) {
        return false;
    }

    protected void serverDownReact(String serverkey,
            String[][] constraints) { /* N/A */ }
```

```
    protected void serverConnectReact(String serverkey, Object ref,
            String[][] constraints) { /* N/A */ }

    protected boolean localrequestShutdown(Object credentials) {
        return false;
    }
}
```

6.3 Running the HelloWorld Server

From a terminal window, opened to the top-level ADE folder, start a registry (”./runaderegistry”). Then compile the two Java files (if the server were to become a permanent part of ADE, you can add it in to the ”mkade” script), and run the server:

```
javac com/helloworld/HelloWorldServer*.java

./runadeserver com.helloworld.HelloWorldServer
```

A rapid stream of ”Hello World” should start piling up on your screen.

6.4 Beyond the Bare Minimum

6.4.1 Specifying update frequency

The default frequency is set to 100ms. This can easily be changed to any other amount by calling ”`this.setUpdateLoopTime(this,2000);`” in the constructor. Always pass a reference to ”this”, followed by number of milliseconds between ”ticks”.

6.4.2 Communicating with other servers

Many ADE servers rely on information from other servers for their operation. We can simulate this with having the ”Hello World” message be appended by something like the elapsed simulation time in an ADESimEnvironmentServer.

Importantly, when requesting a server type, remember that it’s the *interface* name that is being requested, *not* the ”Impl”. Also, rather than specifying the name as a string (which can change if the server is moved to a different package, or is renamed), it may be easiest to call ”getName()” on the type’s class (e.g., ADESimEnvironmentServer.class.getName()).

The simplest method simply keeps trying and trying to find a connection. First, create a private Object field for the reference (”private Object envServerReference;”). Then, while envServerReference is null, we’ll want to have the server keep trying to get a connection to the ”client”, ADESimEnvironementServer, sleeping for a few seconds between failed attempts.

```
public HelloWorldServerImpl() throws RemoteException {
    super();
    this.setUpdateLoopTime(this, 1000);

    while (envServerReference == null) {
        System.out.println("Establishing connection...");
        envServerReference = getClient(
                ADESimEnvironmentServer.class.getName());
        if (envServerReference == null) {
            // if didn't establish a connection
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                System.out.println("Can't sleep");
            }
        }
        System.out.println("Connection established!");
    }
```

Note that even if the server has to wait before a connection is established, updateServer() call will still continue to be called. Therefore, it is best to change updateServer and localServicesReady method as follows:

```
    protected boolean localServicesReady() {
        return (envServerReference != null);
    }
    protected void updateServer() {
        if (!localServicesReady()) {
            return; // nothing to do
        }
        System.out.println("Hello, world!");
    }
```

As for using the connection, it is done by calling a method that is in the interface of the callee server (in this example, ADESimEnvironmentServer). For example environment has a method "public String getElapsedTime() throws RemoteException;". To call this method from the HelloWorld server, one would do:

```
try {
    System.out.println("Sim time = " + (String) call(
            envServerReference, "getElapsedTime"));
} catch (Exception e) {
    System.out.println("Exception " + e);
}
```

A few things to note about "call":

- The "call" syntax is "call(reference, methodName, args...)". (0 or more arguments, comma-separated)
- "call" return an object, so casting is always necessary. That object can be a custom complex data structure, but it *must be Serializable*.

- ”call” throws a number of exceptions, which must be handled (either with a generic Exception, or specific ones).
- call” CANNOT accept ”null” arguments. Due to Java reflection and ADE’s infrastructure, the method would not be correctly identified, and would result in a failed call. A workaround is simple enough, though: if, for example, you need to pass a null array list, simple create and pass a brand new empty array list, rather than passing null.

6.4.3 Broadcasting the server’s own methods

In order for your server to perform actions or return information at another server’s command, you must write a public function in the server’s *interface*, which may be either ”void” or return a *Serializable* object (for example, primitives such as int and bool, basic objects like Strings, automatically serialized collections like ArrayLists, or your own custom serializable class). The method must also be declared as ”throws RemoteException” (a necessary formality).

An example interface declaration (the implementation is trivial) is as follows:

```
public void setHelloMessage(
        String newMessage)throws RemoteException;
```

6.5 Writing server visualizations

6.5.1 Overwriting getVisualizationSpecs()

To begin creating a visualization for your ADE server, create the following method within your server (it overrides ADEServer’s default response to this method, which is simply to return an empty data structure)

```
@Override
public ADEGuiVisualizationSpecs getVisualizationSpecs()
         throws RemoteException {

    ADEGuiVisualizationSpecs specs = super.getVisualizationSpecs();
    // ask the super-class for visualizations specs, which,
    //     at the very top of the class hierarchy, will just return
    //     an empty ADEGuiVisualizationSpecs data structure.

    specs.add( {visualization_name}, {visualization_class_type},
               {optional_startup_parameter}, {additional arguments...} );
    ...

    return specs;
}
```

The method must return an ADEGuiVisualizationSpecs datastructure. As for adding visualizations to the specs data structure, the add method takes the following arguments:

- *Visualization name*: The name should ideally be short and contain no spaces. Examples include ”camera”, ”lasers”, ”STM”, etc. The name is used in the server context menu of ADE’s SystemView, and in specifying the visualization by name on the command-line. The name must be UNIQUE within a particular server’s list of visualization.
- *Class type*: The visualization’s class object (i.e., VisualizationX.class)
- *Startup parameter (not required)*: Defines whether the GUI should always show, show on a –GUI flag, or show by explicit request only (for resource-intensive visualizations). If left off, the visualization will be assigned the default ”show on –GUI flag” parameter.
- *Additional arguments*: Any additional arguments that the visualization may want to take in its constructor.

When all is said and done, your overall getVisualizationSpecs() method might look something like this (note that the server is *not* limited to just a single visualization!):

```
@Override
public ADEGuiVisualizationSpecs getVisualizationSpecs()
            throws RemoteException {
    ADEGuiVisualizationSpecs specs = super.getVisualizationSpecs();
    specs.add("Lasers", LRFServerVis.class);
    specs.add("Camera", ADESim3DCameraVisualization.class);
    return specs;
}
```

6.5.2 Creating the Visualization window

Having overwritten getVisualizationSpecs, you must create a class for whatever classes you’re adding to the visualization specs. The class MUST inherit from `ade.gui.ADEGuiPanel` and accept an `ade.gui.ADEGuiCallHelper` as the first parameter in a constructor. Then, on the very first line of the constructor, you must call super with that very same `ADEGuiCallHelper` argument, along with a GUI refresh rate in milliseconds:

```
public class VisionServerVisSTM extends ADEGuiPanel {
    public VisionServerVisSTM(ADEGuiCallHelper guiCallHelper) {
        super(guiCallHelper, 100); // 100ms = 10x / second

        // TODO:  initialize the visualization display
    }
}
```

NOTE: visualization constructors can also take additional arguments, *after* the first ADEGuiCallHelper argument; in those cases, be sure to also add values for those arguments in the specs.add() method.

The visualization class must also implement a *”refreshGui”* method, to indicate what should be updated or redrawn every time that the GUI is refreshed (e.g., 10x/second)

```
@Override
public void refreshGui() {
    // TODO:  do whatever refreshing code (see below)
}
```

If your visualization panel displays the visualization by drawing (for instance, by overriding the paint method, or by *containing* a panel that overrides a paint method), here would be a good place to call this.repaint() (or innerPanel.repaint() – for an example, see VisionServerVisSTM). Alternatively, you may want to update existing components (such as labels) by simply calling their .setText() (or equivalent) methods.

One final thing to note: when the visualization is invoked via the -g/–GUI command-line, the visualization will *only* show up once the the server is "ready". So, be sure that you're properly responding in your server's "protected boolean localServicesReady()" method.

6.5.3 Other ADEGuiPanel methods worth overriding

In addition to refreshGui(), which is a required override, there are two more methods that may well be worth overriding:

- *public Dimension getInitSize(boolean isInternalWindow)*. By default, this simply turns around and calls this.getPreferredSize() on this panel. In many cases, that is exactly what you want – IF the panel will know how to determine its size properly (such as based on components contained within). If you're doing custom GUI drawing, however, the panel might think its preferred size is 0. In that case, you should override either the getInitSize, or, better yet, the getPreferredSize() of the panel. getInitSize can also be used for setting the size depending on whether or not the visualization is "an internal window" (e.g., lives inside of ADE's SystemView, and must fit alongside other windows, as opposed to being a JFrame on its own right, launched via "–GUI")
- *public String getInitTitle()*. Override it to set the title to something you want (no need to call super), otherwise it'll just be the server's name. Remember that you can always call setTitle as the application is running if you want to change the title (for example, to reflect the status of the server)

6.5.4 Communicating with the underlying server

Of course, the visualization is helpful only so long as it can communicate with the underlying server. Fortunately, the communication mechanism is built-in, and is handled via a "callServer()" function, which carries the same arguments as the ADEServer "call" method, EXCEPT a reference of the server (since the server is obviously the server for which the visualizer is instantiated). So if I you want to obtain the VisionServers's data for visualizing, simply do:

```
try {
    myVisualizationData = (Vector<VisionServerVisSTMData>)
                          callServer("getVisualizationData", 0.5);
} catch (Exception ex) {
```

```
    System.err.println("Could not communicate with vision server! \n\n" + ex);
}
```

The declaration of getVisualizationData is part of the VisionServer's interface, which accepts a confidence number as its argument. Note that the call helper can handle arbitrary amounts of arguments, so you're not limited by the amount of things you can pass. Also note that the callServer requires that it be put in a try-catch block. As for the method that gets called, the methods is declared in the underlying server (in this case, VisionServer) just the same way that any ADE method is declared, e.g.:

```
public Vector<VisionServerVisSTMData> getVisualizationData
          (double conf) throws RemoteException;
```

A few things to note:

- As with ADE calls, methods that are used to communicate with the server must be *in that server's interface*.
- Also as with all ADE calls, the return value of the method must be something serializable. E.g., it has to be a primitive type, an automatically serialized type (String, ArrayList, etc), or a a custom class that implements the Serializable interface.
- Remember that the method will get called 10x/sec, and that the data has to travel over the network. Thus, care should be taken to ensure that only relevant data gets sent (e.g., better to create an ArrayList of only the relevant information, rather than pass some huge existing structure). It is also better to group data into a single serializable datastructure, rather than make a dozen individual calls. If the amount of data is truly huge but little of its gets updated at a given moment (as is the case in ADESim), it may be best to keep some sort of history on the server end, and a counter on the visualization end, so that the visualization can get only "diff" updates, except at the beginning, or what it falls behind (note that, to reduce CPU load, visualizations stop receiving updates when they are minimized; so that would be one example of when, after a window has been de-minimized, it might need to receive all of the data again, rather than 1000s of "diff" entries). See `com.adesim.ADESimMapVis` for an example; but note that you probably won't need something quite so sophisticated this in all but exceptional cases.
- Note that communication *need not be one-way*: a callServer method can be used to *act upon* the server, such as to notify a server to update parameter X when a checkbox gets clicked. Remember, however, that since multiple visualizations can be running simultaneously, you'll want to re-load the state of the checkbox every time in refreshGui(), so that one person's checking of it is reflected in another person's visualization!

6.5.5 GUI utilities and Swing hints

- The ade.gui package contains Util.java and UtilUI.java, both of which (particularly the latter) contain a handful of useful GUI methods (putting a stack trace in a messagebox, creating a custom file chooser, etc).

- There is also an ade.gui.icons.IconFetcher class, that provides an easy way of retrieving an image from a chosen set of useful public-domain icons, http://tango.freedesktop.org/Tango_Icon_Library, found under ade/gui/icons/size16. To use the IconFetcher, simply do:

```
IconFetcher.get16x16icon("help-browser.png")
```

- To create a menu bar, set the Panel's layout to BorderLayout, and add a JMenuBar to its north. I.e.,

```
this.setLayout(new BorderLayout());
JMenuBar menuBar = new JMenuBar();
... initialize the menu...
this.add(menuBar, BorderLayout.NORTH);
```

- Remember that Panels with border layouts can be placed inside each other recursively, and therein lies their incredible handiness.