Web-Based Object Inspection

Dick Cowan and Martin Griss

Software Technology Laboratory
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304 USA

dick_cowan@hp.com, martin_griss@hp.com

December, 2001

ABSTRACT

To facilitate a natural web presence for selected objects we have combined a lightweight HTTP server, a generic reflection-based object inspector, and a suite of extensible presentation adapters. For many Java objects, under application control, there exists the need to view and possibly change their attributes at run time. An object's attributes are traditionally made available via a collection of public get and set methods. Using Java's reflection technology it is a simple process to discover these getters and setters. Combining this with a very lightweight HTTP server, we have demonstrated the ability to view and control virtually any run-time object using a standard browser, without the need to add any specialized code to the object. Given the results of an object's inspection, a customized HTML document is prepared for presentation by a presentation object, thereby affording extreme flexibility and customization in what is shown in the browser window. There exists a presentation adapter class designed to both provide the default presentation as well as enable simple variants to extend it. This facilitates powerful presentation control and information filtering with very small amounts of code. This technology lays the foundation for better monitoring, configuration, or control tools. It also provides an easy way to experiment with web-based user interfaces to objects, applications, agents, personal assistants and databases of all kinds early in development.

Keywords: Inspection, JavaBean, JSP, browser, http server, object web presence.

1 INTRODUCTION

Many Java applications and systems have a need for web-based interfaces, particularly in a distributed context. The need for such a capability first arose as part of our work to develop an industrial strength multi-agent system, in which the monitoring and control of the behavior of many agents, running on different platforms, motivated a web-based inspection solution, which we call "the Inspector." Using an MVC approach [Gamma et al., 1995], we combined a lightweight HTTP server, a generic reflection-based [REFLECT] object inspector, a suite of extensible presentation adapters, and specialized type decorators, to produce a flexible toolkit to rapidly build a web-presence for applications and even individual objects within the applications. As we developed the Inspector, we realized that the technology and approach was much more general, and allowed us to quickly produce a natural web presence for almost any Java object, under application control. We are able to associate a single Inspector with one or more objects running on the same JVM. The Inspector allows viewing, and possibly changing, the attributes of the objects, by associating a web page with each inspected object at run time.

In section 2 we provide a high-level design overview of our Inspection solution, and illustrate how it is used. In section 3 we show a variety of presentations that can be produced. Following that, in section 4 we describe the detailed architecture and some aspects of the implementation. In section 5, we discuss related technologies (JavaBeans™, Java Server Pages - JSP™ and HP ChaiServer™), while in sections 6 and 7 we discuss possible extensions and summarize our conclusions.

2 Design Overview

An object's attributes are traditionally made available via a collection of public get and set methods. Using Java's reflection technology it is a simple process to discover these getters and setters. Combining this with a very lightweight HTTP server,

we can view and control virtually any run-time object using a standard browser, without the need to add any specialized code to the object. Given the results of an object's inspection, a customized HTML document is prepared for presentation by a presentation object, thereby affording extreme flexibility and customization in what is shown in the browser window. We use a presentation adapter class that has been designed to both provide the default presentation as well as enable simple variants to extend it.

The overall architecture of our solution is based on the well-known Model/View/Controller (MVC)[Krasner & Pope, 1988; Gamma *et al*., 1995] design pattern, as described below:

1. Object Inspector – Implements the Model part of the design pattern. Implements an interface so as to make alternative implementations possible. Responsible for inspecting a Java object using reflection technology. Its two major activities are to inspect an object (using its getters) and return the inspection results or to update the object (using its setters) and return the outcome of each update.
2. Inspection Server – Implements the Controller part of the design pattern. Implements an interface so as to make alternate implementations possible. It is built on a very lightweight HTTP server that listens for new connections on a specified port number and spawns a thread to handle new connections. It only supports limited get and post operations. It maintains a tree of views of objects and index pages, where each view consists of the triplet of objects: (target object, presentation, label), and maps this triplet into a URL that will be used to reference this view. Views are stored on index pages and these pages may reference other index pages thus enabling a very manageable and flexible presentation. Each target is inspected with the Object Inspector and the results are transformed into an HTML document by the presentation object. An empty get request on the specified port results in either an index list of inspectable targets (see Figure 1), or if there is only one target the inspection results.
3. Presentation Adapter – Implements the View part of the design pattern. Implements an interface so as to make alternate implementations possible. This object produces the output HTML document for viewing by the browser. It is through this interface that the target object may be viewed or updated, or operations invoked. The design of the presentation adapter is such that it implements a structured hierarchy of override-able methods. The three layers to this hierarchy are as follows:
    1. createXXX – Methods that control the outermost logic to create a major portion of the output document. Example: createInspectionView to create the output of the inspection view of an object.
    2. writeXXX – Methods that control a significant portion of the output document. Example: writeInspectionRow that produces the output for a single inspectable item.
    3. getters – Methods used to return many configurable properties. Examples: getValueHeader returns the label for the value column, allowUpdates indicates if updates are allowed, and fetchMethodName retrieves the name of the attribute about to be displayed.

This layered presentation implementation facilitates powerful presentation control and information filtering with very small amounts of code. We have used this as a foundation for a variety of better monitoring, configuration, or control tools, for our agent-based systems, calendar and profile editors, and other applications. It also provides an easy way to experiment with simple web-based user interfaces to distributed objects, applications, agents, personal assistants and databases of all kinds early in development. The design of the presentation adapter has been shown to make it very simple to override the desired level of output production with a minimum of code, as we will illustrate in section 3.

As an example, we show a small code fragment that we will embellish as we describe additional features. Examples 1 and 2 show the small amount of code needed to start the Inspector on an object.

```
// Example 1.
// Start an inspector on this object, showing its public get and set methods
// Start the server on port 8080[1]
Inspection inspector = InspectionServer.getServer(8080, 10);
// Creates a default data presentation for this object. Title will be its class name.
```

[1] If port 8080 is already in use on this machine, it will try the next 10 successive ports, until a free port is found, or an IOException will be thrown.

```
inspector.add(this);


// Example 2.
// Add another object to the inspector.
Object test1 = new TestObject1();
// Adds a data presentation page for test1 with custom title
inspector.add(test1, null, "Test1 View");
```

3 Presentation

Using the presentation adapter, we discuss three alternatives.

3.1 Generic Presentation

Figure 1 shows the browser display of the top level index page when the controlling application has made six objects and two nested index pages available for inspection. Clicking on one of the links will display the desired object as formatted by its associated presentation adapter. The controlling application selects which objects are inspectable by the inspection server and the index view will display the target collection.

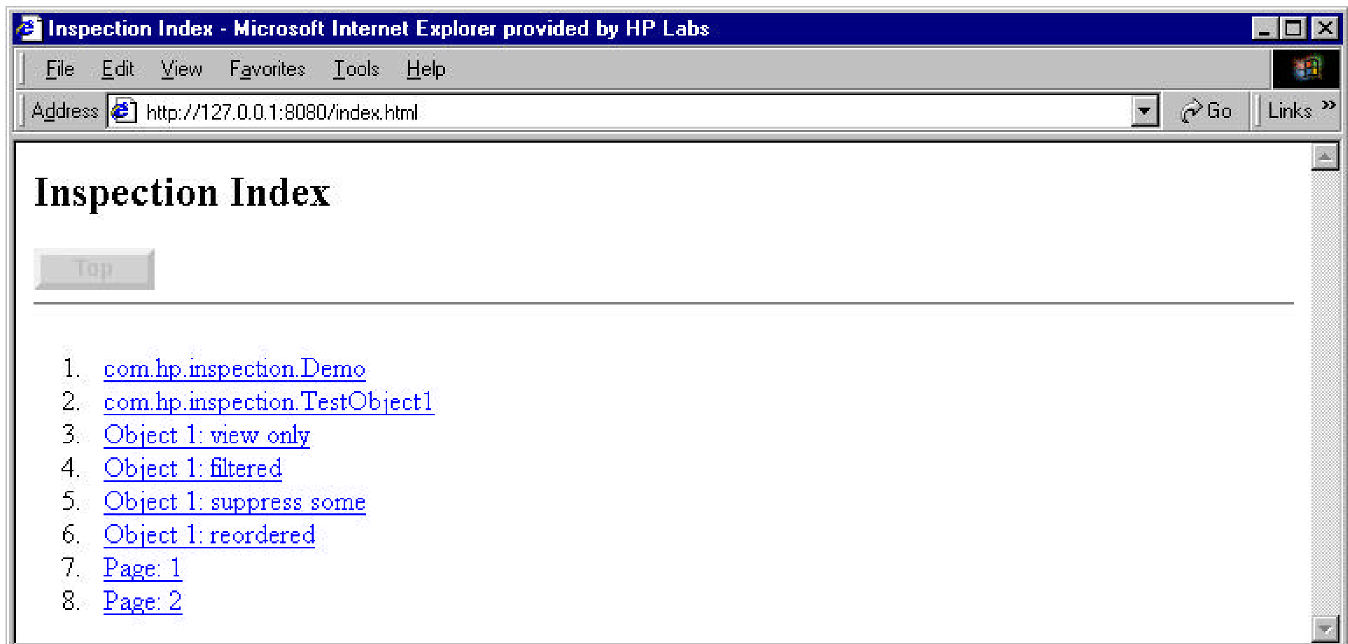

Figure 1 – Top-level index page

Figure 2 shows the browser view of the first data page (marked as item 1 in Figure 1) produced by a very generic presentation view. This presentation view is useful as a default for most objects. This presentation class has been designed for easy customization.

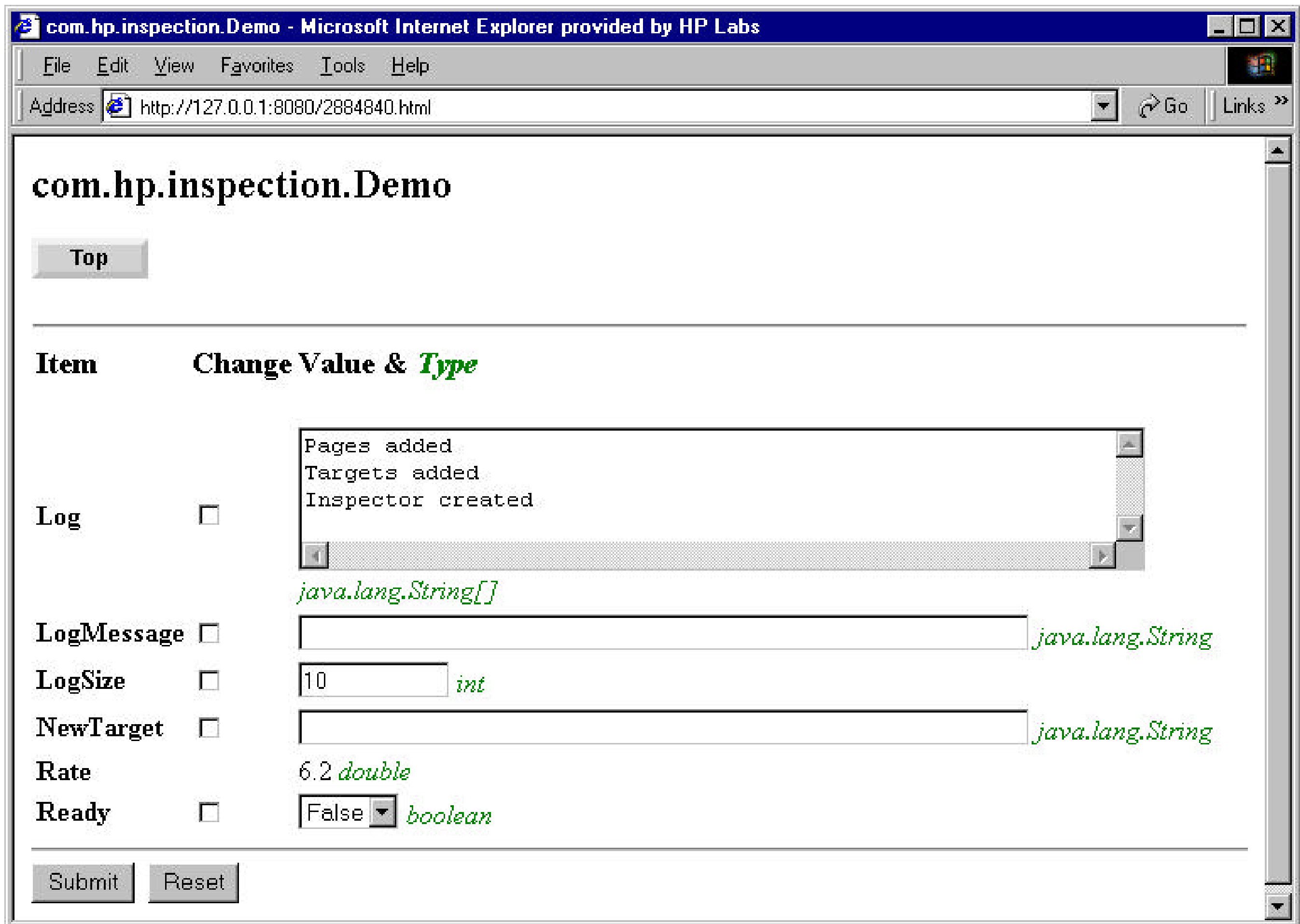

Figure 2 – Default presentation

This default generic presentation implementation has a standard header and footer with generated titles and buttons, and displays the data in three columns. The first "Item" column is the name of the getter minus its getter prefix (typically "get", "is" or "has"). The presentation lists the data alphabetically. The second "Change" column is a set of check boxes to be used to select which items you wish to modify when you click the "submit" button. This presentation has enabled updates if the target object has a setter for this item. The third column shows the value obtained for the target's getter as well as its type. If updates to the data value are permitted the data will be shown in a data entry box. The default title is generated from the target object's class name. The special presentation handling for String[] and boolean are handled by specialized type decorators.

3.2 View Presentation

By simply extending PresentationAdapter we can modify the default editing presentation, shown in Figure 3a, to one for viewing only, as shown in Figure 3b.

All that is required is to override one method as follows:

```
// Example 3. (Compare Figure 3a and 3b)
// Change the presentation by overriding a method in the generic presenter
inspector.add(this, new ViewOnlyPresenter(), null);
….
Class ViewOnlyPresenter extends PresentationAdapter {
  //
```

```
//      Override a presentation method to disallow all updates (default is to return true).
//      This also suppresses the submit and reset buttons, and all check boxes
//
            public boolean allowUpdates() {
                    return false;
            }
}
```

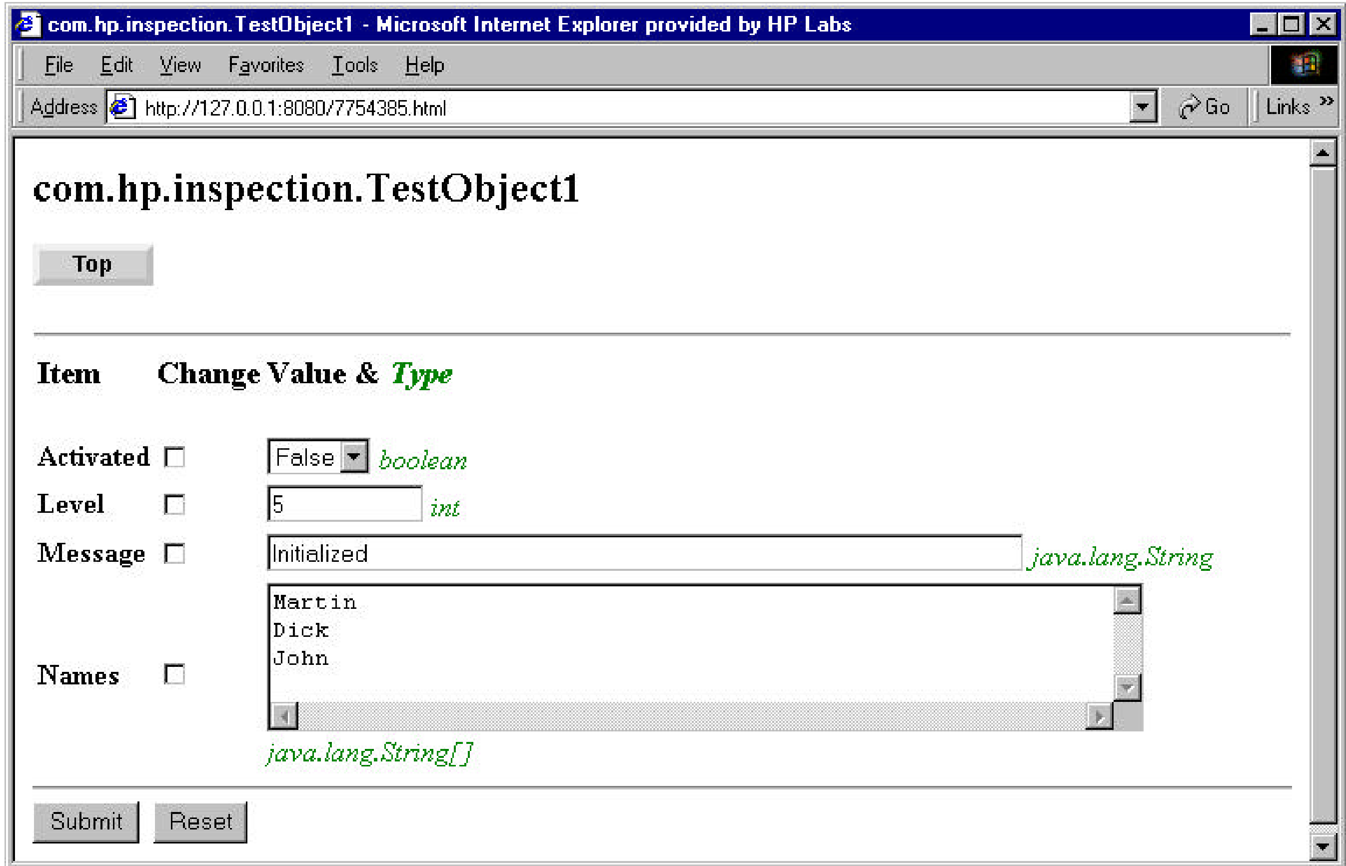

Figure 3a – Default "editing" presentation of TestObject1

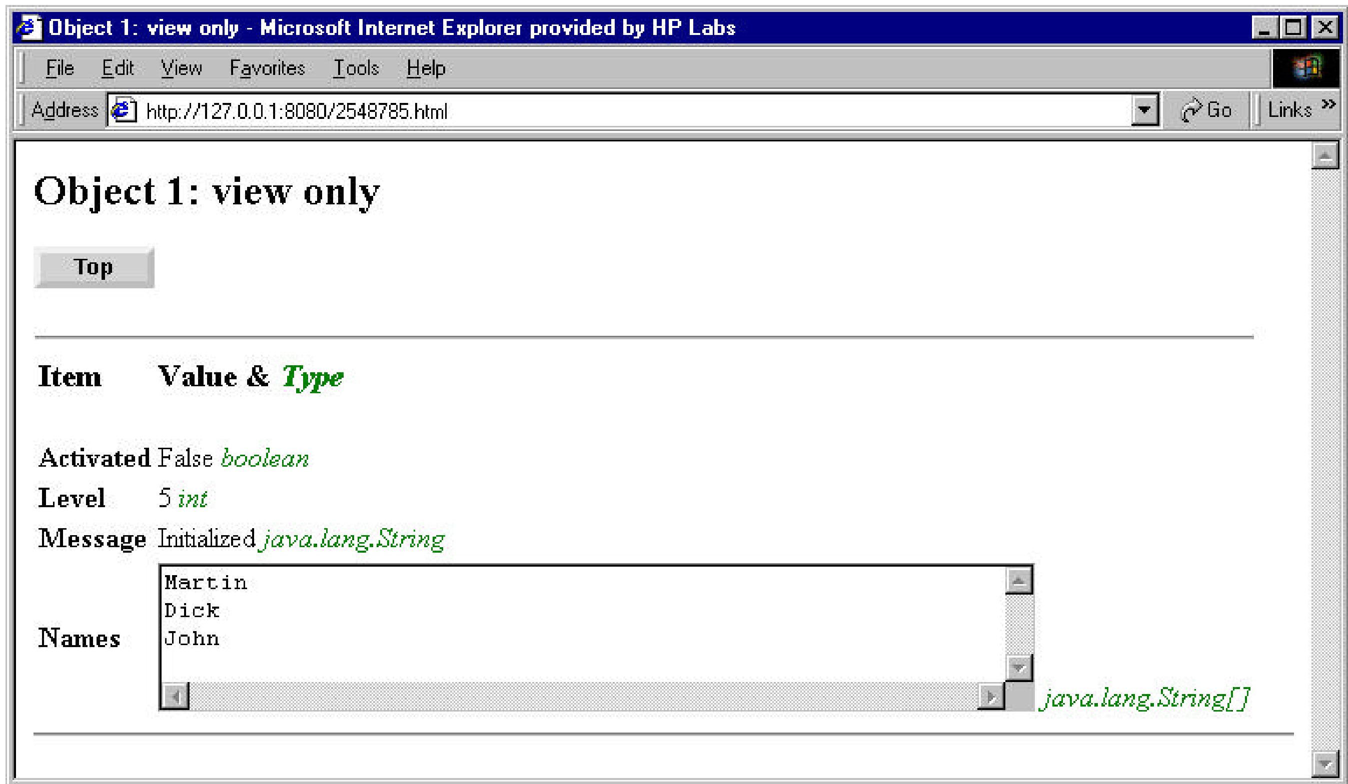

Figure 3b – Modified “view only” presentation of TestObject1

3.3 Other Presentations

In this section we will demonstrate or discuss other presentation implementations.

3.3.1 Change fonts, titles and subtitles, and buttons

```
// Example 4 (See Figure 4)
// Change title color, subheading and button label, returning text strings or HTML fragments.

// Change title color and emphasis
public String formatHeading(String aTitle) {
  return "<h2><font color=blue><i>" + aTitle + "</i></font></h2>";
}

// Add a subheading
public String getInspectSubHeading() {
  return  "<font color=red><i>Use Editor <u>VERY</u> carefully!!</i></font>";
}

// Change the submit button label
public String getSubmitLabel() {
  return "CHANGE!";
}
```

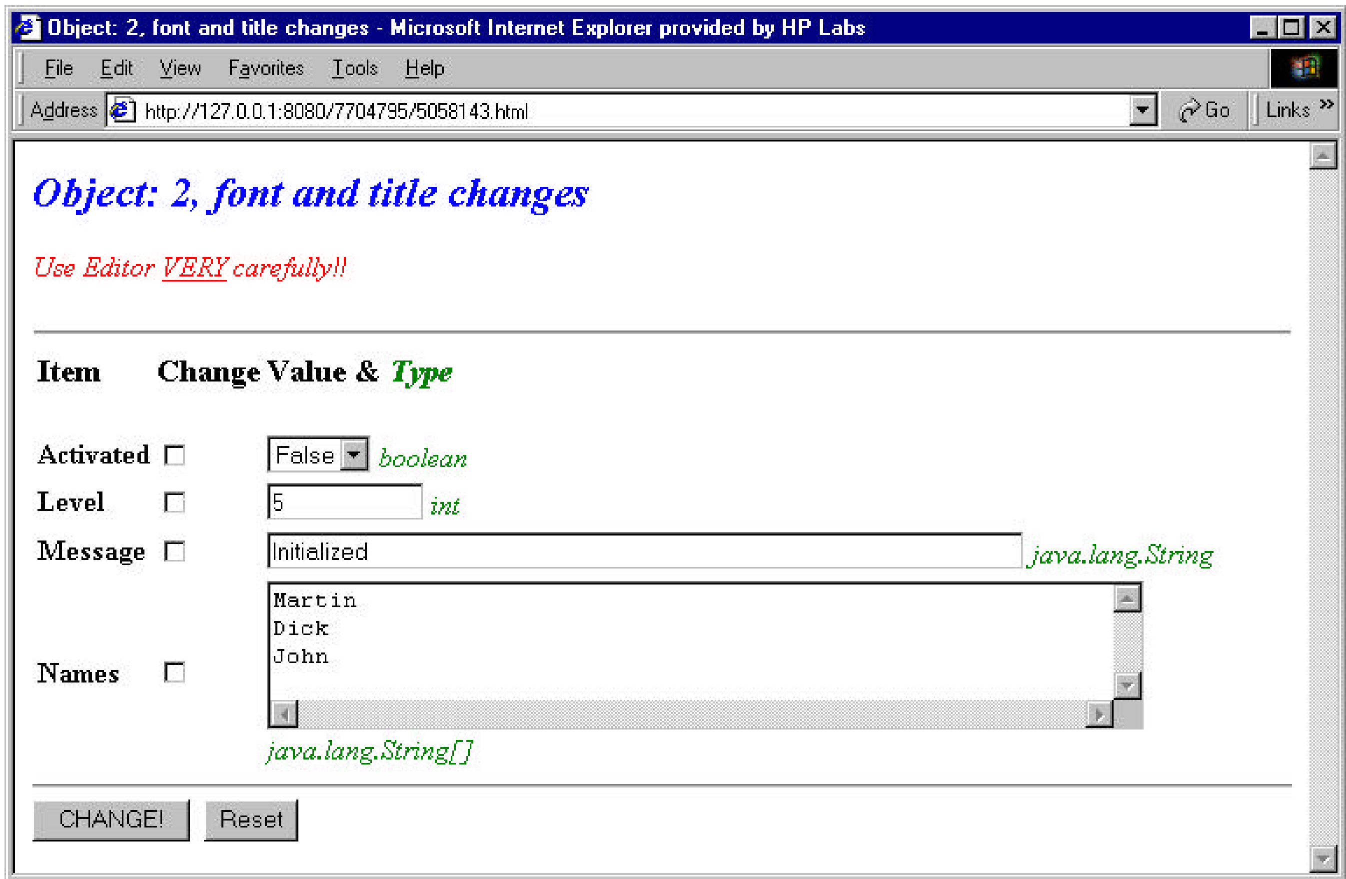

Figure 4 – Changing fonts and titles

3.3.2 Suppress display of specific method

```
//Example 5 (See Figure 5)
// Suppress display of Message, keep all others
public boolean isItemViewable() {
        String methodName = fetchMethodName();
        if (methodName.equals("Message"))   return false;
        if (methodName.equals("Activated"))   return false;
        return true;      // allow all others
}
```

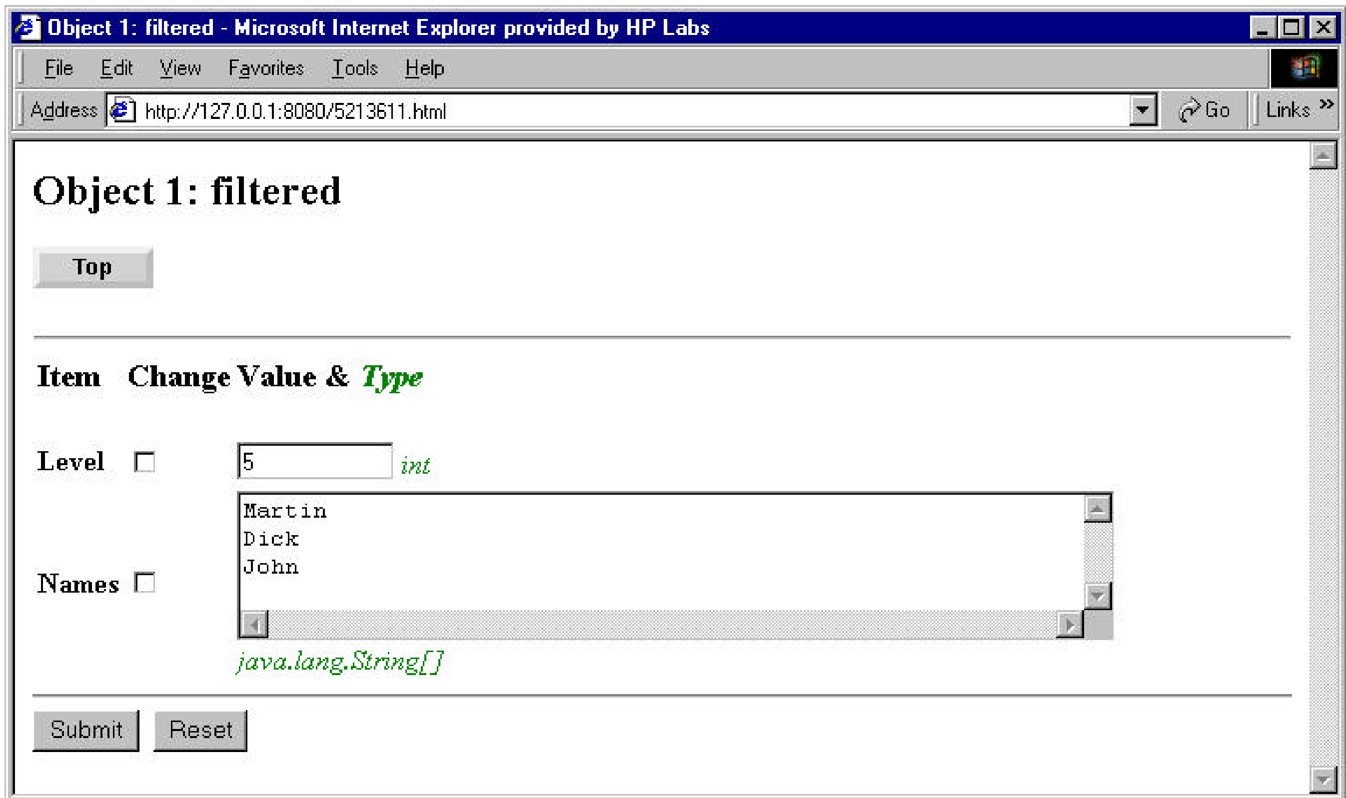

Figure 5 – Filter out selected methods in TestObject1

3.3.3 *Suppress editing on selected methods*

```
// Example 6. (See Figure 6)

// Suppress setters for Activated and Message to pass
 public boolean hasSetter(String metodName) {
        if (methodName.equals("Message"))    return false;
        if (methodName.equals("Activated"))    return false;
        return true;            // allow all others
    }
```

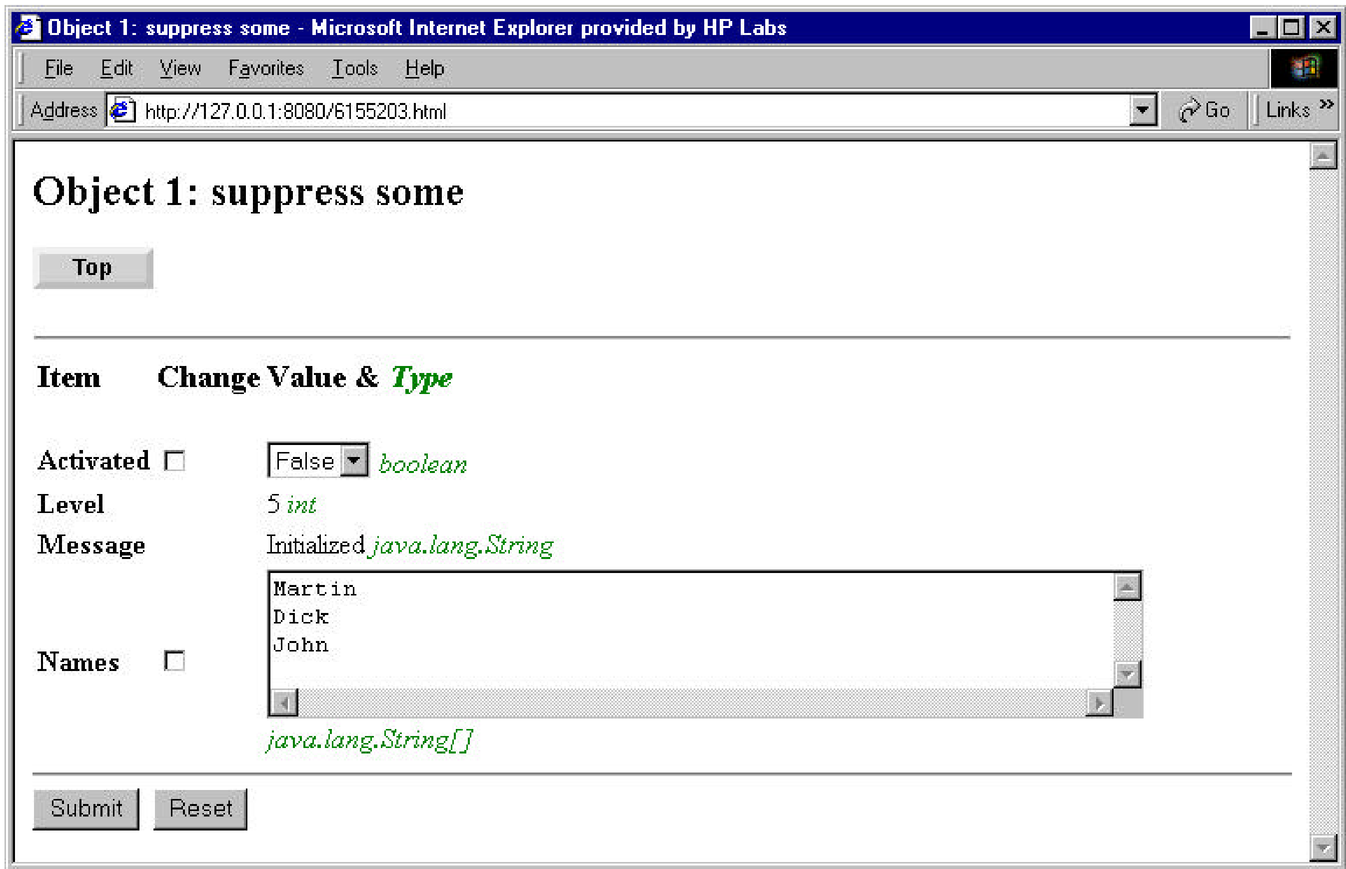

Figure 6 – Suppress editing on selected methods in TestObject1

3.3.4 Manually arranging the order of the presented items, suppressing some.

```
// Example 7 (See Figure 7).
//
public Vector reorderFilterMethods(Vector raw) {
    Vector sorted=new Vector();

     moveInspectionString(sorted,raw,"Message");
     moveInspectionString(sorted,raw,"Level");
     moveInspectionString(sorted,raw,"Names");

     //moveRemainingInspectionStrings(sorted,raw);    // if not all ordered manually
    return sorted;
  }
```

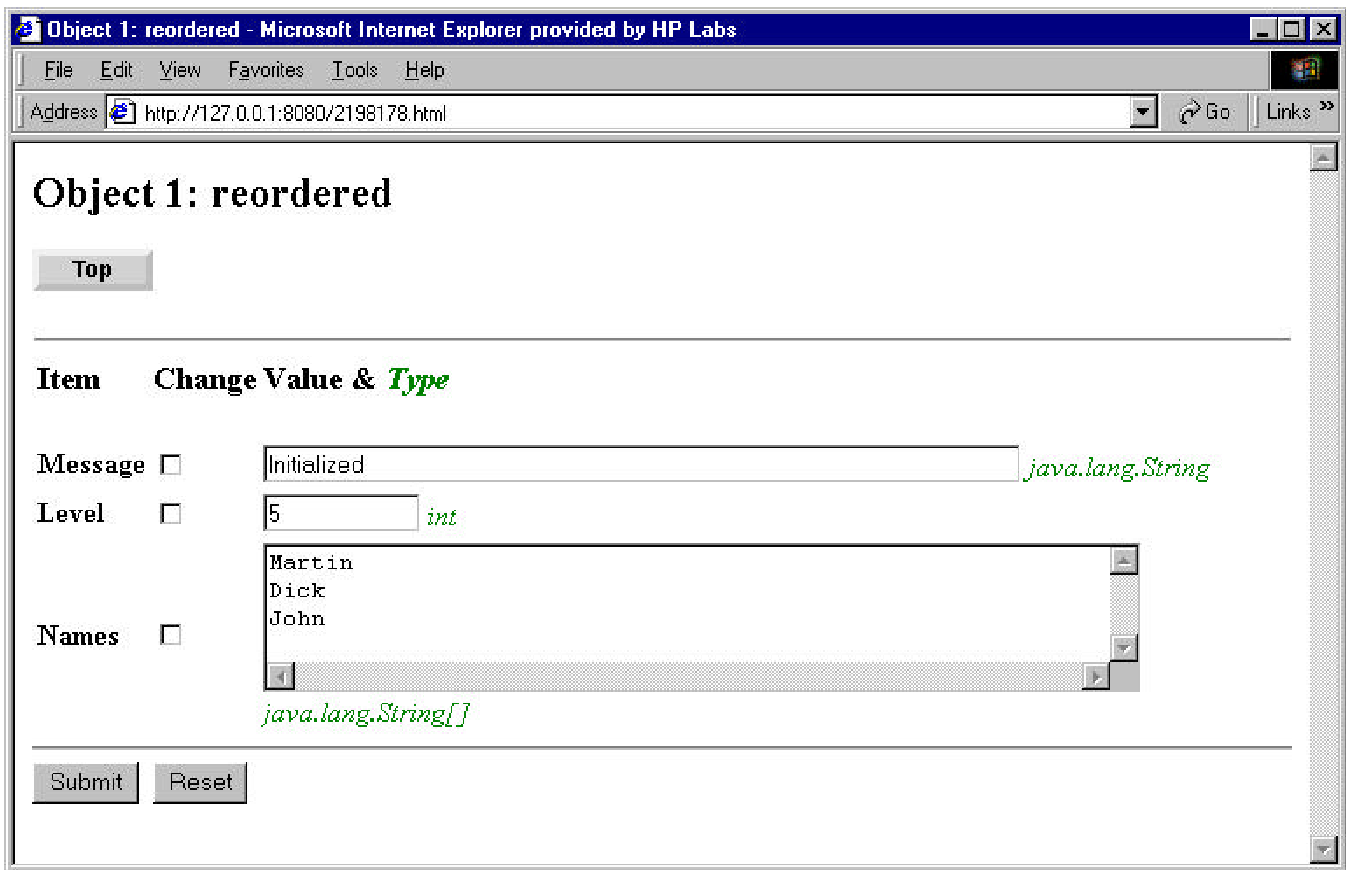

Figure 7 – Manually reordering and filtering methods in TestObject1

3.3.5 Other possibilities we have considered, but not yet fully explored, include

- Provide a natural presentation and type decorators for properties, HTML and XML data.
- Investigate the usage of frames to enhance presentation format.
- Make the user interface more dynamic through the use of JavaScript.
- Format the presentation for mobile appliances through the use of WML (Wireless Markup Language) or HDML.

3.4 Index Pages

Figure 1 showed a top-level index page containing links to 6 inspectable objects and 2 sub index pages. Given the ease of inspecting objects one may quickly find that the size of the inspectable collection becomes large and a single index page unwieldy. To alleviate this one can create and interlink multiple index pages. Figure 8 shows the subsequent index page if one had clicked on item 7 shown in Figure 1.

Pages manage views to either target objects or other pages. Here is the code that produced the output shown in Figure 8.

```
// Example 8. (See Figure 8)
// Create a page of views
TestObject1 test2 = new TestObject1();
Page p1 = inspector.getInspectionPage();  // Request a new page
p1.add(test2, new ViewOnlyPresentation(), "Object 2: view only");
p1.add(test2, new FilteredPresentation(), "Object 2: filtered");
p1.add(test2, new ViewSomePresentation(), "Object 2: suppress some");
p1.add(test2, new ReorderedPresentation(), "Object 2: reordered");
```

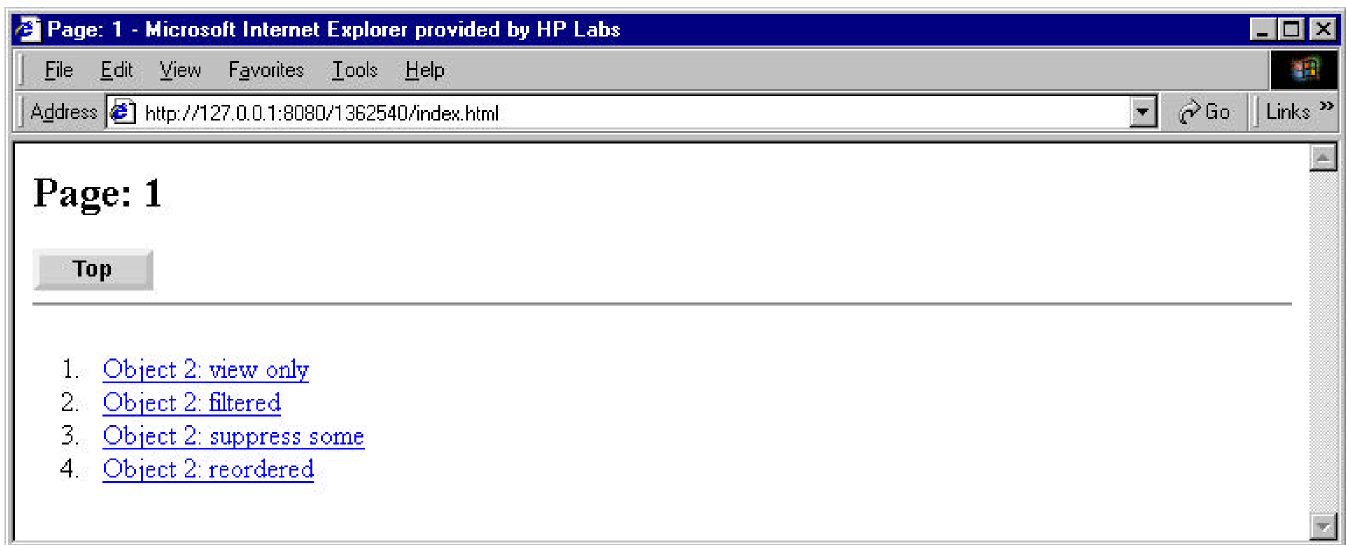

Figure 8 – Nested index page, referenced from top index page

To add this page to the inspector's page we would do the following:

inspector.add(p1, null, "Page: 1");

Now, along with the 6 objects previously presented on the top index page, there will be a link to the newly created sub page, entitled "Page: 1". This new sub page shows 4 different presentations of the same target object.

A link to the same page can be added to another page, p2, using:

p2.add(p1, null, "Page: 1, again");

3.5 Dynamic addition and removal of views, pages and execution of methods

It is very easy in any particular object to access (via getters) or change (via setters) any attribute; to dynamically add or remove inspection targets; change presentation objects; add or remove pages; or run arbitrary methods. For example, we have used this in our agent work to send ACL (Agent Communication Language) [ACL] messages, to unlock protected operations, enable or install new visible operations, etc.

4 Detailed Architecture and Design

The overall architecture of this system is shown below in Figure 9. Shown in the upper right corner is an existing application that is modified slightly to add an instance of the Inspection Server. That application will control the port assignment on which the inspection server will listen as well as controlling what objects may be inspected. Objects are added to the inspection collection using Inspection Server's add method and removed using its remove method. The add method also optionally accepts a presentation object and label which will be used to generate the actual HTTP response. These associations are combined to form a view and added to the inspector's page. This permits the same target to have differing presentations, as was shown in Example 8.

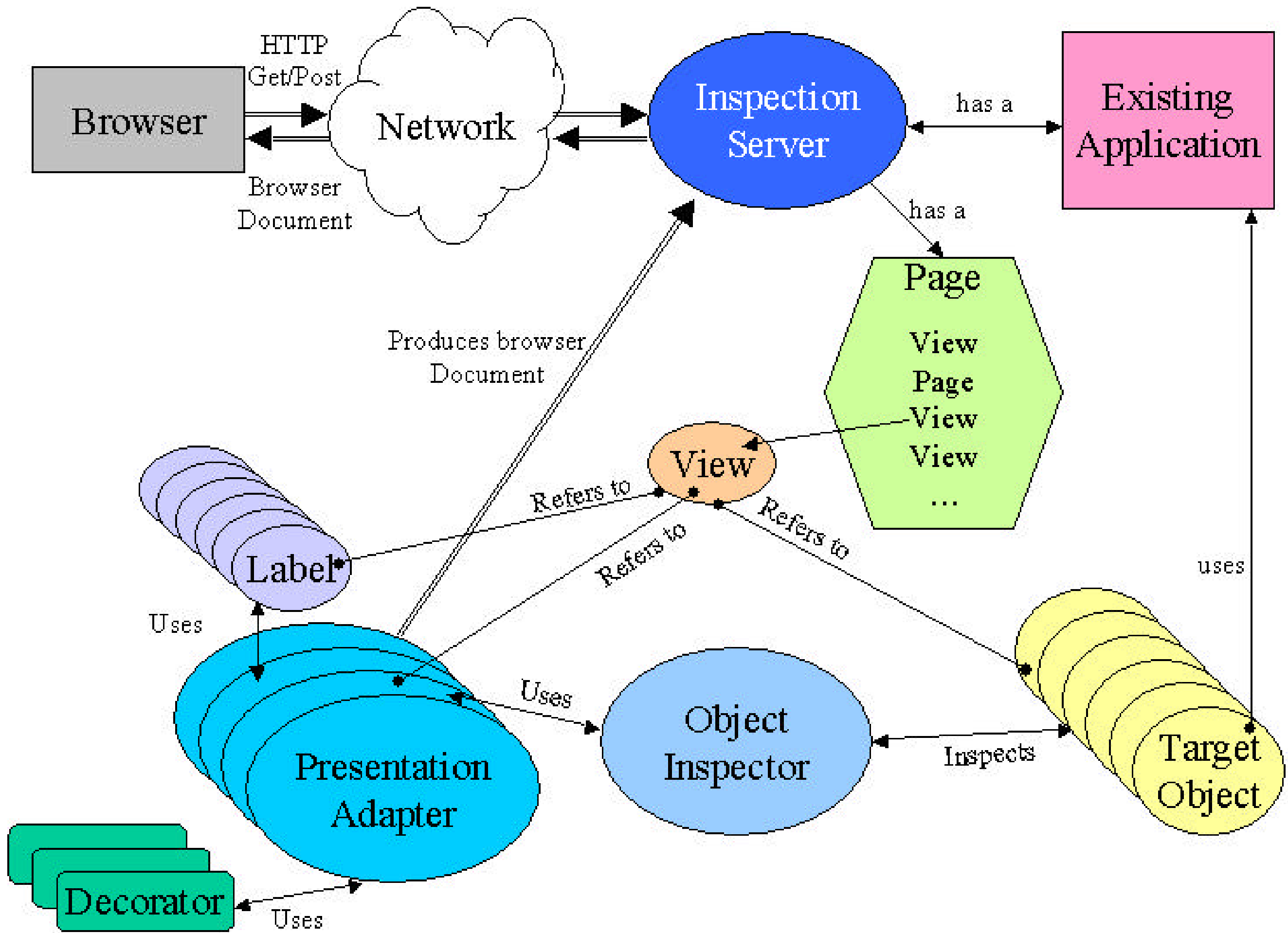

Figure 9 – Architecture of Inspection

4.1 Inspection Server

Listens for HTTP connections on a port number given to its constructor. It supports a limited GET functionality for accessing pages, target object references, or images from the same jar file as that containing the inspection classes. It handles POST functionality for updating target object.

4.2 Object Inspector

This section gives an overview of the Object Inspector that uses Java's reflection technology to examine and update target objects.

4.2.1 Acceptable Methods

Only public methods are acceptable.

4.2.2 Acceptable getter

For each acceptable method, for it to be considered an acceptable getter, it must meet the following conditions:

1. Method takes no parameters.
2. Return type is not void.
3. One of the following apply:
   - Method name begins with "get" and next letter is upper case.
   - Return type is boolean or Boolean and method name begins with one of the prefixes contained in a configurable collection (typically {is, has, can, allow}).

Although other methods may return a viewable result, calling other than a clear getter is dangerous due to actions they may invoke (i.e. side effects).

4.2.3 Viewable results

To be accepted, we must be able to view the result of the getter. That result could be a primitive (e.g. int, boolean, etc.), a standard Java object (e.g. java.util.Properties), or a custom object (e.g. com.hp.agent.coolagent.CoolAgent).

If the class is primitive, the test will be based on the primitive's true object counterpart since the invoked method always returns a true object. Since Java's Object class implements a toString() method this should return true almost always. The actual test is to see if the class implements a toString() method which takes no parameters and returns a String object.

4.2.4 Acceptable setter

To be able to update a data item, the target class must implement a setter method that meets the following requirements:

1. There exists a public void method whose name begins with "set" and the remainder of the name matches the acceptable getter minus its prefix.
2. That setter method takes a single parameter whose type matches that returned from the acceptable getter.
3. That argument type is either a String[] or an object which has a constructor which takes a String or String[].

4.3 Presentation

This has been discussed extensively in sections 2 and 3. The most interesting aspect is the layered structure of overridable methods, conforming to the prototypical structure and substructure of a display page, as illustrated in Figure 10. Combining string and HTML fragments into larger pieces, which are then assembled in rows in a table and placed within a header and footer, creates the resultant HTML document.

5 Related Implementations

The two most significant related implementations are JavaBeans[JavaBeans; Watson, 1997] and JavaServer Pages[JSP; Bergsten, 2000]. Also related is HP's ChaiServer, a relatively small embeddable HTTP-based object invocation model, similar in spirit to SOAP[SOAP], but also offering default web page presentations[CHAI 2001].

5.1 JavaBeans

1. Like our work, JavaBeans use Java's inspection technology to examine targets. However, JavaBeans focuses on inspecting a Java class while this technology focuses on inspecting a runtime object (instance of a class).
2. JavaBeans assumes that inspection is being done as part of application construction while this technology is focused on run time object inspection.
3. JavaBeans is highly biased towards being used from within a development IDE (Interactive Development Environment) while this technology is used at run time and uses a lightweight HTTP server and a standard web browser for the user interface.
4. JavaBeans can use introspection and standard design patterns to find the getters and setters, as we do; they also use property sheets, customizers and custom property editors to provide non-generic interfaces to JavaBean attributes at design time. In particular, a developer can add a BeanInfo class to provide explicit information about the methods, properties, events, etc, of their bean.
5. Finally, JavaBeans Activation Framework (JAF) [JAF], provides some capability at run time to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and to instantiate the appropriate bean, including perhaps a viewing operation.

5.2 JavaServer Pages (JSP)

1. Requires the use of a JSP [JSP] application server, such as TomCat or Apache extension, which is far too heavyweight to consider embedding in applications or appliances to provide object inspection over the web.

2. JSP is an addition to a web server while this technology provides a complete lightweight web server sufficient to meet its needs.

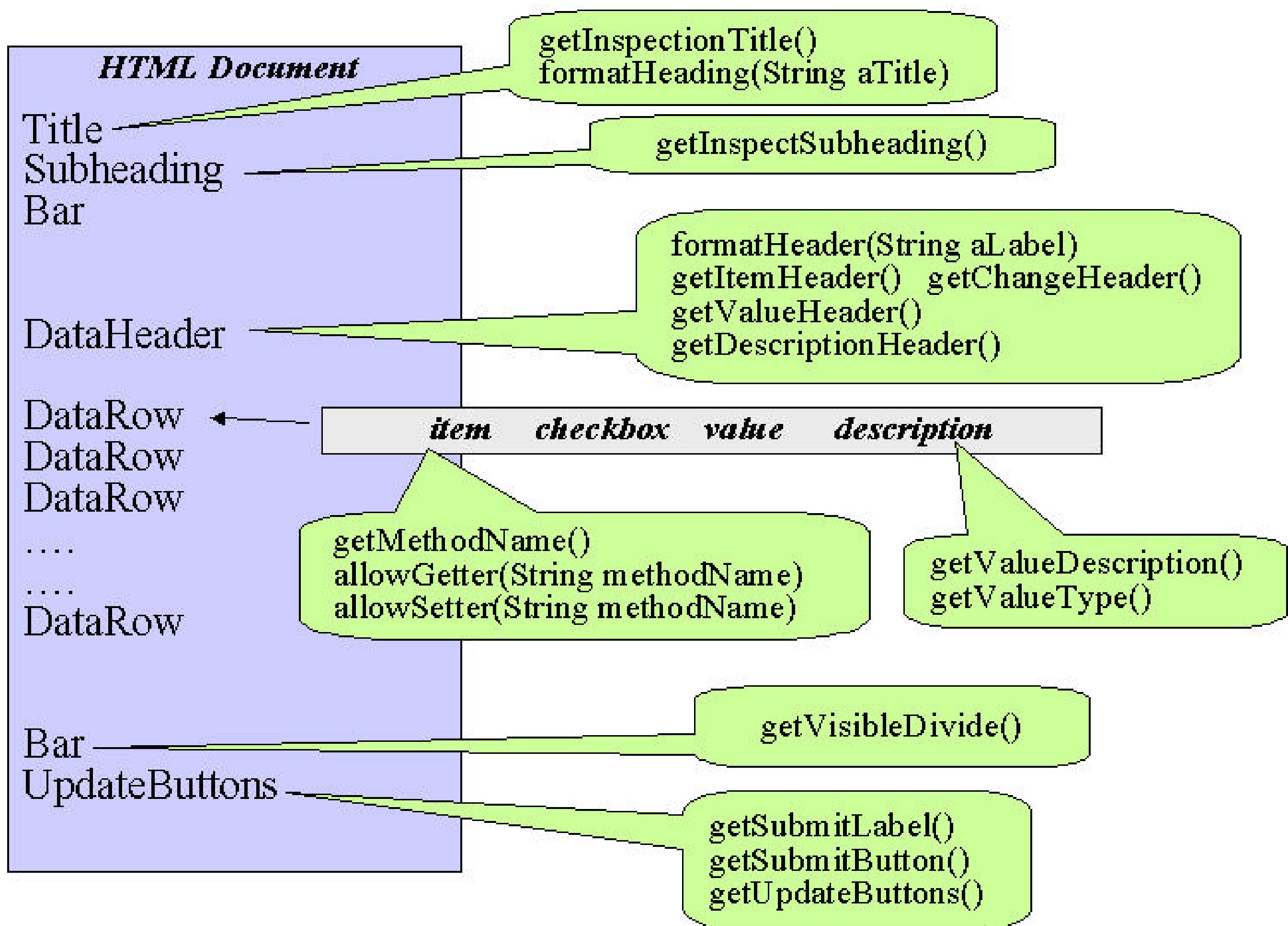

Figure 10 – Structure of a HTML document and related presentation methods

5.3 HP's ChaiServer

ChaiServer [CHAI 2001] is designed to enable embedded devices to be managed through the World Wide Web using standard browser technology. While ChaiServer is also based on a relatively small footprint web server, the significant differences between ChaiServer and this work are as follows:

1. The managed entity is typically an embedded appliance and hence a primary focus is on customization using downloadable modules known as "Chailets". Chailets interact with arbitrary hardware using Java Native Interfaces (JNI).
2. Object inspection using Java's reflection would not be important as typically, specific Chailets are developed completely with presentations for particular needs, and rather than accessing attributes of arbitrary objects, ChaiServer focuses on providing remote object invocation, using the typical http://server/chailet.method?parameter1=x¶meter2=y syntax, including invoking Chailets to modify the behavior of the ChaiServer itself. We would instead expose an appropriate public setter method on the InspectionServer object itself to achieve a similar effect.
3. Since the managed device is assumed to be autonomous and self-running, notifications of changes using SMTP or HTTP for event distribution is an important. Object inspection does not utilize a notification mechanism.

6 Possible Extensions

The following items have been discussed and may be experimented with in subsequent versions:

1. One can imagine adopting a convention that any public method whose name begins with "invoke" and which takes no parameters may be called to invoke some process. Ideally, these conventions should be done in a manner so as to preserve the neutrality of this technology. We could even include parameters with an appropriate set of data types, by having a special InvocationPresention. See also [SOAP]
2. Inspect the object returned by a getter. For any getter that returns an arbitrary publicly accessible object, it should be possible, using your browser, to add this new object to the inspection collection. For example a new check button could appear next to entries that meet this condition.
3. Investigate the usage of frames in a presentation class to enhance the displayed data.
4. Investigate substituting an alternative for "browser based" management using some form of instant messaging such as Jabber [Jabber].
5. Provide HTML macros and XML transformation presentations to enrich the range of presentations without undue additional mechanism.

7 Conclusions

Using this technology, we now have the ability to enrich both the usefulness of the presented data as well as to facilitate greater control over target objects. We have used this technology in our agent development work to greatly simplify the viewing and management of most objects, typically using small amounts of custom presentation code. Inspection has become one of three "core services" which most objects inherit. The other two core services are configuration using properties and journaling to console, file or Jabber client. This effort has resulted in a flexible, lightweight, standards based, process facilitating the access and control of objects at run time using browser technology.

REFERENCES

[Adams, 2001] D.J. Adams, "Programming Jabber," O'Reilly, 2001.

[ACL] Agent Communication Language. See http://www.fipa.org/specs/fipa00061/

[Bergsten, 2000] Hans Bergsten, "JavaServer Pages," O'Reilly, 2000.

[CHAI 2001] HP ChaiServer is part of the HP Chai Appliance Platform suite. HP offers a free download of ChaiServer, and other components. See http://www.hp.com/emso/products/chaisvr/index.html

[Gamma *et al.,* 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design patterns," Addison-Wesley, 1995. (See the discussion on page 4 about the SmallTalk implementation of MVC).

[Jabber] Jabber OpenSource Instant Messaging system. See Documentation and software available from: http://www.jabber.org/

[JAF] JavaBeans Activation Framework (http://java.sun.com/products/javabeans/glasgow/jaf.html)

[JavaBeans]. See documents reference from http://java.sun.com/products/javabeans/

[JSP] JavaServer Pages. See http://java.sun.com/products/jsp/

[Krasner & Pope, 1988] Glenn E. Krasner & Stephen T. Pope, "A Cookbook for using the model-view controller user interface paradigm in Smalltalk-80." JOOP, 1(3):26-49, Aug/Sep 1988.

[Watson, 1997] Mark Watson, "Creating Java Beans : Components for Distributed Applications," Morgan-Kaufman 1997.

[REFLECT] Java Core Reflection. See http://java.sun.com/products/jdk/1.1/docs/guide/reflection/

[SOAP] Simple Object Access Protocol (in XML). http://www.oasis-open.org/cover/soap.html

JavaÔ is a trademark of Sun. ChaiServerÔ is a trademark of Hewlett-Packard