Building Configurable Applications in Java

Mark C. Little and Stuart M. Wheater
Department of Computing Science,
Newcastle University,
Newcastle upon Tyne,
England, NE1 7RU

Appeared in the Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems, May 1998.

Abstract

There are many reasons why applications may require configuration, however the one which dominates Java applications is that of security restrictions. Because an application may be provided different capabilities by different users, it becomes difficult to write "build-once, run-anywhere" applications. Insisting that all security sensitive applications execute within controlled or restricted environments may limit the types of application which can be built. Therefore, in this paper we shall describe how we have constructed a configuration infrastructure in Java which allows applications to dynamically adapt themselves to the types of security restrictions that exist when they are executed. Because the system does not change the language it is portable across Java implementations. We shall also describe how we have used this system to build a toolkit for the construction of electronic commerce applications, which allow atomic transactions to span Web browsers and servers.

1. Introduction

Java has rapidly become one of the standard programming languages for the Web. The benefits of the language have been well documented [1][2], and include the ability to dynamically load code across the network, and to run applications on virtually any platform. Before Java the Web was a relatively static environment. However, programmers are now able to use Java to turn it into a general purpose distributed system [3].

There are many reasons why applications may require configuration, e.g., to incorporate bug fixes and new implementations [4][5][6]. However, the one which dominates Web applications is that of *security restrictions*. There are obvious security implications whenever a user downloads code from the network. Java security is imposed by a SecurityManager object, which defines what a program can, and cannot do [1][7]. Generally a Java program cannot remotely communicate with a node other than the one from which it was loaded, neither can it write to the disk of the machine on which it is being run. If the program is loaded directly from the local disk then these restrictions are relaxed. However, each implementer of a Java run-time can provide a different SecurityManager implementation, which may impose different constraints. Therefore, a program written for one Java implementation may not be able to execute on another.

There are two obvious solutions to this problem: (i) all objects must reside within domains which have well-behaved security constraints (typically Web servers), or (ii) modify the Java language and the run-time and provide an implementation of the SecurityManager which relaxes security restrictions [8][9]. Unfortunately, neither of these solutions is general enough. The first solution is unnecessarily restrictive in environments where SecurityManagers do allow programs increased flexibility. The second solution lacks portability, the very reason for using Java, as it requires users to have access to specialised implementations.

The approach to be described in this paper does not rely on modifying the language or the interpreter, yet is flexible enough to enable an application to configure itself to make use of the resources a given SecurityManager permits. We use the Gandiva model [4] to allow multiple implementations of components suited to different security restrictions to be provided by programmers and selected by an application at run-time, based upon the limitations in place when the application executes. We shall describe how we have developed a Java implementation of the Gandiva model, illustrating the advantages for configurability provided by Java. We shall also describe how we have used this system to build a toolkit for the construction of electronic commerce applications which use atomic transactions to control operations on persistent objects within the Web [10][11].

1.1 Digital signatures

To provide improved flexibility in security management policies, digital signatures were recently introduced into Java. Prior to being downloaded, programs can be signed with a unique signature for each provider. Users can associate a digital signature with a set of capabilities, e.g., being able to read from the user's home directory. Whenever a signed program attempts to perform an operation which would normally result in a security

violation, the SecurityManager inspects any capabilities assigned to it. If the capability exists which allows the program to perform the operation then it is carried out, otherwise a security violation exception is raised.

Being able to specify capabilities on a per user/domain basis allows more complex Java applications to be built. However, it also leads to further problems in being able to build truly portable Web applications: the capabilities assigned to a program by a user may not be known until it has been downloaded.

2. Software design model

We believe that techniques applicable to configurable distributed systems can be used to address the problems previously described: an application can be constructed once and be (dynamically) configured to suit the security environment in which it executes. Moreover we would like to allow programmers to be isolated from these reconfigurations in order that they can concentrate on building the applications. There are a number of software models for constructing configurable systems which meet our requirements, e.g., [5][6]. However, we have used the Gandiva model presented in [4].

2.1 The Gandiva model for configurable software

In the Gandiva model, applications are considered to be constructed from software components which are split into two separate entities: the *interface component* and the *implementation component.* Interactions between implementation components can only occur through these interface components. A core part of this model is that the binding between interface and implementation is configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation to be bound to an interface, it occurs in a way that allows this request to be changed without modifying the application.

2.2 Object-oriented implementation

In an object-oriented language like Java, it is possible to map interface components and implementation components onto *interface and implementation classes* respectively. However, although Java has its own interface type it is used for conformance purposes (similar to pure virtual base classes in C++). We require the binding between interface class and implementation class to be evaluated when the interface class is instantiated. Therefore, we use delegation to accomplish this, i.e., the interface class explicitly delegates all work to an implementation class [4].

In order to leave this binding until run-time we must specify it as data and not within the code of the interface class. The instance of the interface class (*interface object*) uses this data to create and bind to the correct instance of the implementation class (*implementation object*). To provide this separation of interface component and implementation component requires the following classes:

(i) *the interface class*: users interact with instances of this class, which defines the public operations that can be invoked on the implementation. The only implementation specific information present in the class definition is a reference to an instance of an *implementation interface*, to which the interface delegates all operations.

(ii) *the implementation interface*: this is a Java `interface` and all implementations accessible to an interface class implement it.

(iii) *the implementation class*: instances of this class represent the implementation of an object. Implementation classes can be derived from multiple implementation interfaces.

Figure 1 shows a UML object structure formed by the above classes. (An optional Control class can also be provided by the implementation to access its non-functional characteristics, e.g., setting timeout and retry values for an RPC mechanism [4].)

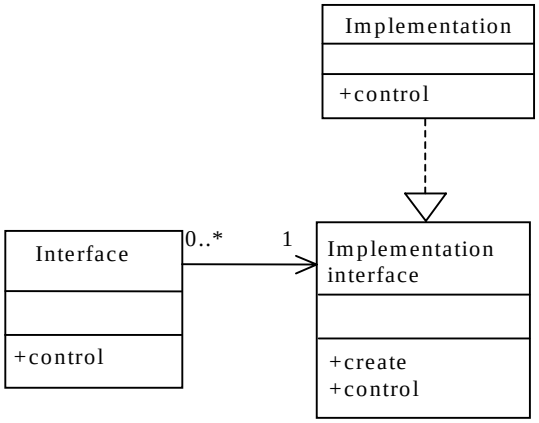

Figure 1: Interface and Implementation relationship.

3. Java implementation

Gandiva provides a set of classes to support the construction and use of interface and implementation classes [4]. The two most important classes for our discussion are ObjectName, which is responsible for storing and retrieving the configuration information required by an application, and the Inventory, which is responsible for managing repositories of implementation classes and returning new instances to the application.

3.1 ObjectName

The ObjectName class is used to control the configuration of an application, e.g., the mapping of interface classes to implementation classes, and any

initialisation data required by implementations. This configuration information is maintained as a set of name (string), value pair *attributes*. Instances of ObjectName are used to specify and store these attributes between successive instantiations of interfaces.

An interface object uses the attributes of ObjectName to determine the type of its implementation; this implementation can also use the ObjectName to further configure itself. If multiple bindings are allowed for the interface, e.g., because of possible security restrictions, the ObjectName can specify alternate implementations. These alternates may be prioritised within the ObjectName.

The signature of the Java implementation of ObjectName is shown below:

```
public class ObjectName implements Serializable
{
// the supported attribute types
public static final int SIGNED_NUMBER = 0;
// for C++ compatibility
public static final int UNSIGNED_NUMBER = 1;
public static final int STRING = 2;
public static final int OBJECTNAME = 3;
public static final int CLASSNAME = 4;
public static final int UID = 5;

public int attributeType (String attrName)
                              throws IOException;
public String firstAttributeName ()
                              throws IOException;
public String nextAttributeName (String attr)
                              throws IOException;

// Now a series of set/get methods for each type
// of attribute. We show only one for simplicity.
public long getLongAttribute (String attrName)
                              throws IOException;
public void setLongAttribute (String attr,
                              long value)
                              throws IOException;


public boolean removeAttribute (String attrName)
                              throws IOException;

private NameService _nameService;
}
```

There are methods for creating new attribute name, value pairs, and for retrieving an attribute given its name. Additionally, it is possible to query the type of an attribute using `attributeType`, and to iterate through all of the attributes using `firstAttributeName` and `nextAttributeName`.

To enable the configuration information to be stored in a flexible manner, `ObjectName` stores and retrieves the information using a separate NameService interface and implementation [4]. Therefore, the means of storing this configuration data can be changed simply be changing the NameService implementation. For example, the JDBC (Java Database Connectivity) API is a standard SQL database access interface, providing uniform access to a wide range of relational databases. By providing a suitable NameService implementation, the `ObjectName` data could be maintained within such a database. However, to minimise external dependencies, our current implementation for Web applications embeds the `ObjectName` data within the HTML document which is downloaded with the Java application. The HTML document is created automatically from a separate description language.

3.2 Inventory

This is an interface class and a set of implementation classes, and is at the core of the system; it is a repository for implementation classes and provides a means for the dynamic creation of objects based upon their class names. Several implementations of the inventory exist, each tailored for specific applications, e.g., one which checks versions of implementation classes registered with it and returns the latest version. Populating the inventory with implementation classes can occur:

1) *statically at build time*: specific implementations are "hard-wired" into the inventory when the application is constructed.
2) *dynamically at run time*: implementation classes may be dynamically loaded across the network or from the local disk. This has the advantage of flexibility, but requires the sources of these implementations (e.g., Web servers) to remain available while the application is being configured.

Because the inventory is accessed through a well-defined interface, changing its implementation does not require any changes in an application. For simplicity we show only a representative set of the Inventory interface methods, without the exceptions they throw:

```
public class Inventory
{
public Object createVoid (String typeName);
public Object createObjectName String typeName,
                    ObjectName paramObjectName);

// A handle on the application's inventory for
// bootstrapping purposes.
public static Inventory inventory ();
}
```

The create methods take the name of the implementation class (a string) to instantiate and, depending on the method, pass additional parameter(s) to the created implementation. For example, `createObjectName` will pass the ObjectName parameter to the implementation when it is created. In order that the inventory can deal with any Java implementation class, it returns all created objects to the caller (the interface) as instances of the Java `Object` class, which is the base class from which all Java classes are derived. The interface can then safely convert this back to the required interface implementation class.

3.2.1 ClassLoaders and dynamic inventories

Java has been designed to have a minimal run-time footprint: code needed by an application can be dynamically loaded across the network or from disk (assuming the SecurityManager allows) by a ClassLoader object. It is the responsibility of the ClassLoader to load the byte streams representing new classes, and that loading them does not violate the security policy. Typically class representations must reside within a specific search path for the ClassLoader to find and use them. However, it is possible for users to override the default ClassLoader to provide implementations more suited to their requirements [2].

The basic definition of the Java ClassLoader is shown below (all Java types are represented by an instance of `Class` which can be used to create instances of that type):

```
public abstract class ClassLoader extends Object
{
public Class loadClass (String name)
                  throws ClassNotFoundException;

protected abstract Class loadClass (String name,
                                    boolean resolve)
                  throws ClassNotFoundException;
};
```

Given the string name of the class, the virtual machine calls `loadClass` which either finds and returns the `Class` representation or throws an exception. Therefore, the combination of the ClassLoader and Class are a natural way to implement a dynamic inventory in Java. In the following section we shall briefly describe one such implementation, the *network ClassLoader inventory*.

3.2.2 The network ClassLoader inventory

The network ClassLoader inventory obtains implementation code required by an application from a *network implementation repository*: machines on which implementation code resides co-operate to share their code base. Figure 2 illustrates how, by communicating with a single member of this repository, an application can gain access to implementations maintained by other repository members. The repositories are all written in Java, and each repository member can only load class implementations from a specific search path. Communication with and between repository members requires authentication and uses Java's Remote Method Invocation (RMI) mechanism.

If an application is configured to use the ClassLoader Inventory, programmers need only ensure that implementation code is available to a single repository site. When the inventory requires code for a new implementation it will contact one of the repository members, which (potentially) will search the entire repository for the necessary code. The repository to contact may be specified within the initial configuration information, or may be imposed by SecurityManager restrictions, e.g., for most Web browsers the repository member must reside on the machine from which the program was downloaded.

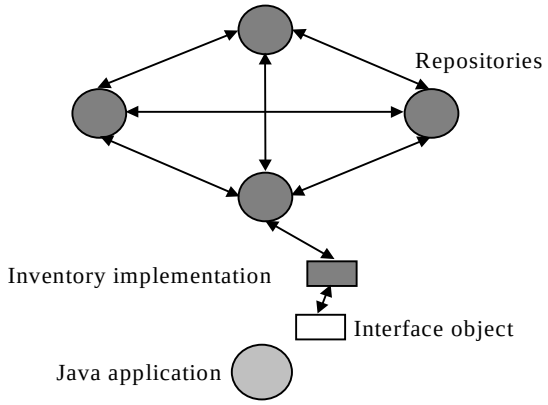

Figure 2: Network implementation repositories.

3.3 Interfaces

When an interface is instantiated it is responsible for binding to an appropriate implementation. Figure 3 illustrates how this occurs.

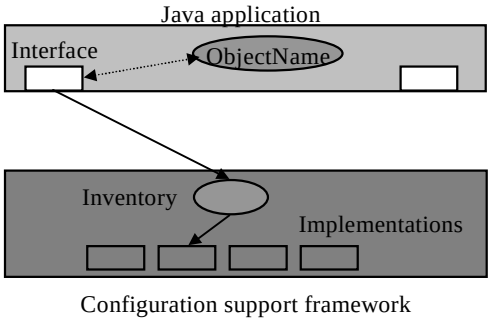

Figure 3: Application execution environment.

To determine which implementation it requires, the interface interrogates the ObjectName to obtain the name of an implementation and passes this name to the inventory. If the requested implementation has not been registered with the inventory, or cannot be used within the current environment, e.g., because of security restrictions, then the binding will fail. If alternative implementations are specified within the ObjectName (e.g., to tolerate different security restrictions) then the interface can try to use them. If there are no implementations which can function for the interface, all subsequent invocations on the interface will fail. Importantly, none of this is visible at the application level: the programmer simply creates and uses an interface object.

The Gandiva build-time support includes an interface/implementation code generation tool: given the definition of an implementation interface, this tool will automatically create an interface class with all of the necessary binding code, and (optionally) skeleton implementation classes which the programmer can populate with code. For example, consider an

implementation interface to a (simplified) persistent object store service, used for storing the state of an object between instantiations:

```
interface ObjectStoreImple
{
public ObjectState readState (ObjectID id)
                     throws ObjectStoreException;
public void writeState (ObjectState state,
                        ObjectID id)
                     throws ObjectStoreException;
};
```

Ignoring error checking, part of the interface class produced by the code generation tool is:

```
public class ObjectStore
{
public ObjectStore (ObjectName objName)
{
  Inventory inventory = Inventory.inventory();
  String type =
      objName.getClassNameAttribute("ClassName");
  ObjectName oName =
    objName.getObjectNameAttribute("ObjectName");
  Object ptr = inventory.createObjectName(type,
                                          oName);

  // if returned object is of right type
  // then cast to actual type.
  if (ptr instanceof ObjectStoreImple)
    imple = (ObjectStoreImple) ptr;
  else
    imple = null;
}

public ObjectState readState (ObjectID id)
                      throws ObjectStoreException
{
  if (_imple != null)
    return _imple->readState(id);
  else
    throw new ObjectStoreException("No imple");
}

// the implementation object
private ObjectStoreImple _imple;
};
```

3.4 Determining security restrictions

In order to configure itself to operate within a specific security environment, an application must be able to determine the restrictions imposed by that environment. At bind time an interface must be able to determine whether the implementation it receives from the inventory can work within the current security restrictions. Therefore, each implementation object must provide a `canExecute` method which returns either true if it can execute within the current environment, or false if it cannot. When the inventory returns an implementation object, the interface calls this method to determine whether the object can function. If it cannot, the interface can ask the `ObjectName` for the name of another implementation, and pass this to the inventory.

This technique of delegating to the implementation the requirement for determining whether or not it can execute within a given environment allows us to remove any knowledge of specific security restrictions from the Gandiva framework and interfaces. An interface can only tell whether an implementation can function by calling the `canExecute` method. New security restrictions can be accounted for by providing new implementations without requiring changes to either Gandiva or the interfaces which use the implementations.

To determine whether or not it can function within the security environment, the implementation object may extract information from the `ObjectName` it is given when it is created, e.g., the location of the object store database to use for persistence, or the name of the remote host to contact. It can then obtain a reference to the current SecurityManager and query its restrictions accordingly. Shown below is the `canExecute` method for a simple object store service which writes to the local file system. In the example below, the implementation has been written to cope with a fairly typical security policy, which restricts the file system access of Java programs to specific locations, e.g., a temporary file store. If the security restrictions prevent access to this location the implementation will not function and the interface must try an alternate implementation.

```
public SimpleObjectStore implements
                                ObjectStoreImple
{
public boolean canExecute ()
{
  // First get handle on current
  // SecurityManager.

   SecurityManager manager =
                  System.getSecurityManager();

   if (manager == null)
     return true; // no restrictions!
   else
   {
     // There is a SecurityManager, so
     // interrogate it to find restrictions.
     try
     {
        // Assume these file names were read
        // from the ObjectName when this
        // implementation was created.
        manager.checkRead("/ObjStore/data");
        manager.checkWrite("/ObjStore/data);
        manager.checkDelete("/ObjStore/data");

        return true;
     }
     catch (Exception e)
     {
        // SecurityManager raised an
        // exception, we could try alternate
        // location.
       return false;
     }
   }
}
}
```

At present application programmers must implement the `canExecute` methods. However, we are examining the

possibility of automatically generating this code from a high-level specification language.

4. Configurable Web commerce applications

One of the reasons for building this configuration framework in Java was to support the construction of electronic commerce applications, and in particular those which require atomic transactions in a Web environment [10][11]. The Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. For example, if a user purchases a cookie (a token) granting access to a newspaper site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an indeterminate state. For resources such as documents, failures may simply be annoying to users; for commercial services, they can result in loss of revenue and credibility.

Atomic actions are a well-known technique for guaranteeing application consistency in the presence of failures. Web applications already exist which offer transactional guarantees to users. However, these guarantees only extend to resources used at Web servers, or between servers; browsers are not included, despite being a significant source of unreliability. Providing end-to-end transactional integrity between the browser and the application is important: in the previous example, the cookie *must* be delivered once the user's account has been debited. Cgi-scripts cannot provide this level of transactional integrity since replies sent *after* the transactions have completed may be lost, and replies sent *during* the transaction may need to be revoked if the transaction cannot complete [12]. This is an inherent problem with the original "thin" client model of the Web, where browsers were functionally barren. With the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. However, the constraints imposed by Java SecurityManagers can directly affect transactional applications which may require, for example, to make state updates (e.g., cookies) persistent by accessing the local disk.

Using the Java implementation of Gandiva, we have designed and implemented a transaction toolkit for the Web, *JTSArjuna* [11]. The toolkit, which is based upon the Arjuna system [13], allows transactional applications to span Web browsers and servers, and benefits from the configurability described previously: an application can be made transactional without compromising the security policies operational at browsers and servers. Finally, the toolkit complies with the OMG Object Transaction Service (OTS) [14][15] and the Java Transaction Service (JTS) [16] standards. It is beyond the scope of this paper to describe how JTSArjuna provides end-to-end transactional integrity for Web applications; we shall concentrate only on its configurability. However, the interested reader is referred to [10][11] for a detailed description of the other aspects of the system.

4.1 Transaction standards for distributed objects

The Object Management Group (OMG) has specified a Common Object Request Broker Architecture (CORBA) that at the basic level consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other [14][15]. At the next level a number of system level services have been specified, which include persistence, concurrency control, and the Object Transaction Service (OTS). The OTS is a *protocol engine* intended to guarantee that transactional behaviour is obeyed, but it does not directly support all of the transaction properties. As such it depends on other system level services for the required functionality, e.g., persistence and concurrency control services. The Java Transaction Service (JTS) is a direct mapping of the OTS to the Java language using the standard Java IDL mapping [16].

4.2 JTSArjuna model

The interfaces defined by the OMG for building transactional applications are too low-level for most application programmers. For example, the programmer is required to manage persistence, concurrency control etc. on behalf of every transactional object, and registering it with each transaction it participates with. Therefore, we have provided a higher-level API which attempts to hide many of these details from programmers. This API has been based on the experiences gained from extensive use of the original Arjuna system [13][17].

The JTSArjuna model for building transactional applications exploits object-oriented techniques to present programmers with a toolkit of classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control [11][13]. Each class is concerned with a single functionality, and these classes form a hierarchy, part of which is shown in figure 4.

StateManager is the class responsible for naming, persistence and recovery control of persistent and recoverable objects. When not in use, persistent objects are stored in an object store, whereas recoverable objects are always memory resident. To satisfy the durability requirement, StateManager makes use of a persistence service when activating (loading) and deactivating (storing) persistent objects. The programmer must provide `save_state` and `restore_state` methods for each class which specify which parts of the object's state to store/restore.

LockManager is responsible for two-phase locking, and uses instances of the Lock class, in conjunction with a suitable concurrency control service, to accomplish this. The programmer uses the `setlock` method to acquire appropriate locks (read or write locks in the current implementation); the atomic action mechanism is responsible for releasing these locks. AtomicAction is the class which implements the OTS protocol engine, and this class uses StateManager in order to make any transactional decisions persistent.

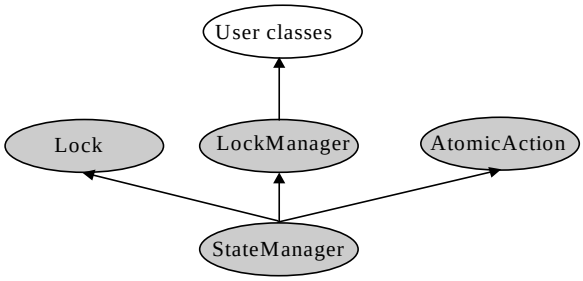

Figure 4: JTSArjuna class hierarchy

By inheriting from LockManager, user classes are automatically transactional, with LockManager and StateManager being responsible for guaranteeing the ACID properties. Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: JTSArjuna guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and that in the event of failures crash recovery mechanisms are invoked automatically.

4.2.1 Configurable implementations

Both the persistence and concurrency control services required by transactions can be directly affected by the security restrictions imposed by a Java SecurityManager. Therefore, these services are required to be configurable within the toolkit and have been implemented using interface and implementation separation. This allows the toolkit and applications written with it to execute within different environments.

Figure 5 shows a transactional user class inheriting from LockManager. Internally, LockManager uses the concurrency service (CC) through an interface; currently there are three different implementations which that interface can use:

1) a local disk implementation, where locks are made persistent by being written to the local file system; this allows locks, and therefore objects, to be shared between different Java virtual machines.
2) a purely local implementation, where locks are maintained within the memory of the virtual machine which created them; objects cannot be shared between virtual machines.
3) a remote implementation, where the implementation which LockManager uses is actually a client-side stub (proxy) to a remote service, which may reside within another browser or at a Web server.

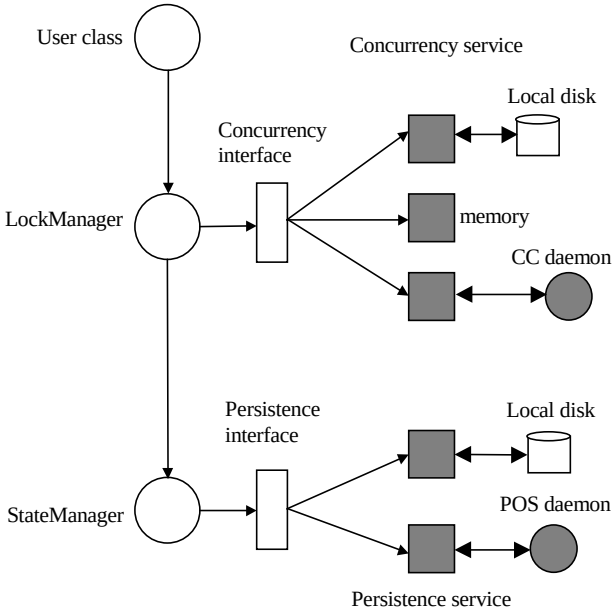

Figure 5: Configuration hierarchy

Likewise, the interface through which the StateManager class uses the Persistent Object Service (POS) can bind to either a local (file-system) implementation or a remote implementation. (In memory implementations are used only for recoverable objects.)

Shown below is an example of the configuration information necessary for the simple transactional object shown in figure 5. (The '~' and '!' characters preceding each attribute value are used for runtime type checking by `ObjectName`.) Importantly, there are no requirements from the application user: all implementations will be loaded across the network when required.

```
<HTML>
<HEAD><TITLE>Example Applet</TITLE></HEAD>
<BODY>
<APPLET CODE=TranApplet.class WIDTH=400
HEIGHT=200>
<PARAM NAME=OSClassName1
VALUE="~LocalObjectStoreImple">
<PARAM NAME=OSLocation1
VALUE="!/tmp/ObjectStore">
<PARAM NAME=OSClassName2
VALUE="~RemoteObjectStoreImple">
<PARAM NAME=OSLocation2
VALUE="!glororan.ncl.ac.uk">
<PARAM NAME=CCClassName1 VALUE="~LocalCCImple">
</APPLET>
</BODY>
</HTML>
```

The preferred type of the persistence service is LocalObjectStoreImple, with the attribute name OSClassName, and the location of the object store is the directory /tmp/ObjectStore. If this fails, the interface can

use the alternate implementation RemoteObjectStoreImple which is on the specified machine. The concurrency service is local. If the programmer wishes to change the configuration of the application, only modifications to the HTML document are required.

5. Conclusions and future work

We have shown that to build many types of Java applications which are truly "write-once-run-anywhere", configuration support is required. We have also shown how that support can be provided using the interface and implementation separation model of Gandiva, and how features of the Java language can be used to facilitate configuration. Finally, we have used this framework to provide support for the construction of configurable Web commerce applications using atomic actions and persistent objects. The framework presented should allow applications to be constructed which can cope with many different types of security restrictions. We intend to construct a range of applications with different requirements on security to confirm this.

New Java features offer further possibilities for configuration support. JDK 1.1, introduced classes to support reflection and introspection [1], making it possible to query the capabilities of a class at run-time and determine, for example, what methods it provides, what parameters they take, and what exceptions they raise. Using this reflection API an application can use code it had no prior knowledge of simply by invoking the class and its methods through their names. Therefore, implementations need not conform to an implementation interface for interfaces to be able to use them. In addition, JDK 1.1 introduced the concept of object serialisation: the complete state of an object, including any objects it refers to, can be written to an output stream, and this stream can be used to recreate that object at a later time. Therefore, once an application has been configured, the configuration could be "frozen" using this mechanism, and automatically recreated later, without a need for the original ObjectName.

Acknowledgements

The work reported here has been supported in part by a grant from UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708).

References

[1] D. Flanagan, "Java in a Nutshell 2nd Edition", O'Reilly, 1997.

[2] K. Arnold and J. Gosling, "The Java Programming Language", Addison Wesley, 1996.

[3] R. Orfali et al, "Client/Server Programming with Java and Corba", Wiley, 1997.

[4] S. M. Wheater and M. C. Little, "The Design and Implementation of a Framework for Configurable Software", Proceedings of the 3rd International Conference on Configurable Distributed Systems, May 1996, pp. 136-143.

[5] J. Magee et al, "A Constructive Development Environment for Parallel and Distributed Programs", Proceedings of the 2nd International Workshop on Configurable Distributed Systems, March 1994, pp. 4-14.

[6] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 425-436.

[7] J. S. Fritzinger and M. Mueller, "Java Security", Sun Microsystems, 1995.

[8] M. Atkinson et al, "Draft Pjava Design 1.2", Department of Computing Science, University of Glasgow, January 1996.

[9] A. Garthwaite and S. Nettles, "Transactions for Java", MS-CIS-96-17, University of Pennsylvania, 1996.

[10] M. C. Little et al, "Constructing Reliable Web Applications Using Atomic Actions", Proceedings of the 6th Web Conference, April 1997, pp 561-571.

[11] M. C. Little and S. K. Shrivastava, "Distributed Transactions in Java", Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155.

[12] T. Sanfilippo and D. Weisman, "Applications of the Secure Web Technology in Transaction Processing Systems", The Open Group Research Institute, November 1996.

[13] G. D. Parrington, S. K. Shrivastava, S. M. Wheater and M. C. Little, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8, No. 3, pp. 253-306, Summer 1995.

[14] "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.

[15] R. Orfali et al, "The Essential Distributed Object Survival Guide", Wiley, 1996.

[16] V. Matena and R. Cattell, "JTS: A Java Transaction Service API", Sun Microsystems, December 1996.

[17] S. K. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system," International Workshop on Distributed Computing Systems: Theory meets Practice, Dagsthul, September 1994, LNCS 938, Springer-Verlag, July 1995.