Controls Overview

Table of contents

1. Overview

1.1. The Problem with J2EE: Complexity

J2EE provides a rich set of component types, protocols, and system services that can be used to assemble an application or service. But as the scope of the J2EE architectural design space has grown, the complexity of assembling solutions has also grown. This has created a problem for developers who are not J2EE experts: J2EE solutions are often too complex for non-experts to effectively utilize them.

An objective of the Beehive community is to expand beyond the systems software developer who has traditionally built J2EE solutions to enfranchise those developers who may have less experience with object-oriented design, building distributed systems, and Java/J2EE.

The goal is to enable a new collaboration where the base J2EE distributed system architecture and back-end components can be designed and built by the J2EE system software developer, then assembled into exposed web user interfaces, web services, or applications by the application developer.

Without this new collaboration, the application developer might have to learn a variety of new technologies and APIs to work within the architecture.

Consider a simple example: A systems developer has built a distributed system where services are exposed as Enterprise JavaBeans and JMS queues. An application developer new to J2EE is tasked with building a web user interface that integrates with these services.

To accomplish this task, the application developer now has to learn how to:

- Create a JNDI context and lookup resources. If resources are app-scoped, then he must learn how to provide the appropriate deployment descriptor configuration.
- Use interfaces of exposed EJBs to access methods, including understanding differences in usage depending upon whether the exposed EJBs are Stateless Session Beans, Stateful Session Beans, or Entity Beans.
- Obtain JMS connections/sessions, and references to queues.
- Construct and enqueue a JMS message.
- Properly manage the resources associated with the above, such that vital system resources (such as connections) are used efficiently and correctly. The cost of a subtle mistake can be poor system performance or even system failure.

What initially appears to be a simple task in the abstract (call these EJBs or enqueue a message that looks like this) can devolve into hours or days of reading J2EE HowTo books and Javadoc API references, getting the right deployment descriptor values configured, and

calling all the right APIs, at all of the right times, in the right order. In the resulting application or service, often the directly application-related code (i.e. calling the method or building message contents) is a small fraction of the total code required to accomplish the task.

Here is an example of the code required to invoke a single method (`buy()`) on an exposed EJB using standard J2EE APIs:

```
Trader trader = null;

try {
    InitialContext ic = new InitialContext();
    TraderHome home = (TraderHome)ic.lookup("MyTraderBean");
    Trader trader = home.create();
    TradeResult tradeResult = trader.buy(stock, shares);
    return tradeResult;
}
catch (NamingException e) {
    ...
}
finally {
    if (trader != null)
        trader.remove();
}
```

A common solution to this problem is often to task the J2EE professional developer with constructing facades or custom frameworks that hide some of the underlying complexity of the resource access mechanisms and provides appropriate guarantees that system resources (connections, sessions, handles, etc) are utilized properly. But constructing these intermediate abstractions is an inefficient use of (often scarce and expensive) systems development resources. Depending upon the "thickness" of the intermediate abstractions, this approach can also have performance or application deployment footprint implications.

1.2. Solution: Controls: A Unified Client Programming Model

Controls reduce the complexity and learning curve associated with acting as a client of J2EE resources by providing a unified client model that can provide access to diverse types of resources. Controls provide the following features for reducing the learning curve for non-expert developers:

1. To a Control client, Controls appear as JavaBeans that can be instantiated and used for resource access. Controls present operations on the J2EE resource as methods on a JavaBean interface.
2. Properties that parameterize resource access can be set using Java 5 metadata. This

configuration mechanism is consistent across all J2EE resource types.
3. Controls provide a consistent model for discovering the resource's configuration options and operations.
4. Controls can also provide transparent (to the client) resource management of connections, sessions, or other resources to be obtained on behalf of the client, held for an appropriate resource scope to achieve best performance, and then released. This resource management mechanism frees the client from having to learn or understand the acquisition mechanisms, and from having to directly participate in guaranteeing their release. The Controls architecture provides this functionality by defining a simple resource management contract that can cooperate with an outer container to manage resources at the appropriate scope (for example, bounded to a transaction context or outer container operation or request scope).

Using a Control to expose the Trader EJB in the earlier example, the code to invoke the `buy()` method can become:

```
traderControl.buy();
```

The Trader Control fully encapsulates the JNDI lookup as well as the home/bean interface operations needed to get an instance of the Trader EJB and invoke the buy() method on it, and exposes the JNDI name of the EJB as a property that can be set via a metadata annotation.

Controls also provide an extensibility model that allows customized views of a resource to be constructed, with discrete operations defined as methods on the control. For example, it is possible to define a custom operation on a Control type representing a JMS queue resource that uses metadata attributes to define the format of the message with message contents set from message parameters. This enables an application developer to construct new customized facades for resource access less effort.

The goal of the Controls architecture is not to define the standards for how specific resource types will be exposed; rather, it is to guarantee that when exposed they will have a commonality in mechanism that makes them easier to understand and use by developers.

1.3. Another Problem: Tooling Challenges

Beyond adding to overall complexity, the diversity of J2EE resource types and access mechanisms also makes it difficult for tools to offer assistance to developers who need to use them.

For existing client models, the configuration of resource access is often some combination of resource-specific API usage and deployment descriptor entries. This

generally requires custom IDE code that knows how to generate the right (resource-specific) code or configuration entries.

Specific resource types often need custom code in order to define wizards or property-driven user interface that aids in the process of defining a client of the resource. There is no common mechanism for discovering the potential set of configurable attributes for a resource type. This means that any graphical presentation of client attributes or wizards must be custom-authored based upon resource type.

Once configured, there is the secondary problem of how the IDE represents a configured client resource in source form. There are at least two potential options: save the attributes as generated source code and/or deployment descriptor entries that are resource-specific or define a canonical representation that is native to the IDE. Both are problematic. Two-way editing can be difficult, if the canonical format is generated source code or descriptors are visible to the end user and directly editable. Using some IDE-specific canonical representation (either based upon a closed framework or configuration data) means the configured client abstraction isn't portable to other development environments or editable outside of the IDE.

Using the IDE to develop directly to native resource APIs or descriptor formats is also lacking in that it doesn't necessary have an associated constraint or extensibility model. If a resource should be consistently accessed with a particular configuration or expected semantics, there is no good way to describe resource constraints for clients or to enforce that they are followed. A concrete example is a JMS queue where it is expected that messages will always conform to a specific format or contain an expected set of properties. There is no good way of representing this constraint to the client, short of runtime errors when the message does not comply with the constraint.

The lack of a single canonical representation also makes it difficult for the systems developer to collaborate with the application developer, short of constructing and exposing custom facades for client access. But even then, there is the IDE problem of knowing what facades are available, and how they should be configured and used once selected.

Without any well-defined source format for representing client resource configuration, packaging models, or discovery mechanisms, there is no non-proprietary way for the IDE to present the notion of configured resources, nor to pre-configure client access to resources.

1.4. Solution: A Unified Tooling Model

Controls, like the JavaBeans upon which they are built, are designed for tooling. Beyond the common programming model presented to developers, Controls also offer resource discovery

and property introspection mechanisms that allow an IDE to locate available Controls and present and interactively configure their properties.

Because Controls expose operations, events, and properties using common mechanisms, an IDE can support client use cases based upon these mechanisms as well as a common authoring model for defining new types of Controls, without the need for a large amount of resource-specific code.

Using a common client model allows a single base of IDE code to allow the use of a variety of resource types, based upon introspection. Using a shared model (and code) for presenting and configuring client access also results in a consistent user experience when working with resources, both on the client and authoring side. While the developer might be using a diverse set of resources in the course of building a user interface, service, or application, the learning curve from a user interaction perspective can be reduced in the same way that it is reduced from an API perspective by having a common model.

Controls extend the base properties support of JavaBeans to add support for metadata annotations, constraints, and an extensibility model, allowing an IDE to define new Control types that are pre-configured for specific resource access use cases.

The earlier programming example showed a simple customized Control defined to access an Enterprise JavaBean advertised at a particular JNDI location. This example could easily have been constructed by an IDE using JMX to explore advertised EJBs on a J2EE server, and then generating the necessary Control definition that exposes the EJB with the specific home/business interfaces represented as operations on the bean and the correct JNDI location pre-configured as an attribute.

The Controls architecture supports the definition of configuration options list for a particular Control type. This lists the base set of properties that are associated with the type and can be used to:

- Specify the attributes in the set that can be configured using metadata annotations, and the syntax for doing so. This enables an IDE to present property-style selection of metadata-based attributes and values, as well as providing the ability to validate the annotations on any usage of the type and relationships between annotations.
- Specify the attributes in the set that should be settable dynamically using property getters/setters on instances of the type. This can be used to support auto-generation of Control types with property accessors based upon the attribute set.
- Derive a schema for representing the configuration of the attribute set using XML. These can be used in common tools for state management (to persist the representation of a Control instance and its attributes as XML) as well as in an externalized configuration mechanism that allows attributes to be bound externally using deployment descriptor-style configuration files. This makes the construction of instance introspectors

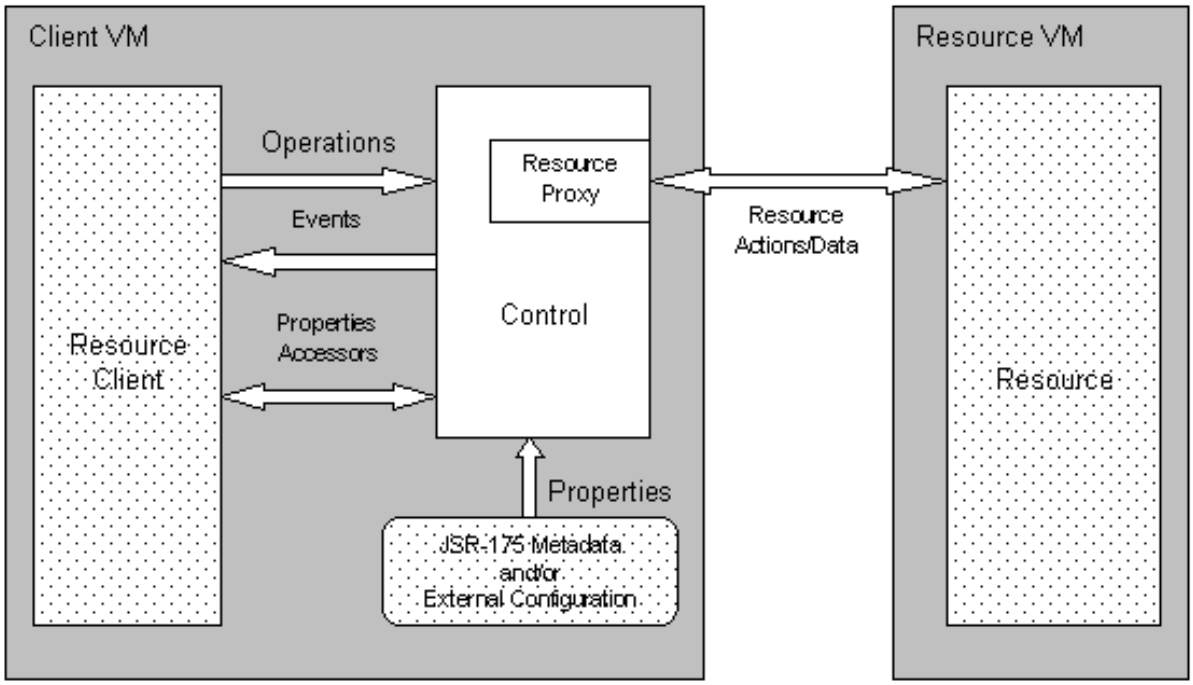
and administrative tools much more straightforward, as compared to using ad-hoc deployment descriptor formats.

Controls also provide a JAR-based packaging mechanism, for how Control types can be discovered within a jar.

The Controls architecture provides a well-defined packaging model that enables system vendors, 3rd party providers, or J2EE system developers (in the collaborative scenario) to distribute controls that offer client access to provided services or components. An IDE can then discover packaged controls to present them in a palette or list of available resource types for client use.

2. The Controls Architecture

The following picture shows the basic runtime architecture and the relationships between a resource client, the associated Control, and the accessed J2EE resource:



The Resource Client represents user code in a web application, service, or application that needs access to the J2EE resource. The Resource Client and supporting Control will always live in the same virtual machine and communicate directly using local Java method invocation. The accessed resource may or may not reside within the same virtual machine,

depending upon the nature of the resource and the application server environment.

Dynamic property accessors and resource operations are exposed on the Control and used by the client to initiate resource access. Data from the resource may be returned as return values from operations or fired as events on the bean event interface to registered listeners.

Resource access may be parameterized by metadata annotations declared directly on the Control instance, class, or method declarations, or by properties provided to the factory-based constructor. In addition to this, there is an external configuration model for how properties can be bound from external configuration (ex. deployment descriptors), enabling deploy-time binding of attributes. Examples of resource attributes that might be parameterized by metadata or external configuration or JNDI names associated with resources, resource or protocol configuration, message formats, etc.

The Control itself will often hold a reference to a resource proxy associated with the accessed resource, and will use the proxy to enact operations requested by the client. Examples of resource proxies are EJB home or remote stubs, JMS connections or sessions, web service client proxies, etc. The Control manages the state and lifetime of this proxy reference, coordinated by a set of resource management notification events that are provided to it indicating how long the proxy resources can be held by an outer container that determines the resource scope.

The actual communication between the resource proxy and the resource itself is generally a function of the underlying resource. For EJBs, it might reflect communication via RMI or local Java invocation, for web services it might be service invocation based upon the exchange of XML documents over standard web protocols.

The following sections describes some of the key features and attributes of the Controls Architecture:

2.1. Public Interface / Private Implementation / ControlBean Wrapper

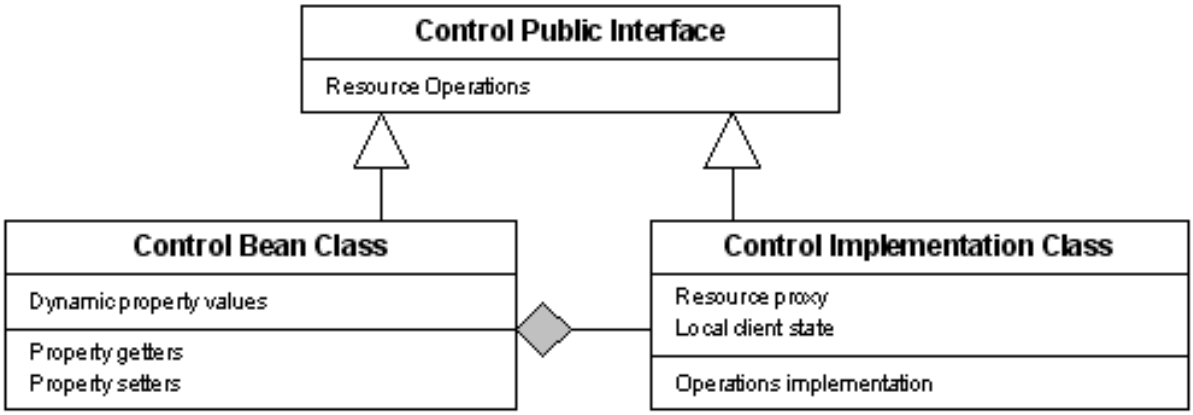
The definition of a new resource type in the Control architecture is composed of three distinct classes:

- The public Control Public Interface defines the set of operations and events that are exposed for the resource type. For example, if you were authoring a control for database access, the class DatabaseControl would be the public resource interface for the database resource. Clients to the database control would call the methods on this class to get resources from the database.
- The private Control Private Implementation class provides the implementation of resource operations as well as proxy resource management. In the case of a database control, the class DatabaseControlImpl would provide the implementation of the various

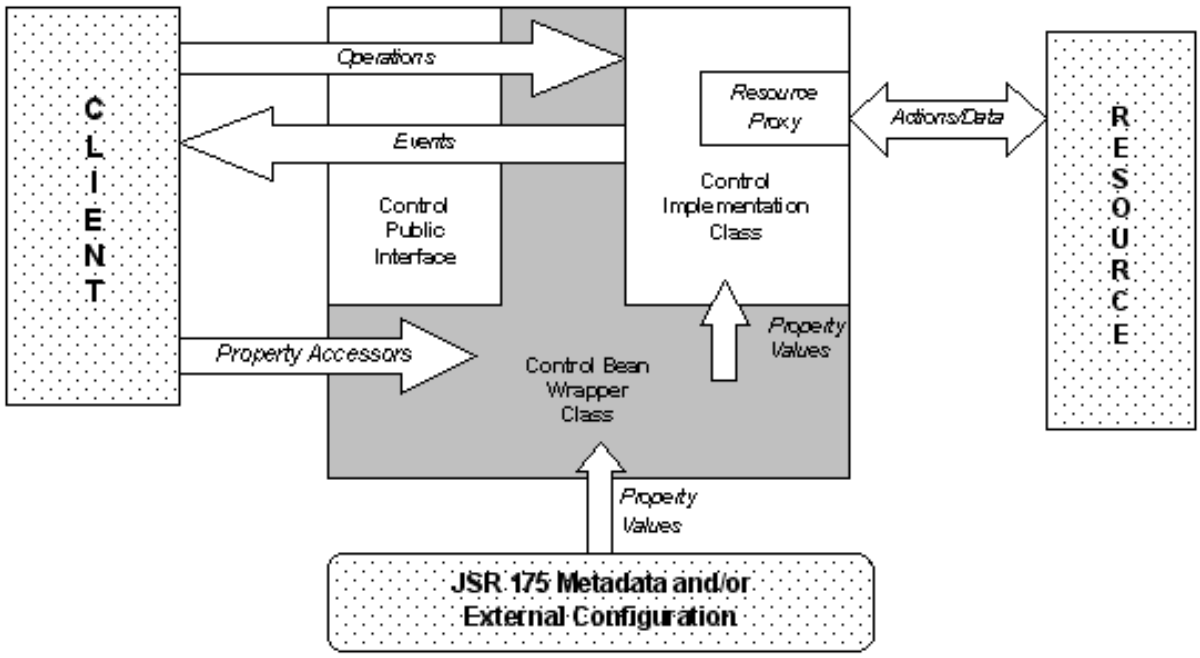
methods that access the database resource.

- The Control Bean Wrapper class is the JavaBean wrapper around the implementation class that provides the property accessor implementation (for example, through the definition of metadata annotations), per-instance storage of dynamic properties, and property resolution services. It also performs initialization of contextual services and nested Controls. (Note that the control author is not responsible for authoring this class directly: it is an artifact of the Beehive controls build process.)

The relationship and functions of these classes is summarized in the following diagram:



The following picture shows how these 3 classes work together to fulfill the runtime responsibilities shown in the earlier architecture diagram:



2.2. A Flexible Property Model

A key aspect of the Controls architecture is a flexible configuration model for how resource access attributes will be resolved. Properties can be used to parameterize resource access, providing attributes such as JNDI names for local resources, web service URLs, connection attributes, etc.

It is possible to introspect a bean and set the available set of properties. Additionally, Controls move beyond the traditional property setter/getter to provide some additional capabilities:

- Enables the assignment of Control properties using metadata annotations on Control classes, instances, or methods.
- Provides a consistent externalized property binding model, so resource attributes can be managed without requiring changes to source code.

The three property configuration mechanisms (programmatic property accessors on the Control, metadata annotations on Control declarations, and external deployment descriptor-style configuration) have a well-defined property resolution precedence that is implemented and enforced by the Control base implementation. The precedence (from highest to lowest) is:

- Programmatically set property value

- Externally configured property value
- Metadata-defined property value

In other words, the resource client can override a value defined by either externalized configuration or metadata, and a value defined in externalized configuration can override a metadata-defined value.

To ensure that this flexibility is not misused where it is not desirable, it is also possible to declaratively specify the mechanisms that can be used to set attribute values. So an attribute could be marked as 'read-only' from a programmatic perspective, and would only have a getter and not a setter, or a metadata-based attribute could be marked as bound in a 'final' way that prevents override by either external configuration or programmatic mechanisms. This is useful in the previously described collaborative scenario, where the J2EE Systems Developer who is responsible for resource access definitions via Controls might want to constrain the flexibility that the Control consumer has in modifying those definitions upon use.

In the DatabaseControl example, an attribute might exist to set the JNDI data source of the resource database. For this attribute, it might be desirable to set the value programmatically, externally, or using metadata annotations.

The declaration of the DatabaseControl metadata annotation and member might look something like:

```
@PropertySet
public @interface ConnectionDataSource {

    /**
    * The jndi name of the DataSource.
    */
    @AnnotationMemberTypes.JndiName(resourceType =
AnnotationMemberTypes.JndiName.ResourceType.DATASOURCE)
    String jndiName();
}
```

This defines a metadata attribute (@ConnectionDataSource) that has a String member value named 'jndiName'.

An example of setting the jndiName member of the DatabaseControl metadata attribute inside of client code might look like:

```
@DatabaseControl(jndiName="someJndiDataSource")
public DatabaseControl myDBControl;
```

The DatabaseControl will also advertise the following JavaBean property accessor methods:

```
public String getJndiName();

public void setJndiName(String jndiName);
```

This accessor could be used from client code, as in the following example:

```
myDBControl.setJndiName("someJndiDataSource");
```

The configuration of the jndiName member based upon external configuration can take the shape of a traditional XML configuration file:

```
    <databaseControl:databaseControl
xmlns:databaseControl="http://openuri.org/my/myDatabaseControl">
<databaseControl:jndiName>someJndiDataSource</databaseControl:jndiName>
    </databaseControl:databaseControl>
```

Note that you can turn off programmatic access and external configuration of control properties through the `hasSetters` and `externalConfig` properties on @PropertySet:

```
@PropertySet(externalConfig=false, hasSetters=false)
```

By default, these properties are set to true.

Also, @AnnotationConstraints.AllowExternalOverride allows an annotation author to configure a single annotation as externally overridable without changing the shape of the ControlBean. This is especially useful when individual method- and parameter-level annotations need to be configured externally. This is in contrast to `@PropertySet(externalConfig=true)` which causes the generated ControlBean to have getter/setter methods for *every* annotation attribute.

2.3. Resource Views: Extensibility by Interface

Controls also support an extensibility model that allows operations on a resource to be defined using a customized interface that extends the base public resource interface, and defines metadata-annotated operations on the resource. This enables the construction of

"views" or specific resource use cases, defining a more-specific set of resource operations or events.

As an example, take the basic DatabaseControl that provides simplified database access using JDBC, and hides and manages the details of how JDBC connections are acquired and released from the client programmer.

This DatabaseControl could also define an extensibility model that allows the execution of JDBC prepared statements as operations on an extended interface, and marshals the returned ResultSet back to native Java types. When extended in this manner, the resulting extended control presents a view of the JDBC resource as a set of methods that result in the execution of predefined PreparedStatements.

An example of the customized interface for this Control might look like:

```
import javax.sql.SQLException;

import org.apache.beehive.controls.api.bean.ControlExtension;
import org.apache.beehive.controls.system.jdbc.JdbcControl;

@ControlExtension()
public interface CustomerDatabase
    extends DatabaseControl {

    @SQL(statement="INSERT INTO CUSTOMERDB (ID, NAME) VALUES ({id},
{name})")
    int newCustomer(int id, String name)
        throws SQLException;

    @SQL(statement="SELECT * FROM CUSTOMERDB WHERE ID = {id}")
    Customer findCustomer(int id);
}
```

In this simple example, each operation on the interface corresponds to a SQL prepared statement to be executed. Metadata annotations on the methods are used to define the additional semantics required, in this case the actual SQL statement to invoke.

Support for Extensibility by Interface is optional. The Control author has full control of whether extensibility is or is not supported, as well as the ability to define and implement resource-specific semantics associated with extended operations on the control type.

2.4. Contextual Services

Given their use case (resource access), it is possible to use Controls from a variety of

different runtime contexts: within web tier containers (servlets, JSP, JSF), within web services, standalone Java applications, even from within EJBs. Given this diverse set of contexts, Controls have a flexible model for how they integrate with any outer container or component model and for how services will be obtained from them.

Controls may need access to contextual services to support resources. One example of client-side contextual services might be security services to access a credential repository or to provide data encryption/decryption services. Services may be contextual, because the actual implementation might vary based upon the type of container in which the Control is running. As an example, a security contextual service might provide different implementations for Controls running in the EJB tier (by delegating to an enclosing EJBContext) vs. Controls running in the Servlet container vs. Controls running in a standalone Java application.

Contextual services can also define an event model, so contextual services can also declare and fire events on Controls that have registered in interest. As an example, a basic ControlContext contextual service is provided as part of the base Controls architecture. This contextual service provides common services for Controls, such as access to properties, as well as a set of lifecycle events for Controls.

The discovery and implementation model for Controls Contextual Services will be based upon the JavaBeans Runtime Containment and Services Protocol (Glasgow) (http://java.sun.com/products/javabeans/glasgow/#containment) that is already shipping as part of J2SE.

2.5. Resource Management

The Controls architecture defines a unique set of lifecycle events and a resource management contract between Controls and the execution container they are running within. There are three primary motivations for this:

- To enable the Control implementation to implicitly obtain supporting client-side resources (connections, sessions, etc) on behalf of the client.
- To enable the Control to hold these client-side resources for an appropriate resource scope (across multiple client invocations) to achieve optimal performance and utilization of resources
- To ensure that client-side resources obtains on behalf of the client are consistently released at the end of the appropriate resource scope.

The key is that resource management should be transparent to the client. The Control resource management design makes the Control implementation class the responsible party, instead of placing this burden upon the client of the resource which is the common approach

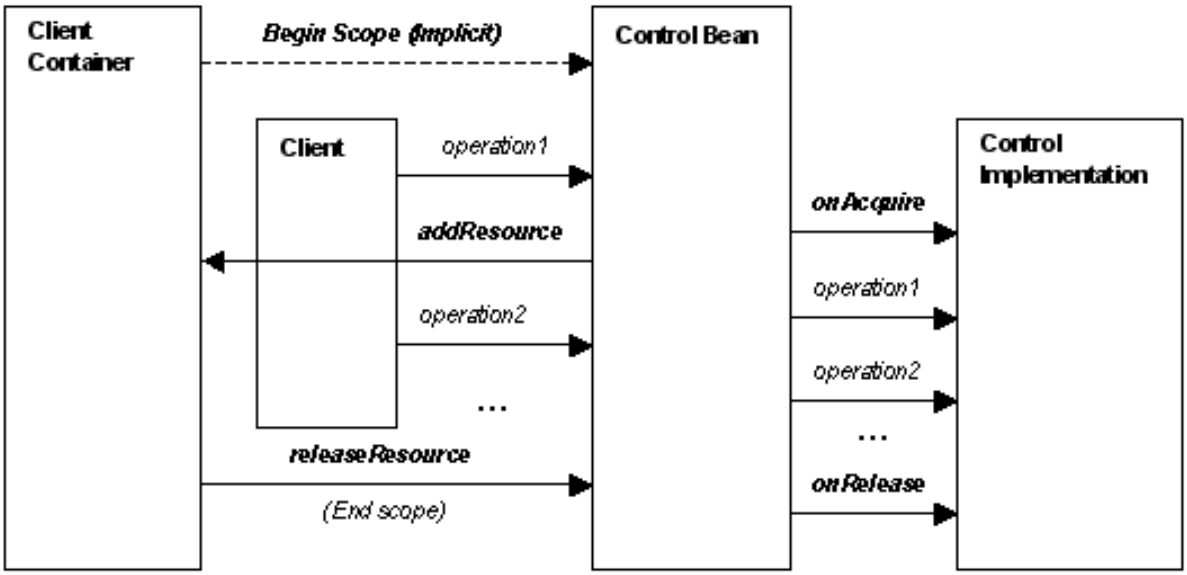
associated with most J2EE resource types.

This is achieved by defining two basic lifecycle events that will be delivered to the Control Implementation Class:

- onAcquire: the onAcquire event is delivered to a resource implementation on the first client invocation within a resource scope. This provides an opportunity to obtain any basic client-side resources necessary to support operations on the Control. For example, a Control that was providing access to a JMS queue might use the onAcquire event to obtain a JMS connection, session, and a reference to the target queue.
- onRelease: the onRelease event is guaranteed to be delivered to every control implementation instance that has received an onAcquire event during the current resource scope, at the end of that scope. This provides the opportunity to release any of the resources obtained during onAcquire event processing. For example, a JMS connection and session could be appropriately closed and the queue reference reset to null.

The definition of resource scope is delegated to the outer container within which the Control is executing. For example, if the Control is executing within the web tier, the resource scope might be bounded by the duration of processing of the current http request. For a Control running in the EJB tier, the resource scope might be the current EJB method invocation or possibly even by the current transaction context.

The following diagram shows the basic mechanics of this contract:



The Client Container has two basic responsibilities: to maintain an accumulated list of

Controls that have acquired resources, and to invoke releaseResources API on each of them at the end of the appropriate resource scope. The Control Bean is responsible for delivering the onAcquire event to the Control Implementation instance, for notifying the Client Container that resources have been obtained, and for delivering the onRelease event to the implementation when notified by the Client Container.

This diagram also demonstrates the transparency of resource management to client code itself; the client is only invoking operations, and all of the necessary underlying resource management is done by interactions between the Client Container, Control Bean, and Control Implementation.

2.6. Composition Model

The Controls architecture also supports a composition model, so it is possible to define a Control type that nests another Control type. This makes it possible to extend a physical resource abstraction with a logical abstraction that lives entirely on the client side. Composition is useful for the construction of facades or to add additional client side operations, events, or state to the nested Control abstraction.

Composition of Controls is supported using the mechanisms defined by the JavaBeans Runtime Containment and Services Protocol (Glasgow).

2.7. Packaging Model

The Controls architecture provides a simple JAR-based packaging model that enables Controls to be packaged for distribution. The model defines a simple manifest file that describes the set of Controls within a jar. Tools can quickly introspect and build palettes of available controls based upon this packaging model.

It is possible to place Control jar files at a variety of classloader scopes (system, application, or module) for client use cases.

3. The Controls Client Model

The Controls architecture actually offers two related client models with slight different characteristics:

- A programmatic client model, where the client explicitly specifies Control instance attributes to factory-based constructors, and does direct registration of event listeners and event handling.
- A declarative client model, where Control instance attributes are specified using metadata annotations, and event routing is implicit based upon a set of basic naming conventions.

The two offer the same basic functionality; but in the programmatic model the client takes explicit responsibility for construction of Control instances and event routing; in the declarative model, the Control container provides initialization and routing services on behalf of the client. The programmatic model directly exposes the details of how initialization and event handling takes place; it is likely a more comfortable environment for the professional J2EEdeveloper or one who is already comfortable with constructing and handling events from JavaBeans. The declarative model hides many of these details, making it much easier for less experienced developers (and development tools) to quickly declare and configure Control instances and create event handling code to service events.

3.1. Programmatic Client Model Example

The programmatic client model follows the basic pattern of JavaBeans construction and event handling:

```
DatabaseControl myDBControl =
(DatabaseControl)Controls.instantiate(classloader,
"com.my.DatabaseControl");
myDBControl.setJndiName("someJndiDataSource");
```

In the example above, a factory-based instantiation API (Controls.instantiate()) is used to construct a new instance of the DatabaseControl. It is programmatically initialized to the desired configuration.

3.2. Declarative Programming Model Example

The following example is equivalent to the preceding example, but uses declarative style construction:

```
@DatabaseControl(jndiName="someJndiDataSource") com.my.DatabaseControl
myDatabaseControl;
}
```

In this example, the DatabaseControl instance is declared with attributes specified using metadata annotations. There is no implicit construction of the bean instance; the client container for the ControlBean will recognize the presence of the Control declaration and will implicitly initialize the instance.