APPLICATIONS OF MODERN PROGRAMMING TECHNIQUES IN EXISTING CONTROL SYSTEM SOFTWARE*

B. Frak, T. D'Ottavio, W. Fu, L. Hoff, S. Nemesure,
Brookhaven National Laboratory, Upton, U.S.A.

Abstract

The Accelerator Device Object (ADO) specification and its original implementation are almost 20 years old. In those last two decades ADO development methodology has changed very little, which is a testament to its robust design [1], however during this time frame we've seen introduction of many new technologies and ideas, many of which come with applicable and tangible benefits to control system software. This paper describes how some of these concepts like convention over configuration, aspect oriented programming (AOP) paradigm, which coupled with powerful techniques like bytecode generation and manipulation tools can greatly simplify both server and client side development by allowing developers to concentrate on the core implementation details without polluting their code with: 1) synchronization blocks 2) supplementary validation 3) asynchronous communication calls or 4) redundant bootstrapping. In addition to streamlining existing fundamental development methods we introduce additional concepts, many of which are found outside of the majority of the controls systems. These include 1) ACID transactions 2) client and servers-side dependency injection and 3) declarative event handling.

INTRODUCTION

Java ADO framework was created to supplement existing C++ RAD based development environment. Its main objective is to simplify the Accelerator Device Object development without sacrificing any of the existing functionality. Java ADO codebase is streamlined and stripped from redundant calls in the user layer. At the same time it's more transparent than its C++ counterpart, which hides some of its complexity behind its domain specific language.

The framework is geared towards Java developers who have at least basic understanding of Collider Accelerator development infrastructure, but are not necessarily experienced RAD developers. On the other hand a proficient ADO designer will benefit from a much faster development cycles. The framework is well documented with its own set of wiki pages and numerous examples in our code repository.

Containers running Java ADOs are expected to run in a middleware layer between the front-end computers and the client space. They are a natural fit in systems, which run thin clients and need a persistent business logic backend, but they can also be used as drop-in replacements in place of existing ADO managers.

*Work supported by Brookhaven Science Associates, LLC under contract no. DE-AC02-98CH10886 with the U.S.

CONCEPTS AND TECHNIQUES

Java ADO design leverages several concepts and technologies to achieve the aforementioned objectives. Three of them discussed in this section have been around for a number of years and have been tried and tested in both academic and commercial settings.

Convention over Configuration

This fairly obvious, but at the same time very powerful software paradigm simply seeks to cut down on the inherit code complexity without sacrificing any flexibility by removing any exceptional and uncommon states from the initial design matrix [2]. The conventions have been carefully selected based on historical usage. Once set, they are for all intents and purposes immutable since new defaults could negatively impact existing applications. Java ADO framework attempts to conform to this standard in the following ways:

- It does not use any external configuration files. All required and optional configuration is weaved in the device object code as Java annotations.
- ADO parameters features are extracted from these annotations. Any missing information is inferred from the Java field context, which defines the parameter. This information includes, but is not limited to: data types, parameter and property names, category type and features. Code snipped, which illustrates this behavior is shown in Fig. 1. Note that both parameters end up with the same feature set – the top one has its name, type, count and category resolved at runtime from both the field context and the associated annotation, while the bottom declaration has its features explicitly declared.
- Default runtime behavior associated with a device object's state change automatically triggers a number of predetermined actions – some of them are mandatory and cannot be overridden, while others, which inherit their behavior from the base configuration, can. For example asynchronous notification falls into the latter category. By default all assignments, which change a value of an ADO property generate a system wide notification to all clients subscribing to this property. This behavior is an accepted default, but it can be overridden in three different ways (Fig. 2.)

ISBN 978-3-95450-139-7

```
@AdoParameter
@AdoProperty(category = AdoCategory.DiscreteSetting,
             count = 1,
             name = "value", ppmSize = 1,
             type = AdoPropertyType.String)
@AdoLegalValuesProperty(value = "one,two,three", writable = true)
private String menu = "one";
```

Figure 1: Device object parameter / property configuration.

```
@AdoParameter
@AdoProperty(category = AdoCategory.ContinuousSetting)
// suspend async update for this property
// all updates have to be done manually by calling the
// super.triggerAsync("itemS") method
@AdoManualAsync
private String item;

// suspend all async updates in this method
@AdoManualAsync
public void seItem(String item) {
   // suspend async updates for the itemS property
   // to resume the automatic updates you must call
   // super.resumeAsyncUpdate("itemS")
   super.suspendAsyncUpdate("itemS");
   this.item = item;
}
```

Figure 2: Explicit configuration of asynchronous update.

Aspects

Java ADO framework is built in its entirety around the core concepts of aspect-oriented paradigm [3]. From the moment server starts bootstrapping its devices to the last shutdown call, advices, which cut across the critical code sections, perform validation, trigger asynchronous updates, cache values and manage transactions. Without them Java ADO framework would dissolve device object code into a redundant and overstated disarray of calls to the base class, where business logic becomes a second-class citizen. Developers should not be concerned with device to server or vice versa communication or state management mechanisms unless he or she wants to explicitly change their default behavior. Well designed and placed join-points combined with a chain of single-task focused advices can almost in its entirety eliminate this code pollution, and at the same time make the code easier to manage and maintain. Additionally, because of AspectJ's point-cut flexibility, Java ADO framework essentially removes any need for value object wrappers in the device object layer. From a developer perspective every piece of ADO's stateful data can be a primitive – be it a scalar or an array, which is just one of many examples where AOP helps with a reduction of vertical layers developers need to be concerned about. This in turn promotes code readability and its general transparency.

The principal beneficiaries of AOP are without a doubt device objects parameter setters. Every parameter assignment is either surrounded or followed by the six core aspects (Fig. 3):

- Around set transactional advice, which starts, suspends or resumes current transaction (if any). This aspect is also responsible for any possible rollbacks triggered by a potential unchecked exception.
- Around set validation advice, which runs a sequence of mandatory validators followed by a set of custom ADO specific checks. Any failure throws an appropriate exception, which discontinues both the validation and AOP processing for this set.
- Around set advice, which actually sets a value of the parameter.
- After returning asynchronous advice, which notifies any clients subscribing to the parameter about its state change.
- After returning caching advice, which stores a new value on a disk.
- After returning notification advice, which alerts the alarm system about parameter's transition to another alarm level.

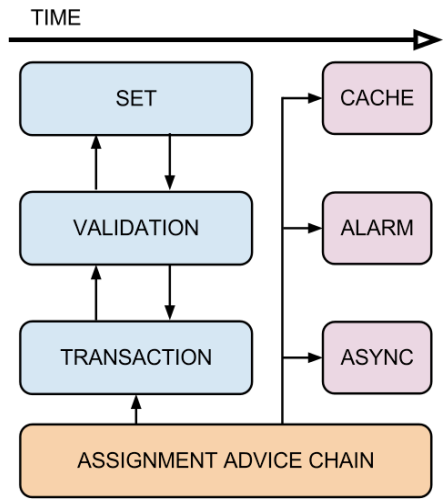

Figure 3: Assignment advice chain.

Note that the last three advices have no dependency on each other and always run in parallel. This set of actions is applied universally to all assignment operations, where left hand side operand is a field marked with the AdoParameter annotation. The aforementioned benefits are directly related to call context irrelevance during the assignment operations. For example neither call past nor its future plays any role when determining validity of the assignment – the only thing that matters at that time is a state of the device object. This "tight" coupling guarantees that all ADO parameters are always consistent - it also means that developers will have a harder time shooting themselves in a foot.

Dependency Injection (DI)

Java ADO framework relies on DI while bootstrapping both the sever container as well as its device objects. The former relies on DI supplemented by a bytecode generator to instantiate and initialize appropriate class proxies. This process implements common base class' abstract methods and generates additional constructs, which supplement the built in validation subsystem.

ADO instances utilize dependency injection for their local or remote device dependency. In either case

ISBN 978-3-95450-139-7

developers simply declare a base interface, which gets filled in by the framework during the initialization stage. Fig. 4 shows an example where the bootstrapping process makes a decision on which type of Ado instance to inject based on the supplied name. Dependency injection implemented in the Java ADO framework is fully context driven. Developers can specify additional parameters, which alongside the supplied name will help the underlying injector with its evaluation. Any unresolved names, unless marked as optional, will terminate the device object's container.

```
// local.test in an ADO running in the same container as
// this instance – the injected type will derive from
// a LocalAdo type
@AdoInject("local.test")
private Ado localAdo;

// remote.test in an ADO running in another container
// either another manager or FEC – the injected instance
// will therefore derive from a RemoteAdo type.
@AdoInject("remote.test")
private Ado remoteAdo;
```

Figure 4: ADO instance injection.

IMPLEMENTION DETAILS

When combined the concepts and techniques described in the previous section open up new possibilities for developers designing Java accelerator device objects. This section will cover a few of these features, which are either not available in the existing C++ ADO framework or their implementation differs substantially from the current design.

Transactions

Java ADO framework implements a standard ACID-like transactional model seen in many popular databases. ACID [4] convention is always guaranteed within server containers running Java ADO implementation. When interacting with legacy systems the framework has to commit device object values within any running transactions, which means that they will be available for reading before transactions commit or rollback their actual states. Rollbacks are still automatic, however the ACID guarantee is no longer applicable. From an API perspective the implementation borrows heavily from the Enterprise Java Bean 3.x model. ADO developers mark methods as transactional using an AdoTransactional annotation. For a finer grained control they also have an option of starting and terminating transactions within methods. Declarative model supports three attributes, which control the transaction context propagation.

- REQUIRED – If invoked outside a transactional context the container will start a new transaction, which will commit or rollback at the end of the method marked with the associated AdoTransactional attribute. If the current thread is already in a transactional context the method executes within that context.
- NOT_SUPPORTED – If invoked with a transactional context current transaction is suspended for the duration of this method call. If invoked outside the context this attribute has no effect.
- SUPPORTS – If invoked within an active transaction it behaves like the REQUIRED case. Invocation outside a transactional context, analogously to the NOT_SUPPORTED attribute, has no effect.

Transactions in the Java ADO framework simplify the state management of local and remote objects, primarily because any rollbacks caused by invalid states or failures are automatic. Some of them maybe happening without ACID guarantees, but that's an acceptable penalty when dealing with legacy systems. Fig. 5 shows an example of an automated versus manual state management. In this example both intS and longS parameters have strict validation checks applied to their values, which might trigger an exception on assignment. We want to make sure that we keep the values of both parameters synchronized, which means in case of an error we want to revert back to a previously known valid state. Transactional method allows us to achieve this objective in just two lines of code. Manual management requires five times as many lines. On top of that we might be modifying values that don't need to be rolled back. For example if the intS assignment triggered the exception we do not need to roll back longS. To fix that we'd need to add additional checks to the manual method. Transactional approach suffers from none of those drawbacks.

```
@AdoTransactional
public void automatic(int value) {
   this.intS = value;
   this.longS = value;
}

public void manual(int value) {
   int tmp1 = this.intS;
   int tmp2 = this.longS;
   try {
      this.intS = value;
      this.longS = value;
   } catch(ModuleException e) {
      this.intS = tmp1;
      this.longS = tmp2;
      throw e;
   }
}
```

Figure 5: Java ADO transaction example.

Value Bonding

Historically interaction with the remote control points has been reliant on Request objects, which expose a standard set of RPC methods available on the remote systems to the clients. Natural extensions to this approach are client side object proxies, which realize remote instances as local constructs (Fig. 4). Java ADO framework allows both types of approaches. In addition it allows for a third type of interaction - and that is value bonding.

ISBN 978-3-95450-139-7

This feature allows ADO developers to closely couple local field value, which could be a device object parameter, with a parameter in another ADO instance. The binding can be either bidirectional or unidirectional. This behavior depends on the AdoBond annotation applied to the local field (Fig. 6). “Incoming” parameter controls the flow of inbound data. If set to true all reads on a local field marked with this annotation are guaranteed to have the latest value of the bonded parameter. Analogously setting the “outgoing” flag to true guarantees that the other side of the bond will have its value updated to match the just updated local side.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface AdoBond {
    String [] context() default {};
    String device();
    String parameter();
    String property() default "value";
    int ppmUser() default 0;
    boolean async() default true;
    boolean incoming() default true;
    boolean outgoing() default true;
}
```

Figure 6: ADO bonding annotation declaration.

Value bonding fits nicely in the middleware platform model, where Java ADO managers often expose or update data from a lower tiered systems either in an unchanged or computed format. Fig. 7 shows an example of outgoing bond, where remote side has its value updated in tandem with its local counterpart. The update occurs in the post assignment advice, which runs synchronously with the operation. Finally bonds like all other framework constructs participate in the ADO transactional model.

```
@AdoParameter(name = "timerIntervalS")
@AdoProperty(category = AdoCategory.ContinuousSetting)
@AdoBond(device = "simple.test", parameter = "timerIntervalS")
private int timerInterval;

public void updateTimer(int value) {
    this.timerInterval = value;
}
```

Figure 7: Value bond in action.

SUMMARY

Java ADO framework provides an attractive alternative for seasoned RAD developers, who are not willing to sacrifice any features, but at the same time want to gain a new perspective on device object development model. The framework is even a better fit for developers, who had limited exposure to the ADO design model and who are building a system, which could benefit from a stateful backend.

REFERENCES

[1] L.T. Hoff and J.F.Skelly, Accelerator Devices at Persistent Software Objects, Nucl. Instr. and Meth. in Phys. Res. A 352 (1994),

[2] N. Chen, Convention over Configuration (1996) http://softwareengineering.vazexqi.com/files/pattern.html

[3] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, C. Boyapati “Aspect-oriented Programming”, (1999)

[4] T. Haerder, T. A. Reuter, "Principles of transaction-oriented database recovery". *ACM Computing Surveys* 15 (4): 287, (1983)

ISBN 978-3-95450-139-7