GENERIC TYPES IN JAVA: ABEANS<T> SPECIFICALLY FOR YOU, MR T?

G. Tkačik, Cosylab, Ljubljana, Slovenia

Abstract

Java 1.5 is not merely a revised Java 1.4 – it is an old language reborn to incorporate generic types, similar to C++ templates, and a host of various new improvements. Abeans, a client-side software library for modelling complex control systems, has been trying for several years (and three major releases) to strike a balance between ease-of-use and the capability of running on various control system architectures. More specifically, we designed custom solutions based on introspection, meta-data processing and code generation in order to foster existing and new Abeans deployments on diverse machines. New Java features will drastically reduce the code footprint and improve code clarity by generalizing naturally over atomic data types in the control system; by incorporating meta-data relevant for generic applications and code generators directly into source code itself; and by allowing Abeans code to monitor and modify itself during execution, a feature impossible to implement in pure Java before version 1.5. We believe that, if used properly, such advanced functionality can be a blessing to control system programmers. Consequently we discuss specific examples where the whole community can benefit from it and show how new concepts will be incorporated into future Abeans releases.

INTRODUCTION

Let me start by quoting from a recent article on JavaWorld that discusses lessons learned from experience with Enterprise JavaBeans up to version 2.1 While pointing out great successes of EJB, it notes that: *"…However, EJB, a core component for middleware in the J2EE stack, has gained the reputation of being too complex and difficult to use, especially for small to midsize business applications. The overhead of EJB infrastructure code and deployment descriptors drains both server resources and, more importantly, developer productivity. Developers often end up writing and maintaining more infrastructure code than business logic code."* [1] To remedy the situation, it points out that development should be driven along the following three guidelines*: "(Firstly), the framework should not impose an arbitrary component model to developers, as such models break the object-oriented design structure. In other words, the framework should support POJOs [2] that developers can extend and reuse inside and outside the application container. (Secondly), the framework should eliminate the need to manually write verbose and often excessive EJB deployment descriptors. A POJO should be able to simply declare what container services it needs. (Finally), the framework should support local access to the POJOs by reference. Java object serialization is slow and should be avoided when possible, especially for most small to midsized applications."*

This lengthy citation illustrates exactly the problems faced by control system community when it wants to develop portable software suites that can run on multiple platforms as opposed to specific solutions (specialized application in one lab), see "Lessons learned so far" [4]. While we are not placing so much emphasis on transactional, persistence and security aspects (yet), we are creating a system that models any machine (as opposed to a business process) and provides a range of services that often cannot be turned into components, since they are interwoven into the business logic code [3].

More specifically, creating Control System Office (CSO) will involve the definition of basic data types exchanged between the application and underlying layer, the parameterization of data sources (sources which provide data from the remote side) and data sinks (application components), and the common views and actions that operate on these data [4]. While modeling the operations over data is clearly an established OOP task, the inability or unwillingness of the community to start by abstracting away the details of the underlying system (as is the case of Abeans Release 3 plugs and models), coupled to the learning and performance overhead carried by abstraction of this kind, makes us seek for another solution: using meta-data to enable the office components to talk to the underlying system without trying to force it to conform to some predefined (and often limiting) design. As a consequence worthy of note, however, we lose the strict vertical separation of layers, and now the underlying layer details become coupled to the application code.

For example, there are the "grand-problems" of creating and describing hierarchy in an uniform way even if unsupported by underlying system or handling different models of data acquisition, such as synchronous and asynchronous; as well as a big number of seemingly smaller but nevertheless crucial differences in the ways people represent timestamps, completion codes, conditions for value-delivery update and so on. We will try to demonstrate how Java 1.5 Annotations, JMX and Generics can help develop a CSO without insisting on standardization of such issues (which seems unlikely). Our goal is then to retain access to the underlying *functionality* without remodeling it, but getting rid of the syntactic coupling; e.g. CSO should be able to talk to either channels of various kinds or even devices with minimal assumptions about their structure, but should not use directly the syntax of the underlying system.

GENERICS

Generics are parameterized types very similar to templates in C++. Learning by example is probably best, therefore let's look at the classic use in containers:

```
 List<Integer>      myList      =      new
LinkedList<Integer>();
 myList.add(5);
```

```
for (Integer number : myList)
     System.out.println(number);
```

Angular brackets denote a parameterized type, in this case a version of linked list that holds only integers. On second line you can notice automatic conversion between primitive (*int*) and its *Object* type (*Integer*), and the third line demonstrates the new *foreach*loop format. There are many good references for generics on the web [5], and we will not discuss their syntax further here.

Beyond standard benefits of their use (compile-time checks vs. run-time *ClassCastExceptions* when used with collections), there are two specific areas where generics can play a big role in CSO, namely modelling elements that depend on data type (such as channels) and management and use of metadata using Java reflection.

Currently, Abeans datatypes library that defines channels contains a lot of repetitive code: there are separate classes for *DoubleChannel*, *LongChannel*, *StringChannel* and so on. While this was a design decision that should increase the performance (by avoiding casts to *Object*) and readability, it also resulted in considerable code bloat, and the synchronization that keeps a *DoubleChannel* sematically aligned with *LongChannel* is done by hand. An equivalent generic type, which for simplicity contains just a mutator and accessor for the dynamic value [6] is clearly

```
interface Channel<T> extends Model
{
     public T getValue();
     public void setValue(T newvalue);
     …
}
```

For example, one might declare channel template that works with numeric types as follows:

```
interface Channel<T extends
java.lang.Number>
{
     …
     public T getMaximum();
}
```

This relatively straightforward example (which does not become much more complicated in practice) is a showcase for how generics decrease the codebase by a considerable factor. The other significant use is in declaring and manipulating metadata. Since using metadata is generic, its use on the modelling layer is carried out through Java dynamic mechanisms. If you have used classes from *java.lang.reflect*, you realize that up to now such code has been overflowing with class casts, dynamic type checks (*Class.isAssignableFrom()*) and ultimately resulted in an unreadable jungle that required a lot of debugging. Since Java 1.5 Java reflection has itself been redefined with generics, so that now, for instance, a *Class* is actually *Class<T>*, where *T* is the class being described. Paralleling this structure of a class *T* and its "meta-description" *Class<T>*, Abeans have been using the notion of descriptors stored in the directory for all modelling elements. For example, for interface *Channel* there has been an equivalent *ChannelDescriptor*, which enumerated requests that can be done on a channel, as well as the constituents of a channel [7].

From Java 1.5 onwards, the *Descriptor* can become a generic type, so that a descriptor for a channel can be *Descriptor<Channel>*. To see how this makes dynamic invocations much safer and clearer, suppose that one would want to create a factory for channel objects based on the type described by the descriptor:

```
interface ModelFactory
{
     public static <T extends Model> T
newInstance (Descriptor<T> desc, URI
target);
}
```

Notice how generics allow one to express the requirement that the argument of the generic method (which itself is a generic type *Descriptor<T>*) must match the return type of the method, which is *T*. Prior to generics, this would be very cumbersome, as one would have to invoke something like:

```
Channel ch =
(Channel)myFactory.newInstance(cd,
"target");
```

and the implementation of *newInstance* would have to introspect *cd*, find its class, and somehow from the class name infer what the run-time type of the return value should be (e.g. one would take the class of *cd*, which is *ChannelDescriptor.class*, get its name, rip off "Descriptor", and do a *Class.forName()* for "Channel", and create a new instance of that). Without generics, this would be a breeding ground for errors, as well as being difficult to learn for beginners.

Speaking on a more general ground, Java generics are a convenience, reducing the number of casts and improving type-safety of the code, but bringing no fundamental new ways of programming. This is in stark contrast with C++, where templates that are expanded by specialization actually allow things that could not be done before (such as template compile-time programming). Nevertheless, the difference in *practical use* of generic versus non-generic programming in Java will be immense, because it works so well with introspective features that are nowadays so widespread in the code because of their power to tie framework components together.

In accordance with the conclusion that excessive modelling should not forced onto the existing models (such as EPICS channels), the Control System Office [6] design would call for only two basic modelling elements, *Target* and *Namespace*, and their respective descriptors *Descriptor<Target>* and *Descriptor<Namespace>*. In the directory, each name would be bound either to the target descriptor (if the name was a leaf node in the directory tree; target nodes are capable of processing *Requests*) or to the namespace descriptor, if the node is not a leaf. For example, *abeans-EPICS://linac/ps1/current/value* could be bound to *Descriptor<Target>* and could process get requests, set requests and so on; in contrast, *abeans-EPICS://linac*, *abeans-EPICS://linac/ps1* and *abeans-EPICS://linac/ps1/current* would both be bound to

namespace descriptors. Although all functionality can be accessed from these directory entries, it is possible now to introduce a notion of types.

Let us declare a *Channel extends Namespace*, and associate with it *Descriptor<Channel>*. Create a node *abeans-types://Channel* that is bound to *Descriptor<Channel>*. Since a channel is a namespace, it should contain targets, such as *value* (a dynamic value), *maximum* (an example characteristic) and so forth. Now instead of putting into *abeans-EPICS* subtree, for each channel, its whole structure (i.e. *value*, *minimum* and so on), just say that *abeans-EPICS://linac/ps1/current* is-of-type *Channel*. In [6] we have discussed how is-of-type relation itself can be implemented in the directory, which finalizes the introduction of types in such a way, that even if the application is unaware of their existence, it will still work (i.e. an application-initiated request to look up the directory for *abeans-EPICS://linac/ps1/current* will find out that this is entity of type *Channel*, which is composed of *current* and *maximum*, among other things, and this is what the application will get returned for its query; the resolution to go from *abeans-EPICS* to *abeans-types* being done completely within directory code unknown to the application). Even though such manipulations would be possible without generic types, they would be simply to cumbersome to manage.

ANNOTATIONS

The second new feature that we discuss in detail is Java annotation mechanism, defined by interfaces in *java.lang*, *java.lang.reflect* and *java.lang.annotation*. The simplest description of an annotation is that it is a piece of information, attached to a Java introspectable element: a package, a class or an interface, a method, a constructor, a method parameter and so on. Such information is inserted into Java code with a special notation that we touch on later; it can be maintained in class files and JVM and introspected during runtime. For example, I might *tag* all methods of my class that do risky things with a *@RequiresAuthentication* annotation; any user, supplied with my jar file will later be able to ask JVM, for each method, if it is annotated with *@RequiresAuthentication* – and moreover, that user might write the code that pops up a window asking for username and password whenever such method is encountered in the execution flow.

Annotations in Java are actually much more powerful than simple tags, such as *@RequireAnnotation* [8]; they can be defined in a way similar to interfaces, and can be constructed with arguments. For example, consider defining the following annotation:

```
public @interface FeatureRequest
{
     int ticketID();
     String synopsis();
     String owner();
}
```

If there is a feature request for a certain class *X*, I can place in front of the declaration of *X* into Java code the following snippet:

@FeatureRequest { ticketID = 20; synopsis = "Add another feature request annotation here"; owner() "gtkacik"; }

It is important to realize that the feature request information gets packaged into bytecode, and is now available to tools that introspect such code (one tool called *apt* is provided with Java 1.5).

There were several motivations for the inclusion of annotations into Java standard: firstly, they provide great means of automatically processing code documentation, as a vastly more capable javadoc tagging system. Secondly, they make writing code generators much easier – there is a lot of boilerplate code required by J2EE, CORBA, RMI and so forth, and keeping all files in sync (IDL, XML descriptors, remote interfaces) was getting to be bigger and bigger burden. And lastly, annotations offer a way of doing aspect-oriented programming, changing the context in which the code gets executed. It is important to realize that annotations should not change the basic functioning of the code: this is up to the actual implementation. But consider a classical example: two updates to a database can be done with or without a transactional context (i.e. guaranteeing that they both succeed, for instance). However, the programmer that codes these database accesses might not know who the persistence provider will be (and if transactional update will be available when the code is deployed). Now it is possible to annotate his or her method with, suppose, *@Transactional*, indicating the intention that transactions should be used if they are available, but not polluting the code with programmatic use of transactional API and not introducing compile-time dependencies on it.

The general scenario is as follows. In a system such as CSO, there will be components produced by different programmers, and deployments that have widely varying requirements. In a medical accelerator one might need to make sure that all remote requests are transactionally stored into the database for later tracing; or that each method that changes the irradiation dose requires authentication. Without annotations, all such methods have to access the framework in code: the programmer needs to know how to access the tracing or security service, which has to have fixed API, has to be instantiated in a well-defined initialization procedure prior to code execution and so on. These so-called "cross-cutting concerns" are a point where pure OOP starts failing to provide replaceable and independent componentization.

With annotations, one writes one's own code as POJO (plain old Java object) that does not reference framework functionality. If such framework support is desired, however, the appropriate methods, classes, parameters and any other relevant elements get annotated by the programmer; and when the code is being executed and if the requested services exist in the framework [9], actions appropriate for each annotation are taken ("@Transactional" methods get recorded into the database, "@Authenticated" methods ask for credentials,

and maybe the annotation also carries information about what level of authorization is needed, for instance).

Such mechanism is vastly useful for Control System Office. Consider two applications, for example. One requires not only the dynamic value data (such as currents in 100 power-supplies), but also the accompanying Quality-of-Service (QoS) data that exists in Abeans currently for all data sources (the timestamp of the last acquisition, completion and alarm codes, ID of the data source, error stack and so forth). The other application requires just quick access to values for a large number of devices. In this case it would be very convenient if the programmer of the application were able to tell the framework that keeping track of all QoS baggage is just an overhead. Currently, this could be implemented either by making some sort of configuration switch (in which case the programmer has to know how to deal with configuration files and service) or invoking a certain method somewhere in the framework. The point is that in both cases something has to be "switched" in a place *other* than in the actual application itself, and it creates a dependency (either a knowledge dependency of the inner-workings of the configuration or an actual compile-time one for API reference). Making an annotation in front of the application class is much easier, and it is also stored along in the same Java source file.

In addition, annotations can be used to give data semantic meaning, in case where it is not obvious from its syntax. We have argued in [4, 7] that in CSO setting, it is important for various components to know how to interpret the data: an array of doubles can be a profile indexed by device, or a time series of values from a single device, and visualization of each is different (for a trivial difference, consider how would one render the x-axis in both cases on the chart). Annotating can be used to supply additional data on various levels of detail (the highest being the annotation of method parameters for each method)

CONCLUSION

Remembering the quote that opens the introductory section, we see that even in J2EE created by professional programmers, the issue of complexity of use was a serious drawback for the users. Abeans Release 3 face similar issues, all stemming back to the fact that Abeans try to accommodate various communication layers below and arbitrary application models above. The solution taken in R3 was to abstract all data exchange to the common denominator, i.e. through plugs and Abeans Engine Request/Response mechanism. On top of this engine, Abeans build their own models, namely the *Channel* model and the *BACI* model.

One can imagine that using the same design principles to create an office suite, a task of larger proportion, given the programming know-how resources of Cosylab and our field, would be an overwhelming task; even if successful, the creation of applications for such office could prove too cumbersome. It is therefore our proposal to leverage new Java technologies, such as generics and annotations, together with a new set of design principles laid out in [7]. The goal of the endeavour would be to create a set of data-source and data-sink components, living in an **existing** framework (such as Java JMX), tied together by **meta-data** stored in a **directory**. Office applications therefore become pieces of code that create **bindings** between sources (managed by next generation of engine that talks directly to communication layer such as CAJ) and sinks (visual components, for instance), and use annotations to specify details for these bindings.

Control System Office would then become much more loosely coupled collection of objects, and annotations would finally provide the mechanism for what we have desired to do before but lacked means of doing so: moving as much as possible from executive to declarative syntax. Of course, not everything is reducible to declarative form – algorithms (equivalent to J2EE business logic) are not; however, a large majority of applications consist of shuttling the data around and playing with its packaging, format and delivery options, which should be representable best by mechanisms specifically tuned for doing this: XML for persistent storage of components, directory for run-time storage of components, and annotations for both persistent and run-time storage of cross-cutting (aspect) oriented metadata.

REFERENCES

[1] Dr Michael Juntau Yuan, http://www.javaworld.com/javaworld/jw-02-2005/jw-0221-jboss4.html

[2] POJO stands for Plain Old Java Object, i.e. a component that can exist outside framework and is not encumbered with framework classes or interfaces that it has to extend or implement.

[3] See http://www.jroller.com/page/colyer/20040531 for a good explanation of Aspect Oriented Programming concept of *weaving.*

[4] G. Tkačik, Beating Commercial Products: Control System Office and Integrated Development Environment Are The Way To Go, PCaPAC 2005, Sokendai, Japan

[5] http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, Sun's tutorial on Java generics

[6] For definitions of dynamic value, characteristics and o ther concepts relating to datatypes, see Abeans Datat ypes Specification http://www.cosylab.com/abeans/datatypes-common/apidocs/index.html

[7] G. Tkačik et al, "A Reflection on Introspection", ICALEPCS 2003, Gyeongju, Korea, October 2003

[8] Note the at-sign @ prepending the annotation name: it looks very similar to the javadoc tag: indeed, *@Deprecated* is now a regular Java annotation that can stand in front of deprecated methods, for instance.

[9] One cannot, of course, get rid of *all* references to the services; however, the purpose is to reduce 1-to-n requirement (1 logging service to n references to it all over the code, for example) to something simpler.