Constants Browser
A GUI Browser for the CLEO III Constants Database

John Bicket
College of Engineering, Cornell University, Ithaca, NY, 14853

Abstract

We have attempted to develop a Graphical User Interface to access the Constants Database used by the CLEO III experiment at Cornell. We will describe in detail the technical approach we have taken in accessing this database and the many difficulties we have encountered.

Introduction

The CLEO III Constants Database stores "constants." Constants are sets of numbers, or parameters, found via calibrating the CLEO detector; these parameters allow to convert electronic signals taken by the detector into physics variables that a CLEO physicist can use to do data analysis.

This database is implemented using the Objectivity/DB, Inc., distributed object database management system (ODBMS) with language bindings for C++ and Java. The constants are stored via the C++ binding. We have chose to use the Java binding for our Constants Browser because of Java's powerful Graphical User Interface (GUI) capabilities.

We will first discuss the various options for this project and why we chose Java. We will proceed with the design of the Constants Browser. The many technological problems we encountered will be explained in detail. We will close with the current status of the project and give an outlook of the future plans.

Design and Goals

This Constants Browser is designed to serve many purposes. It first should contain a GUI that is easy to interact with and very intuitive. It should also be able to display and edit the constants information in the database, depending on the type of data the program is dealing with, along with writing entirely new constants to the database. After this is accomplished, the Browser should be able to construct "meta–versions" of the constants where we can give a symbolic name to a version of particular constants. Finally, the Browser should be able to represent graphically the versions of the constants that are currently in the database.

Choices

Objectivity has provided bindings for C++, Java, and Smalltalk. But our options for solving this problem were not limited by these bindings. We also could have used CORBA to communicate with C++ that was already written and write a Java application with this. Also, a scripting language could have been used to construct the GUI and connect with underlying C++ code to take care of communicating with the Objectivity database. We decided to use Java [2] and the extensions that Objectivity [1] provided and we have been

utilizing the Swing API, a set of tools to create a GUI, which Sun just released in their last Java release. It has many advantages:

1. Portability - writes and compile once, and we can use it across platforms
2. Intuitive - Rather than force new users to learn a command line system, we would prefer to make an application that was easy to use.
3. Code - Objectivity's Java bindings and Java code are, for the most part, easy to read and maintain.

The only considerable disadvantage to working with Java was its slower speed, but this should not be a problem for the type of interactive program we are developing.

Using Java

Since most of the database code was constructed with C++, the extensions that Objectivity provided will allow us to access the database in Java. As a direct result of using Java, a couple of unique problems arose. First of all, Java does not fully support multiple inheritance. It is possible to use interfaces, but since the object hierarchy must be matched with the C++ side that was already written, we had to modify the database structure to use single inheritance structure. We derived all objects in the database from a common base class, which we called `CODIConstBase`. At the same time this solved a bug in the Objectivity Java package dealing with their base class. We had been receiving exception errors when scanning the database for all the `ooObj`s in the containers, but deriving our own base-class solved this problem. After contacting the company, we learned that this problem was a documented bug. We also learned that the ODMG [3] classes that we were using in the database are not supported in Java (ODMG is a standard that dealt with object databases) and were able to eliminate the ODMG classes from the database (they weren't being used under the current implementation). Also, the implementation of the Java swing classes is not thread safe; the application must be single-threaded. This factor also forced us to separate the Java GUI classes from the classes that interacted with the Objectivity Database (which is the sensible thing to do for design structure regardless of this fact).

We have developed our Constants Browser to support modular constant types. Using the built-in class loader, when a class is loaded form the database, it looks for the definition (a .class file) in a predefined area. It dynamically loads the class and supports the use of that class on the fly. At runtime, the Constants Browser dynamically loads the type definition, and uses Reflection to look into the class definition. From here we can review the type of the constant, get the information it contains, and then display it. By treating Classes and Methods like real objects, Java's Reflection [2] lets us get the class type, get public fields of the class, and get public methods of the class on the fly. This ensures that none of the constants definition is hard coded into the Constants Browser. Instead, if we need to add a new type of constant, all that must be done is to define the type and place the definition into the path. This process is shown in Fig. 1 This allows us to wait until runtime to obtain the information we need about the data and lets us create a much more flexible browser. It could, in reality, browse through any objectivity databases that we supplied type definitions for, not just the constants database. Also, when we finish the browser, if we need to change the

definition of certain data type, it does not require another compile of the actual Constants Browser application.

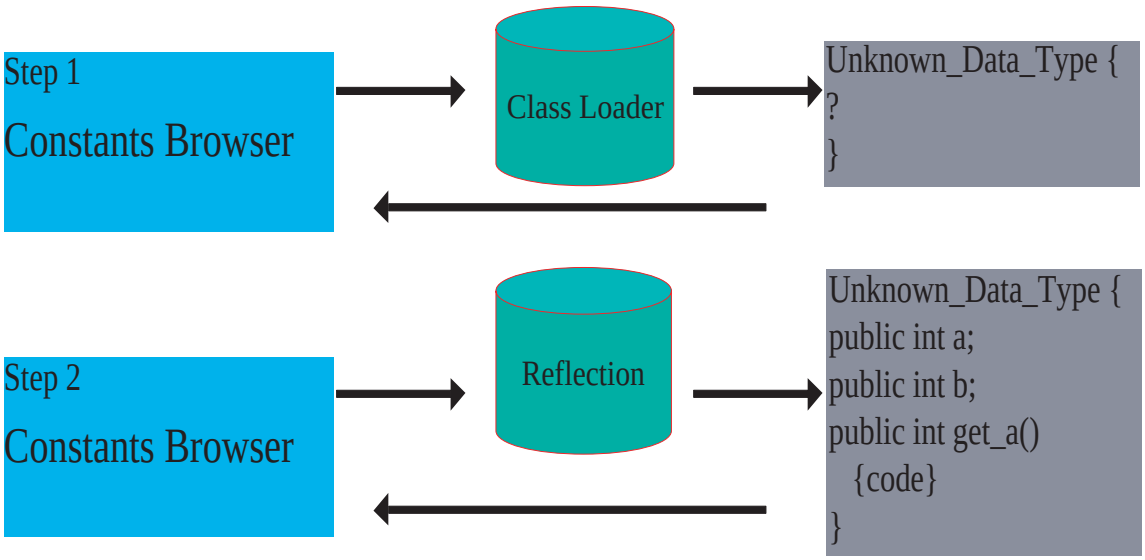

FIGURE 1. This is an example of how the class loader and reflection work together to dynamically load the constants data type. During the first step, the Constants Browser makes a call to the Java Class Loader to load the Unknown_Data_Type. It retrieves the class and the Browser then instantiates a variable. Next, using Reflection, the Constants Browser can determine the public fields and methods of the Unknown_Data_Type and interact with them.

The Objectivity Database Schema

The Schema [1] in the Objectivity Database is what contains a physical representation of the data types in the database. Each federation database contains a Schema; it keeps track of the types of the objects in the database. Every time an object is accessed through a program we wrote, the program and the database consult the schema to determine if the definition of the types we have are identical. If it is not, it complains and in Java you get an exception error. This process is shown in Fig. 2

We have been able to set the schema to the right values for primitive types (i.e. types that are not user defined or built in to the language (classes like strings, integers, longs, etc.)) User-defined classes are supported if they do not contain arrays. The problem with arrays resides in the fact that Java and c++ treat data differently when dealing with memory allocation.

In Java, when a new class is instantiated (when an instance is created of a class) it 'embeds' or makes a memory allocation for most primitive types (like ints, longs, doubles) but just creates a default pointer (a reference) to non-primitive types or arrays and does not necessarily allocate memory for that object. C++, on the other hand, will embed the

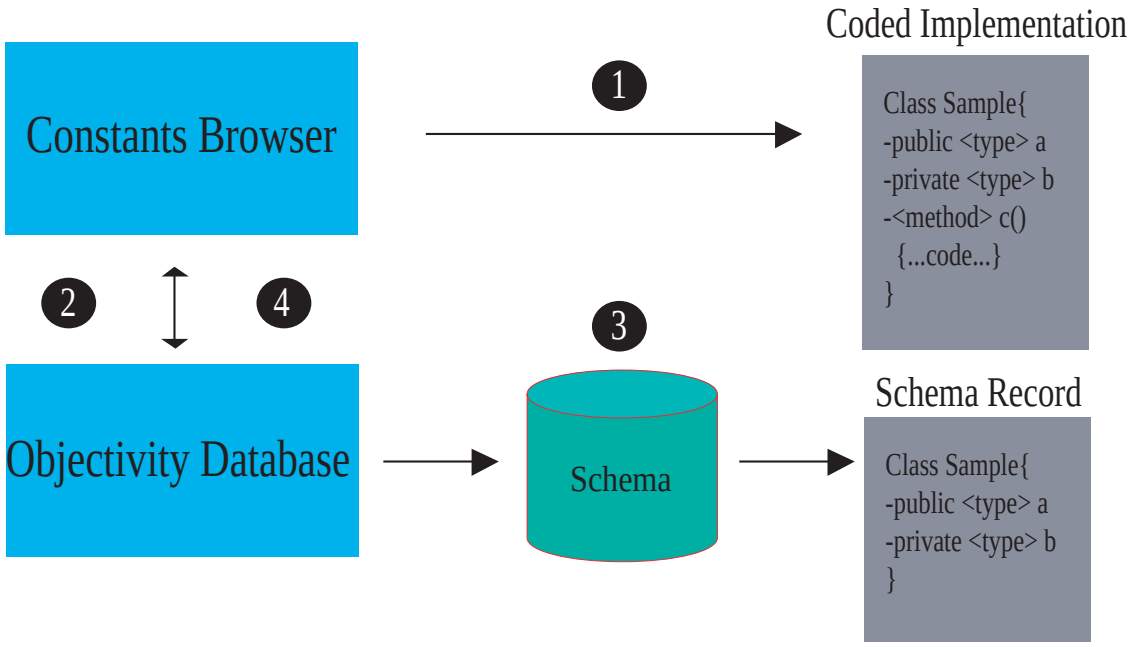

FIGURE 2. A demonstration on how the Database and the Schema work together. 1. The Constants Browser loads the class Sample to be instantiated. 2. The Constants Browser now requests a particular instance of the Sample class from the Objectivity Database. 3. The Objectivity Database consults the Schema and compares the Schema Record and the Coded Implementation of the class, ensuring that the members are identical. If they are not, an exception is thrown. 4. If the comparison is successful, the Objectivity Database returns the instance of the class Sample that was requested from the database.

array into the object. The problem is with arrays; if any type of array is created, Java would create a reference (a type of pointer) to that array while c++ would embed the object. This would not present a problem except that to create a reference to an object in the Objectivity database, the object that is referenced must be persistent (in this case meaning it must derive from a base class `ooObj`). But arrays are only persistent in C++ if they are embedded. If the are not embedded, we cannot link to them. This catch-22 prohibited us from reading the C++ databases.

After contacting Objectivity with this problem, they replied back and gave us some previously unreleased implementations of internal arrays. This will let us write types in C++ and read them in Java. After receiving confirmation that these new types are the only possible means to read complex C++ data types in Java, we will have to re-write parts of the C++ Constants Database code to support the formerly unavailable data types.

Issues with the Schema also arose with the implementation of our data objects; although the schema records the structure of the objects in the database (members in the data structure), it does not record the implementation that you use to interact with that object (i.e. it does not record methods or functions). This was not much of a problem because all the data classes that have been used so far are straightforward and do not involve any complex implementation. We have also developed a convention regarding accessing and setting variables,

modeled from JavaBeans. Each field (as long as it is not an association), if it is a public field, has a method called `get_member_name()` that takes no parameters and returns the member. If we want to be able to manipulate this field, we define a method called `set_member_name()` that takes one parameter that the field is set to. Using Java's reflection classes, we are able to scan over all the methods in an object. When we want to display an object's fields and member values, this allows us to filter out the methods that do not being with `get_`, and then use those methods to obtain values. Also, we can use the same procedure when we would like to display values that need to be set. This practice allows us to restrict which members can be modified in the Browser and emulate the `const` identifier in C++.

Results

We have experienced success in opening and browsing through a database that was written in Java. A large portion of the GUI has been implemented through the Java Swing classes and has successfully interacted with the databases that we have built. Through the Java reflection classes, we are able to create a modular application that dynamically loads the constants types it needs; once the browser is finished, it will require little maintenance when the constants database changes. After being in contact with Objectivity support, we have gained the ability to read from the databases written in C++, although this has not been implemented in the application yet.

Conclusion

After the Constants Database is re-structured to support the internal types that Objectivity provided us, the Constants Browser will be able to communicate with the database. When this primary goal is achieved, the development of the Constants Browser will be able to accomplish the rest of the features originally outline in its goals. When this is completed, the Constants Browser will be a reliable and easy to use tool for the CLEO III Constants Database.

Acknowledgments

I am pleased to acknowledge Dr. Martin Lohner, of the University of Florida, who proposed this Research Experience for Undergraduates project and who gave his time and energy to help me along the way. I would also like to thank Dr. Edith Cassel and Prof. David Cassel for introducing me to the program and taking their time to kindle my interests. This work was supported by the National Science Foundation REU grant PHY-9731882 and research grant PHY-9809799.

Footnotes and References

1. Objectivity for Java Guide. Objectivity, Inc. (1998)
2. http://java.sun.com/products/jdk/1.2/docs/
3. http://www.odmg.org/