JAVA EXPERT GUI FRAMEWORK FOR CERN BEAM INSTRUMENTATION SYSTEMS

S. Bart Pedersen, S. Bozyigit, S. Jackson, CERN, Geneva, Switzerland

Abstract

The CERN Beam Instrumentation Group's software section has recently performed a study of the tools used to produce Java expert GUI applications. This paper will present the analysis that was made to understand the requirements for generic components and the resulting tools including a collection of Java components that have been made available for a wider audience. The paper will also discuss the prospect of using Maven as the deployment tool with its implications for developers and users.

INTRODUCTION

The CERN Beam Instrumentation Group (BI) belongs to the Beams Department (BE). Inside this group, the work of the BI Software Section is to implement real-time servers in C++ that control instruments developed for beam diagnostics located on all CERN accelerators and their transfer lines (LHC, LHC injectors, ISOLDE, LEIR, AD…). The team is composed of around ten physicists, engineers and students with good software skills. The main GUI clients are the hardware specialists in charge of the instruments, along with a few operators and accelerator physicists who use them during special manipulations for additional status and control. Before doing any low level C++ implementation, a design of the instrument is made using a dedicated CERN software architecture called FESA (Front-End Software Architecture) [1] (Figure 1).

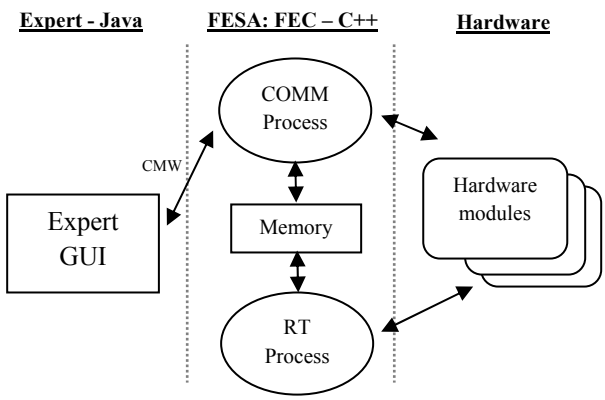

Figure 1: BE Front-End Software Architecture (FESA).

Providing graphical interfaces (GUI) implemented in Java is also part of the section's mandate. Hardware experts need to access their equipments in many different ways for parameter setting, signal visualization, error diagnostics, calibration, data post processing and so on.

Java (JDK1.6) is currently the BE department's development language for all GUIs and the software section's Java developments follow this standard. The low level software architecture, middleware and Java component libraries are provided by the BE Control Group.

PROBLEMATIC

Requests and specifications for new expert GUIs are often very different and depend on the type of instrument, the acquisition electronics and the type of diagnostics to be performed. Therefore, a standardization of a framework's functionalities and a static list of options is insufficient. The software engineer should instead be able to create ad-hoc Java classes and components (graphical or not) to fulfil specific requests.

Nevertheless, since the software requirements, although different, often have a similar overall structure, it made sense to have a common Java framework on top of which everyone could build specific applications. A functionality review of current expert GUIs revealed that a common customizable graphical framework is an efficient way to standardise and facilitate the development of a Java project implementation.

All of CERN's beam instrumentation systems belong to a particular accelerator domain (LHC, SPS, PS…) and are triggered by their own timing events. Each instrument is identified based on a database of devices that links it to a certain device name. The application window should therefore, for example, be able to display the incoming machine cycles and allow easy retrieval of device-names. Another important point is that the number of data types in an acquisition is limited. So, having the same graphical component to input or display a scalar or plot an array using common plot components would be a very efficient solution as it avoids code redundancy and strengthens software quality.

FROM THE OLD TO THE NEW FRAMEWORK

Why Did We Have To Change?

The main purpose of a common expert GUI framework is to provide a skeleton that can easily be built upon, so reducing the need to re-develop large sections of code for new applications and in so doing decreasing the total time spent to create such an expert application.

The old expert GUI framework, in place since 2004, was designed having the same goals in mind, but several issues have appeared over the years. People were therefore using this framework less and less and instead started developing their own applications from scratch. As already mentioned, an internal questionnaire revealed this was mostly due to the rigidity of the framework and the lack of desired functionality. Rigid, because this skeleton was providing a structure in the form of tabs and panels,

which was sufficient at the time when it was designed, but became a constraint as the panels to be shown always needed to be heavily customized. Several missing basic functionalities that were mandatory in terms of beam diagnostic tools also forced programmers to abandon the framework and instead implement their own code.

Besides these concerns, the section was frequently experiencing issues related the dependencies of their applications. All third party libraries coming from other CERN groups were directly obtained at runtime from external repositories and whenever a library lost its backwards compatibility, all the applications depending on it would break.

With the new expert GUI (v2), we tried to address these issues by providing a framework that still has all of the useful functionalities of the old framework, such as the communication and plotting packages, but is also programmatically more flexible (Figure 2). At the same time, we tried to manage the dependency issue by using a specific tool developed to handle this problem. Additionally, other useful components written by members of the section already existed and it was our intention to make these available to all users through the new environment.

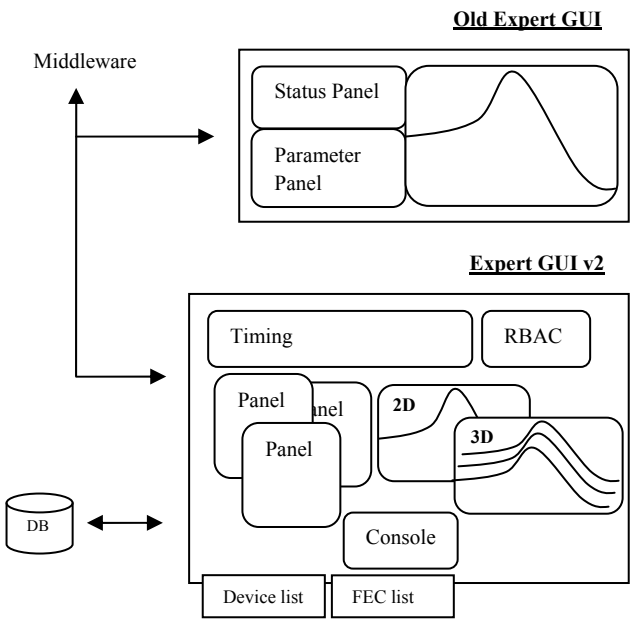

Figure 2: Old and new expert GUI.

What Did We Change?

The expert GUI framework needed a facelift both visually and programmatically. The main transformation was to convert the frame class that always had the same look with fixed tabs and panels to two different classes that inherit an abstract *ExpertGUIFrame*. First there is the *EmptyFrame* that basically provides an empty window where any of the desired components can be included. Second there is the *BasicFrame* which already contains commonly used components such as the RBAC (CERN Role Based Access Control) toolbar and a built-in console at the bottom of the frame.

In order to cover the need for some applications to look like the old expert GUI, a couple of panel classes were included in the new framework (*VintagePanel*, *VintageStylePanel*) that could be easily incorporated into the *BasicFrame*. If other predefined frame classes are desired in the future, they can be easily created with a framework extension.

In the old expert GUI, the very first step to create a program was to create a new main class that inherited properties from the framework main class. While this is still possible in expert GUI v2, by inheriting from the *EmptyFrame* or *BasicFrame* classes, the recommended way to obtain the framework functionality is to fetch a frame instance from a factory class and to use it as a delegate in your main class. This makes sure that the client who is using the provided frames only sees and uses the methods which were meant to be used by the class.

The communication library has basically stayed the same except that there is a new type of communication that deals with database (DB) connections. A new *CommunicationManager* class has been introduced so that the developer need not care about the details of instantiating different kinds of communications (FESA device or DB).

Applications often deal with more than one device, sometimes even with more than one instrument type. To facilitate the initialization of the communication and the handling of those devices, the factory class needs to provide a list of devices and databases. This information is used to create a menu item in the menu bar where all the relevant devices and databases are registered. Additionally, it feeds two manager classes that deal with the actions (button clicks) and the information that accompanies these devices.

Instead of managing the dependencies manually, as is done for applications based on the old expert GUI, we have tried to incorporate a third party tool called Maven, which automatically resolves all dependencies of a project and takes only the specified libraries from a local section repository.

This repository can also be used to share components between the members of the section simply by publishing their libraries into it.

What Are The Benefits?

The expert GUI v2 has become more flexible and it can more easily suit individual needs than its predecessor. For example, if someone wishes to completely write their own application from scratch, they can do so by instantiating the *EmptyFrame* but they wouldn't have to forego the communication library. Or, if someone wants to have an application that looks the same as the old expert GUI, it would also be possible to create it within the same framework. Finally, the new framework offers access to libraries (components) written by others, so everyone can profit from each other's work.

The usage of Maven should simplify the maintenance of the dependencies and generally facilitate the software build lifecycle.

JAVA SOFTWARE PROJECT MANAGEMENT WITH MAVEN

What is Maven?

In short, Maven is a software tool for project management and automation of builds [2], i.e. a build component testing and dependency management tool. Only a subset of the capabilities of Maven has so far been tested within the BE-BI Group.

Maven is not a graphical tool in its own right but a framework that is installed on the system (in an IDE i.e. Eclipse) which assists the development of a project. It is built around the concept of a "build lifecycle". This means that every complete project has to go through all the stages of the cycle which, for example, can contain compilation, testing, packaging (producing a jar), deploying in a local repository, deploying in a remote repository, etc.

The project has to successfully pass all of these stages. In a "traditional" project, these stages would need to be done separately. This manual mode forces the programmer to repeat the same procedure for each stage. Maven attempts to automate these procedures by extracting the pieces of configuration which are needed for every stage (for example: jar name, files to include in the jar, content of the manifest file, deployment location, etc..) and bundles them into a configuration file (Project Object Model) that is specific per project. The most important part of that file in our case is the list of direct dependencies.

Why Do We Need It?

We are not obliged to use Maven, and for us it started more as an experiment to see whether it could fit our needs. We were hoping to profit from the standardisation that comes with Maven as its capability for solving the dependency problem. Its concept of build lifecycle gives many structures almost for free, such as JUnit testing, packaging, JAR signing and generally structured project development. One additional strong argument to study Maven came from the fact that CERN Controls Group was also looking towards using Maven in the future.

Can We Use It?

Our deployment strategy does not implement any versioning of a given application with only a development and operational version at any given time. These two versions used to be deployed by an ANT script into two distinct folders. Each of these folders contains all the Java libraries (JAR) of all the applications, both development and operational versions. Either of these versions can be launched through a program called the *ApplicationLauncher* [3] (Figure 3).

A Maven project can be identified in a repository through three parameters: *GroupId*, *ArtifactId* and *Version*. These three qualifiers determine the folder structure in the local repository where the project is stored. However, if one decides to deploy using Maven, by default it forces you to use the same folder structure. It turns out that our current deployment strategy therefore does not fit well with this Maven approach, requiring some tricks to get rid of the Maven folder structure or significant modification to the *ApplicationLauncher*. In order to stay compatible with the current *ApplicationLauncher* it was therefore decided to opt for an Apache ANT script that copies the deployable JAR files to the desired location.

How Are We Using It?

Maven's dependency management system is declarative, transitive and simple to use. The libraries that a project needs to work properly are declared in its POM (Project XXX YYY) file. The three pre-cited qualifiers are enough for Maven to locate the right library in its repository system. Maven can provide an arbitrary number of repositories where it can obtain the requested libraries. It offers the possibility to specify a local repository which can be used internally in the section and not made public to other developers, allowing the possibility to control and manage changes i.e. updates of external JAR files.

The dependency system's transitivity (inheritance of properties and dependencies) facilitates the development of a project. Only direct dependencies need to be provided to Maven. If the libraries that a project relies on depend on another project, Maven will take care of this. It knows where to fetch the indirect dependencies, since all projects and libraries in the repository come along with a POM file that declares their dependencies.

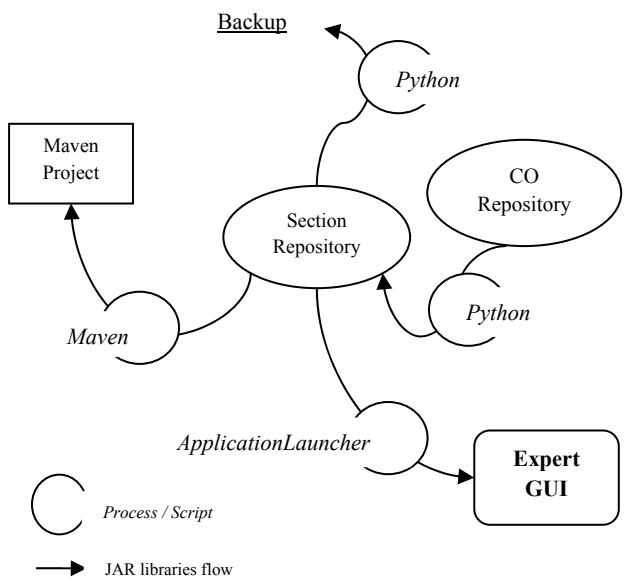

Figure 3: JAR libraries flow.

The inheritance concept, coming from object-oriented programming, is applied to the POM file. Through

inheritance, common settings and properties can be propagated from a "parent" project to projects inheriting from it. This mechanism gives the possibility to define a parent project that contains most of the dependencies and settings that are usually needed in any project. A creator of a new project therefore only needs a POM file to inherit from the parent POM file in order to have all necessary dependencies, to know the location of the keystore for signing JAR files and to have the settings in a manifest file, etc…

In the old expert GUI framework, one needed to check out a template project from the Apache Subversion SVN repository in order to start a new project. Maven also helps in this perspective because it provides a template through a mechanism known as "archetypes". An archetype gives the possibility to steer several properties of a project. It can impose a predefined folder structure on a new project providing opportunity to standardize structure across Java projects. It can also provide additional files such as ANT files or log files and can give a source code template for the entry point of a project.

What Are The Benefits?

It took a rather long time to set-up a generic environment to work with Maven and then to effectively use the tool for development. As it was not possible to deploy our JAR files the way we wanted, we were forced to opt for a hybrid solution using both Maven and ANT. The most comfortable way for us to develop an expert application is to have a simple ANT file that does the cleaning, the compilation, the packing and the final deployment with a simple mouse click as was the case for the old expert GUI. In order to achieve this and spare individual users from unnecessary Maven details, the Maven commands are encapsulated in such an ANT file, with all commands forwarded to the Maven system except for the deployment task which is done directly using ANT commands. To backup files and to retrieve specific JAR files, a Python script is used.

The main advantage of using Maven is the control of the dependency files. In addition the standardization of projects and the use of archetype templates greatly lower the effort required to start a new expert application project.

The incorporation and usage of Maven in our projects is an on-going process and there are still details to be tested. Most of the benefits could probably have been achieved in a different way using different tools. Many of these options, however, would have probably required an unacceptable amount of manual work and would have ended up in a patch work of different components needing to be unified to provide a similar functionality as Maven.

CONCLUSION

The analysis and improvement of our Java software development tools are still on-going. This process is taking time because of several constraints. Keeping backward compatibility with our old applications, sticking to CERN software standards, and covering the needs of all our programmers are just some examples of things to be considered.

The new expert GUI has already given very good results. Users can easily and quickly create a Java project with a pre-defined structure that will allow them to run an application in two mouse clicks. At the same time, they are able to add whatever components they need to libraries that are now common to all. These components have been specified inside our section and therefore already provide most of the functionalities that might be needed in such applications.

The use of Maven is not completed and has led to some integration problems for our Java software architecture. Nevertheless, the handling of the library dependencies and the archetypes are very useful. The CERN Controls group has not yet confirmed that Maven will be one of its standards, but from our experience so far, we will give positive feedback regarding these two interesting features.

REFERENCES

[1] Michel Arruat et al, "Front End Software Architecture [FESA]", http://accelconf.web.cern.ch/accelconf/ica07/PAPERS/WOPA04.PDF, ICALEPCS 2007.

[2] Maven: http://maven.apache.org/pom.html

[3] P. Karlsson, S. Jackson, "The introduction of hierarchical structure and application security to Java web start deployment", http://accelconf.web.cern.ch/AccelConf/ica05/proceedings/pdf/O4_008.pdf, ICALEPCS 2005.