BACKWARD COMPATIBILITY AS A KEY MEASURE FOR SMOOTH UPGRADES TO THE LHC CONTROL SYSTEM

V. Baggiolini, D. Csikos, P. Tarasenko, Z. Zaharieva, M. Arruat, R. Gorbosonov,
CERN, Geneva, Switzerland

Abstract

It is a big challenge to smoothly upgrade the control system of a large operational accelerator such as the LHC without causing unnecessary downtime. We have identified backward compatibility as a key measure to achieve this, because a backward compatible component can be easily upgraded. This document describes the work the CERN Accelerator Controls group does to provide methods and tools supporting backward compatibility.

BACKGROUND

Now that the LHC is operational, we from the controls group get requests from the operations team, which require a high degree of versatility. On one hand, the stability of the control system is necessary to ensure smooth operations while on the other hand, a certain amount of flexibility should be available in order to develop and deploy bug fixes and new functionality. To cope with these requirements, we are making a continuous investment in the quality assurance of our software, to improve the development process [1] and to provide new tools [2]. The work described in this document belongs into this context.

The CERN accelerator control system is highly complex, modular and distributed. The software part is structured as a three-tier system. GUIs at the top layer are written in Java and run on Linux consoles in the control room; the business layer in the middle is also written in Java but runs on powerful Linux servers in the computer center; the equipment control software at the lowest layer is written in C/C++ and runs on front-end computers with real-time-enhanced Linux and LynxOS. The Java part of the control system is composed of roughly 1000 Jar files, which are combined to around 400 different GUIs and 150 different server programs, which are deployed as over 600 processes on 400 machines. The C/C++ part on the front-end computers is represented by around 550 different device types (FESA [3] and legacy GM classes), deployed as over 70’000 device instances on 800 front-end computers. The development of all these components is done by more than 130 developers in 50 different teams. The development and collaboration is organized in a pragmatic and informal manner, with very low administrative overhead. Developers essentially collaborate along the dependencies of their software. If two components depend on each other, the respective developers will coordinate their work as needed. Even though there is no strong centralized organization to coordinate all upgrades done by the different teams, this form of collaboration is very efficient and agile. However, there are certain shortcomings as developers are not always aware of all the other components that depend on theirs, and consequently they may fail to fully coordinate their work with everyone. Therefore, an upgrade may result in down time to the LHC.

THE DEVELOPER’S VIEW

Let us examine the problem from the perspective of an experienced developer, who needs to modify a widely used library, which possibly requires changes in the signature of an API method.

Let us examine the steps this developer will follow. As a first step, he[*] will decide if he may modify the API method or not. He will try to understand if some other component uses that method, i.e., he will examine number and origin of *incoming dependencies*. There are three possible cases: there are no incoming dependencies, just a few of them, or many. In the first and the last cases, the situation is clear. If there are no incoming dependencies, then he can freely modify the method, because he has no backward compatibility constraints. If there are many incoming dependencies from many client components, then he cannot change the method signature; he must modify his library in some backward compatible way.

If there are just a few incoming dependencies from one or two client components, the developer can chose between two approaches: He can either accept backward compatibility constraints or he can break backward compatibility. The two solutions have opposite advantages and disadvantages. In the first case, staying backward compatible makes development more difficult but deployment easier. Backward compatibility constraints (keeping the old method signature) typically lead to sub-optimal solutions with more code to maintain. Also, the developer has to validate that the change is really backward compatible, which might be difficult. Deployment is easier because a backward compatible component can be deployed anywhere without breaking any dependent clients. Deployment can be done selectively, starting with those systems that really need the new functionality, and deferring upgrades of the other systems until the upgrade has been validated. There is no need for wide coordination or big-bang changes.

In the second case, advantages and disadvantages are inverted: breaking backward compatibility makes development easier but deployment more difficult. The developer has no backward compatibility constraints, and can choose the best solution, which generally leads to cleaner results and less maintenance. However, careful

[*] This article uses the masculine pronoun ‘he’ for brevity, but intends ‘he or she’

coordination is needed with other developers responsible for dependent components. They must change their source code to adapt to the new API and re-build and re-test their components. And then, new versions of all components must be deployed at the same time. There are two issues with that: firstly, all this can be difficult to organize, and secondly, the other developers will be unhappy if they need to adapt, rebuild, re-test and re-deploy their client components too often.

The above description admits that our developer has a clearly defined API and that his clients respect this API. Reality though might be different. The developer may not have clearly specified which classes and methods belong to the API, and the clients may disregard the official API and use additional (non-API) classes contained elsewhere in the software component. As a consequence, with an insufficiently specified and enforced API, our developer has to take the precautions above for each and every public method he might want to change.

TOOLS TO SUPPORT SMOOTH UPGRADES

The description above illustrates that upgrading a widely used component is a challenging development task that must be supported by good tools. We have identified four areas for which we want to provide tools: (1) dependency analysis to identify incoming dependencies, (2) backward compatibility validation to verify that API changes are really backward compatible, (3) versioning with rules to clearly inform the dependent clients if a modification is backward compatible, and (4) API consolidation to clearly specify classes and methods belonging to the API and to enforce their appropriate usage.

The following subsections discuss each of these areas, by first presenting existing approaches and tools and then motivating and explaining own developments we did.

Tools to Analyze Incoming Dependencies

Our developer wants to know about incoming dependencies right from inside his IDE. For example, he wants to right-click on a given method and execute a command “show incoming dependencies”. As a result, he expects to get a list of client libraries that use the selected method, with the possibility to navigate to the client source code from where the selected method is invoked.

Most IDEs provide very similar functionality, namely to show the call hierarchy of a method within a given project. However, this functionally only reveals incoming dependencies from within the source code and components (Jar files) present in the IDE. It does not show incoming dependencies from external Jar files. Also, it does not take into account previous versions of components that are deployed in operations, but only shows dependencies between the latest snapshot of the source files.

There exist stand-alone tools capable of analyzing dependencies between a large set of Jars, such as Tattletale [4] and JDepend [5]. They produce a lot of useful information, but they analyze dependencies only at the class level, not at the method or field level as we need.

Therefore, we decided to develop our own tool which is based on a client-server architecture. The server creates a list of all the roughly 1000 Jar files running in production and analyses the byte code of all classes they contain using the Apache Commons Byte Code Engineering Library BCEL [6]. It collects information about method and field-level accesses from one Jar file to another. The dependency information is stored in a database, and made accessible to the client over a remote RMI call. The client is an Eclipse plug-in with the right-click-on-method functionality described above. The analysis is carried out every 20 minutes and takes roughly a minute to run.

Tools for Assessing Backward Compatibility

Our developer also needs tools which help him ensure that modifications he made are really backward compatible. Ideally, backward compatibility is verified early on, during development, warning the developer as soon as he breaks backward compatibility. We first concentrated on tools to assess *binary* backward compatibility for Java. After all, the goal is to deploy a new version of a component (a Jar file) without even recompiling the client code. The Java Language Specification [7] contains clear rules to guarantee binary backward compatibility.

The “PDE API tools” [8] contained in the Eclipse IDE provide exactly this functionality. They assess backward compatibility of a project in the IDE by comparing it with a previous version of the same project. Once this functionality is activated and properly configured, the IDE gives immediate feedback and warnings about backward compatibility violations to the developer. Although this looks like a perfect fit to our needs, there are two issues to overcome. Firstly, PDE API tools (and Eclipse as a whole) use OSGi [9] to declare the public API packages. So far, we do not use OSGi, and introducing OSGi into a software development process is a big decision, which should not be driven only by the needs of one tool. Secondly, a considerable amount of manual configuration is required to properly configure the PDE API tools. We would have to automate this because we cannot expect our developers to do it manually. Therefore, we will need to weigh the benefit of using this tool against the overhead just described.

Of course, comparing the API signatures of different versions is only the first step for checking backward compatibility. It can be considered as a sort of early-warning system for the developer while he modifies his code. To validate backward compatibility further, we rely on other means, such as function tests and our Continuous Integration server. We also validate the core elements of the control system in the Controls Testbed [2], which carries out function and integration tests. In the future, we might also explore more formal approaches to augment the API specification, such as Design by Contract [10].

Versioning Schemes and Related Tools

Once our developer has finished the modifications to his component, he has to increase the version number. All the Java components in the accelerator control system are versioned using a scheme with three numbers separated by dots (*x.y.z*), which are called *major*, *minor* and *micro* (*major.minor.micro*). Semantic versioning [11] assigns a meaning to each of these numbers: if the developer did a (backward compatible) bugfix, he increases the *micro* number, if he added functionality but the overall change is still backward compatible, he increases the *minor* number, and if he breaks backward compatibility, he increases the *major* number. If the developer uses semantic versioning, the clients can simply infer the impact of the change from the change in version number. A typical client will automatically accept new versions of a library if only the minor or micro numbers have changed.

We want to start using semantic versioning, and back it up with tools that automatically calculate the new version based on the changes made to the code. We have identified two tools providing such functionality: PDE API tools and a semantic versioning plug-in for Maven [12].

Tools to Specifying and Enforce APIs

To fully define a Java API, our developer must be able to specify the packages that contain API classes. In addition, he may add further constraints, e.g. to indicate that clients are allowed to use a given public interface (invoke its methods) but not to extend it.

Once an API is specified, it must be enforced, e.g. clients must be prevented from using non-API classes. Standard Java does not provide sufficient mechanisms for this purpose. Therefore, PDE-API tools use OSGi to specify and enforce access to API packages, and Javadoc tags to specify further API constraints. For example, the @noimplement tag indicates that an interface can be used but should not be implemented in client code.

A completely different approach is based on static crosscutting functionality provided by AspectJ [13]. The 'declare warning' construct of AspectJ makes it possible to issue warnings for illegal method access, e.g. a client accessing non-API methods. These checks are executed at compile-time, and simply require the AspectJ compiler and weaver to be executed as part of the build process.

APPLYING THE SAME CONCEPTS TO C++ CODE AND FESA DEVICES

So far, this document has only discussed Java software. The other languages we use are C and C++. We started with Java because this is the area where we need to achieve smooth upgrades first, and because Java provides many mechanisms and tools for what we want to do. Once we have a clear idea about our needs and feedback from developers, we intend to provide similar tools for C/C++.

But already now, we try to use backward compatibility concepts for the low-level software that integrates the hardware devices into the accelerator control system. As for Java, we need to provide guidelines and tools enabling the device developers to achieve smooth upgrades.

All our devices follow the device/property model, which means that a device class has properties (e.g. a Magnet has a Current property, or a Motor has a Position property). The public API of a device class is represented by the properties it exposes to its clients.

Device developers modify the public API from time to time, to make bug-fixes or to introduce new functionality. As for the Java libraries, these modifications can be backward compatible or not. For instance, adding a new property to a device class is a backward compatible modification, whereas renaming an existing property breaks backward compatibility.

Currently the versioning of FESA devices is slightly different from versioning Java software. FESA device classes do have a version, but the device developer is expected to increase it only if his modifications break backward compatibility. He can keep the version unchanged if his modifications are backward compatible. Whenever the version is increased, client code accessing the devices must be re-configured or even re-released to use the new version of the FESA class. If the version is unchanged, the modified FESA device just replaces the old one operationally as soon as it is deployed.

With the current FESA tools, the developer is only *expected* to increase the version when he breaks backward compatibility, but not *forced* to do so. This has occasionally lead to problems in the past. Developers have made non-backward compatible modifications to their FESA device/property API without increasing the version number. After the deployment of these devices, some important client applications stopped working because they relied on the old API, and LHC operations were affected.

We are now improving the development process and tools for FESA development based on the same concepts as described above for Java software development. The following paragraphs shall explain this in more detail.

A FESA developer, who needs to make a modification to his device, carries out the same steps as the Java developer discussed earlier. He first analyzes the incoming dependencies to the device property he might want to change. He then decides whether to stay backward compatible or not. In case he remains backward compatible, he needs tools to validate backward compatibility, and so on.

Regarding the tools, we are developing means to assess incoming dependencies to the device properties of FESA classes. This work is largely based on the Controls Configuration Database (CCDB) [14], which contains the configuration data for all device classes, their properties and devices. It also contains configuration of the most important client applications that use the FESA devices, such as LSA and InCA [15]. Thus, the accelerators controls system is to a large extent data-driven and the challenge is to ensure that a coherent set of configuration data is used throughout the controls system. To achieve that, we have established a strategy with several steps. First of all we need to clearly specify the visibility of the configuration data. Previously all data – data necessary for the operation of the accelerators and data used by equipment experts for low-level device control – was exposed to everyone through public APIs. Now the approach has changed and the data for the FESA classes is divided into a public API used for operations and a private API used only by equipment experts. Another important aspect is that the CCDB needs to have feedback of who uses what from the public API, in order to ensure that only backward compatible changes are made. This feedback allows us to find out which device properties are used in operations, i.e., determine the *incoming dependencies* for a given device property.

However, not all clients of FESA devices propagate feedback to the CCDB regarding the device properties they use. We have important systems that use configuration files or other means to specify the device and properties they need, e.g. the Software Interlock System [16] or the Post Mortem Analysis system [17]. We are currently examining methods to dynamically harvest information about device property usage for all Java client applications. For this purpose, we have instrumented the JAPC communication library [18], which is used for nearly all device access from Java. JAPC dynamically tracks the device properties that were accessed and periodically uploads this information to the CCDB.

Validation of backward compatibility during development is done by the newest FESA development tools, which compare the previous version of the public device/property API of a FESA class with the one under development. If the FESA tool detects a change that is not backward compatible, it forces the developer to increase the version number according to the semantic versioning rules. This should prevent the problems described earlier in this section.

Finally - unlike Java - FESA does not need additional tools to specify and enforce the public APIs, because this is already covered very well by the existing FESA tools.

CONCLUSIONS

We need to upgrade the LHC controls system without causing any unnecessary down time. Backward compatible changes are a means of achieving this. The same concepts of backward compatibility apply to Java software and to FESA devices. In both cases, making backward compatible changes is challenging, and developers need to be supported by good tools. We have identified some promising approaches and tools, and we know how we want to apply them. We are evaluating the different alternatives and have started implementing our own tools where third party solutions are missing or insufficient. This is a non-negligible investment, but we are convinced that it will pay off, by making our work easier and the LHC operations more reliable.

REFERENCES

[1] K. Sigerud et al., “The Software Improvement Process – Tools and Rules to Encourage Quality”, Proceedings of ICALEPCS’11, Grenoble, France.
[2] N. Stapley et al., “An integration testing facility for the CERN accelerator controls system”, Proceedings of ICALEPCS’09, Kobe, Japan.
[3] M. Arruat et al., “Front-End Software Architecture”, Proceedings of ICALEPCS’07, Knoxville, Tennessee.
[4] Tattletale, http://www.jboss.org/tattletale
[5] JDepend, http://clarkware.com/software/JDepend.html
[6] Apache Commons Byte Code Engineering Library, http://commons.apache.org/bcel/
[7] J. Gosling et al., “The Java Language Specification, Third Edition", Addision Wesley 2005, http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf
[8] PDE API tools, the Eclipse Foundation, http://www.eclipse.org/pde/pde-api-tools/
[9] OSGi™, the Dynamic Module System for Java, http://www.osgi.org/
[10] Design by Contract, http://en.wikipedia.org/wiki/Design_by_contract
[11] Semantic versioning, http://semver.org
[12] Semantic versioning plugin for Maven https://github.com/jeluard/semantic-versioning
[13] R. Laddad, “AspectJ in Action”, Manning 2010. http://www.manning.com/laddad/
[14] Z. Zaharieva et al., “Database Foundation for the Configuration Management of the CERN Accelerator Controls Systems”, ICALEPCS 2011, Grenoble, France.
[15] S. Deghaye et al., “CERN Proton Synchrotron Complex High-Level Controls Renovation”, Proceedings of ICALEPCS’09, Kobe, Japan.
[16] J. Wozniak et al., “Software Interlock System”, Proceedings of ICALEPCS’07, Knoxville, USA.
[17] M. Zerlauth et al., “The LHC Postmortem Analysis Framework”, Proceedings of ICALEPCS’09, Kobe, Japan.
[18] V. Baggiolini et al., “JAPC - the Java API for Parameter Control”, ICALEPCS’05, Geneva, Switzerland.