ПЛОВДИВСКИ УНИВЕРСИТЕТ “ПАИСИЙ ХИЛЕНДАРСКИ”, БЪЛГАРИЯ
НАУЧНИ ТРУДОВЕ, ТОМ 36, КН. 3, 2009 – МАТЕМАТИКА
PLOVDIV UNIVERSITY “PAISSII HILENDARSKI”, BULGARIA
SCIENTIFIC WORKS, VOL. 36, BOOK 3, 2009 – MATHEMATICS

DESIGN APPROACHES TO WRAPPING NATIVE LEGACY CODES

Anna Malinova

Abstract. This paper describes some of the results of applying different design approaches to wrapping legacy scientific codes in the domain of plasma physics and simulation of metal vapor lasers. The process of wrapping includes creating Java wrappers through the use of Sun's Java Native Interface and application of design patterns, such as Adapter, Proxy, and Wrapper Façade. A real use case connected with wrapping legacy plasma simulation codes is also presented.

Key words: design patterns, native code, legacy software, wrapping, JNI

1. Introduction

Existing mathematics and physics software libraries are written in different programming languages, which forces developers to generate glue code in order to avoid rewriting from scratch or not using them at all. This paper makes an attempt to connect some well known design patterns to the process of Java wrapping of native legacy scientific codes. Native software denotes code that is ”implemented in platform-dependent code, typically written in another programming language such as C, C++, FORTRAN, or assembly language”, as stated in the Java Language Specification [5]. Native applications in the domain of physics simulations are often considered legacy code, because they are developed with technologies that precede in time the introduction of grid computing, web services and other recent computing approaches and best practices. Our aim is to link these legacy modules with Java-based software environment for numerical simulations and to convert some of the wrapped codes into web services [2, 9, 10 and 11]. Next these web services may be chained into a BPEL process for simulation, as discussed in [12]. The main purpose of this research is to reuse and integrate codes that have been developed in our group (e.g. [4], [6]), as well as codes developed by other scientific groups.

In this paper the design patterns are discussed in the context of invoking native applications from Java. Two integration options when linking to programs in other languages have been investigated in [10] and [11]:

– Invoking the program at operating system level. In this case the java.lang.Process and java.lang.Runtime classes are used to invoke a random program, pass in arguments via the standard input and read results via the standard output.
– Using native methods. In this case the Sun's Java Native Interface (JNI) is used to link the Java code to the native code through Java methods declared as native. Java applications call native methods in the same way they call methods implemented in the Java programming language. Behind the scenes, native methods are implanted in another language and reside in native libraries.

The first method has many limitations and is appropriate when the interaction requirements are relatively simple. In contrast, JNI provides rich interface between native and Java code and is suitable for fine grained interactions. The latter approach is considered in the rest of the paper.

2. Design patterns applied

In the context of creating JNI wrappers, a Java wrapper calls a legacy API through the created JNI glue code, as it is shown Figure 1.

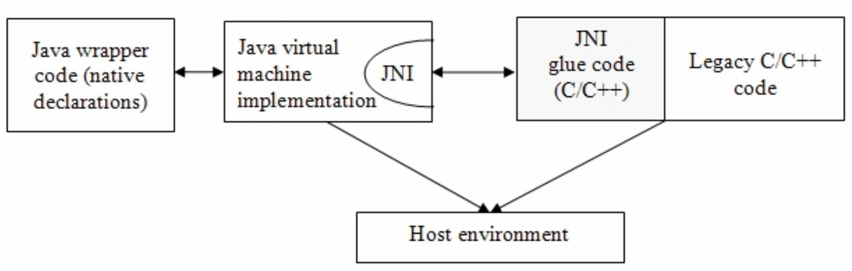
Java wrapper code (native declarations)
Java virtual machine implementation
JNI
JNI glue code (C/C++)
Legacy C/C++ code
Host environment

Figure 1. Interaction between legacy application and its Java wrapper

Figure 1 does not show whether a single wrapper should encapsulate an entire legacy application, or whether multiple wrappers should wrap an application's services individually.

There are two integration options when encapsulating a native legacy application, as discussed in [1]. There could be either one wrapper for the entire application (Figure 2) or several of them, one per a needed functionality (Figure 3). These wrappers can be used to form a library of wrapper classes, corresponding with different native classes or base services.

A single wrapper encapsulating the legacy application's functionality needed by the Java application has the form depicted by the UML class diagram in Figure 2. In this model one wrapper class contains all the methods that invoke the legacy API. When a client (SomeClass) requires something form the legacy application, it sends a message to the wrapper (SingleWrapper), which in turn calls native API through already created JNI stub functions.

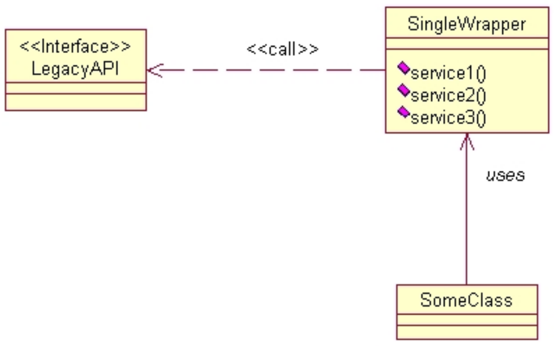

Figure 2. Single class wraps the entire application

Two important characteristics can be noted concerning the model shown in Figure 2:

- This structure does not require wrapping all the services of the legacy application. It only requires one wrapper to provide methods for all the native services that are wrapped;
- The wrapper class does not implement the native API. Instead, the wrapper calls the native methods, as indicated by the stereotype on the dependency relationship in Figure 2. The legacy application implements the API.

The alternative to a single wrapper is shown in Figure 3. There are shown a number of classes each encapsulating different functionality provided by the legacy application.

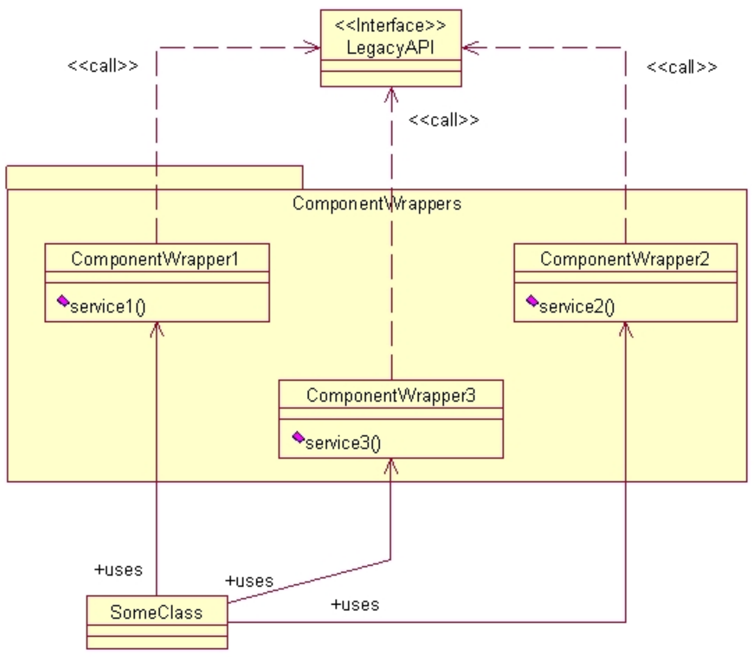

Figure 3. Create a library of wrapper classes

Both approaches allow replacement of legacy services when the new application no longer needs a native legacy application to provide a service. In this case, if a new implementation of this functionality is provided, one method invocation would simply be replaced with another inside the wrapper class.

In the process of creating Java wrappers of native legacy applications, some well-known object-oriented techniques can be applied, such as design patterns. Specifically the patterns Adapter and Proxy [3] are considered. Both are not directly applicable in this situation because here we have client and adapted code written in different languages. In addition it is often a question of adapting non-object-oriented legacy software. Bellow the application of the Wrapper Façade design pattern when wrapping non-object-oriented native code is also described. [17].

The most straightforward way to create wrapper classes through JNI is one-to-one mapping [7, 8]. This approach requires us to write one stub function for each native function we want to wrap. Hence, each Java method declared as native maps to a single native stub function, which in turn maps to a single native method definition. The stub serves two purposes:

– to adapt the native function's argument passing convention to what is expected by the Java virtual machine;
– to convert between Java programming language types and native types.

Adapter design pattern relates to creating JNI stub functions in the sense that it converts the interface of a class into another interface the client expects [3]. Thus Adapter makes it possible for classes to work together where that wouldn't be possible because of incompatible interfaces. This is exactly the reason for creating a JNI adapter – without it the native class and its Java wrapper would not be able to work together.

The class version of Adapter uses multiple inheritance to adapt the Adaptee's interface to Target's interface (see Figure 4), and therefore it cannot be applied with wrapping through JNI since in this case the inheritance is impossible.

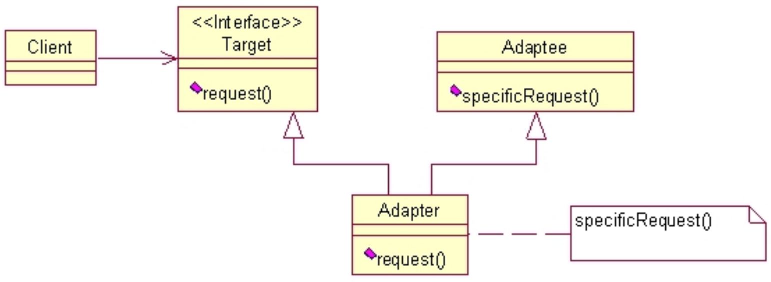

Figure 4. Adapter design pattern – class version

The object version of Adapter realizes the adaptation by using object composition, instead of inheritance. Thus an object adapter lets a single Adapter work with many Adaptees – the Adaptee itself and all of its subclasses (if any).

The object adapter relates to the process of JNI wrapping of native legacy codes – the client sends request to the Java wrapper class; then the JNI stub functions, implementing the Java methods declared as native, make the corresponding invocations of the underlying native functions. As a result, the native interface is adapted to Java interface.

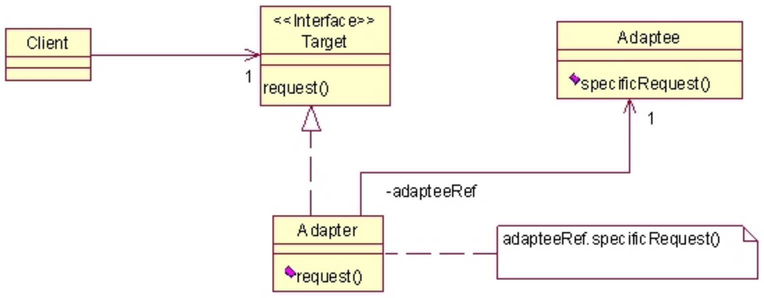

Figure 5. Adapter design pattern – object version

One-to-one mapping addresses the problem of wrapping native functions. These can be standalone C/C++ functions that return result or modify parameters passed into the function, or they can be C++ class member functions. However, if an instance of a C++ class is created in a JNI stub function, another problem arises: how can C++ classes be used by a Java program and keep objects around while the program is running. One way to handle this situation is to define a Java class called "peer class" that corresponds to the C++ class [7, 8]. Peer classes directly correspond to native data structures. Each instance of the peer class corresponds to a C++ object, tracking the state of the object. The Proxy design pattern can be considered when creating Java peer classes that wrap native structures, as it is shown in Figure 6.

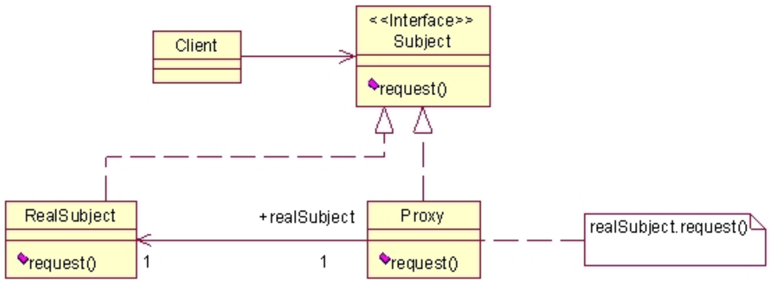

Figure 6. Proxy design pattern

In [3] the Proxy is defined as surrogate or placeholder for another object to control the access to it. Thus the Proxy pattern makes the client of an object to communicate with a representative of this object rather then the object itself. Such a representative can serve many purposes determined by its pre- and postprocessing of requests. Several versions of Proxy can be distinguished: Remote Proxy, Virtual Proxy, Cache Proxy, Synchronization Proxy, Protection Proxy, Counting Proxy, Firewall Proxy. However common characteristics for Proxy classes are the following [3]:

- maintains a reference that lets the proxy access the real subject;
- provides an interface identical to Subject's so that a proxy can be substituted for the real subject;
- controls access to the real subject and may be responsible for creating and deleting it.

The design patterns Adapter and Proxy are related. Adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its object. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be subset of the subject's. Java peer classes that wrap native data structures apply both Adapter and Proxy design patterns. On one hand the existing C++ interface is adapted to Java interface, and from the other the peer class serves as a proxy to the native C++ class it represents, taking care of creating and deleting the instances of this class, and providing interface that is identical to or a subset of the wrapped one.

Integration with non-object-oriented code, written in such languages as C and Fortran, is discussed in [13]. The object-oriented re-architecturing technique presented there implies to use object-oriented architecture (wrapper) around internal elements that are not object-oriented. Examples of object-oriented encapsulation, hiding the underlying routines are provided, i.e. the NAG software from the Numerical Algorithms Group [14]. Although written in C, the NAG C library's functionality can be accessed from other higher-level languages.

The Wrapper Façade design pattern, presented in [17], encapsulates the functions and data provided by the non-object-oriented legacy native API within more concise, portable and maintainable object-oriented class interfaces, as it is shown in Figure 7. An example of occurrence of Wrapper Façade pattern can be found in the context of Internet communications, as discussed in [16]. For efficiency or legacy reasons, many protocols of the TCP stack are implemented in C. And even though Java, trough the JNI, allows a programmer to directly invoke C functions, one or more classes are introduced in order to separate the protocol from the client application.

In Figure 7 the application code invokes a method on an instance of the Wrapper Façade. The Wrapper Façade forwards the request and its parameters to one or more of the lower-level native API functions that it encapsulates, passing along any internal data needed by the underlying functions.

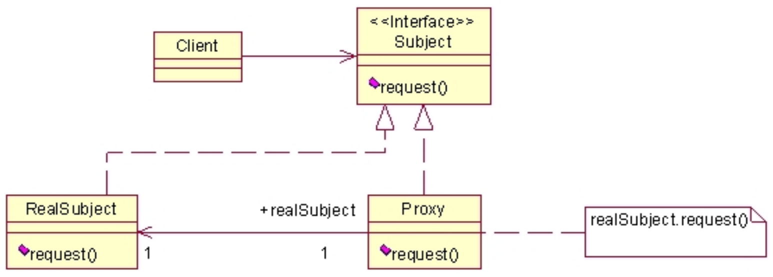

Figure 7. Wrapper Façade design pattern

Concerning the Java wrapping the Wrapper Façade pattern corresponds to creating a C++ class that invokes the non-object-oriented code. That class is then wrapped through JNI. This additional C++ class (classes) provides higher-level object-oriented interface that is easier to maintain and reuse. he alternative to Wrapper Façade is to create a Java class that directly accesses the non-object code through the JNI stubs. This implies that all the methods, declared as native inside it, are also static. However, Wrapper Façade can simplify the wrapping if there is a large amount of native functions to be wrapped.

3. Real use case

In this section are presented recent results of creating Java front-ends for some basic functionality of the Plasimo simulation software [15]. The Plasimo code is multi-physics code for simulating a variety of plasma sources with various degrees of equilibrium, electromagnetic field configurations, flow regimes and geometries [2]. Plasimo is a framework written in C++ and the application of different wrapping techniques was investigated. The Java Native Interface was used to produce a class library that wraps a set of Plasimo's functions and classes. The aim of this wrapping was to give the legacy code access to the new web technologies and best practices. For details refer to [9] and [2].

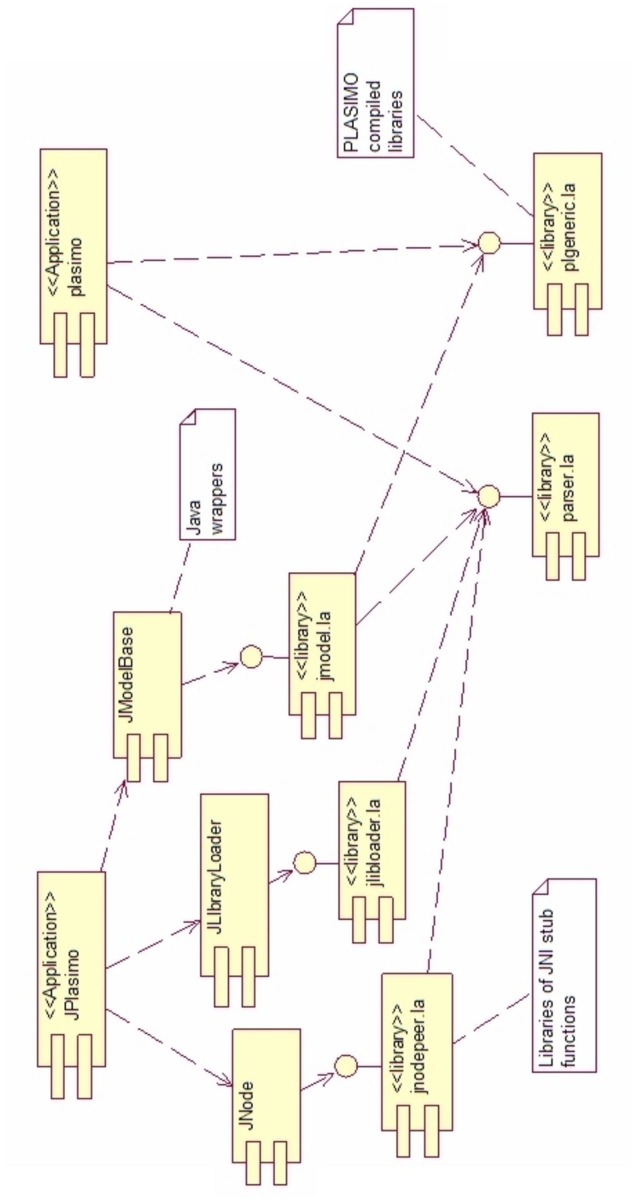

Figure 8. UML component diagram showing the dependency relations between different Plasimo components after the Java wrapping

The techniques ”one-to-one mapping” and creating Java peer classes, related to the Adapter and Proxy design patterns, were applied. The Wrapper Façade pattern was not used because of the small amount of non-object-oriented code that was wrapped. As discussed in the previous section, direct access form the Java class to the non-object code was chosen instead.

In Figure 8 are presented the dependency relationships between different Plasimo components after the Java front-end was created. The application JPlasimo is test application that invokes the native methods of the created Java wrapper classes. The implementation of the native methods is provided by a set of libraries consisting of JNI stub functions, which in turn invoke the Plasimo functions inside the Plasimo compiled libraries.

4. Conclusion

A significant amount of high-performance scientific simulation software in the domain of plasma physics, created during the last two decades, is written in native languages, such as C, C++ and FORTRAN. Since it is highly desirable to be able to reuse these large and complicated software packages, such techniques as wrapping native legacy codes, is still a question of present interest. Considering the object-oriented re-architecturing of non-object-oriented code and design patterns can help with further understanding of the wrapping process and better structuring of the wrapper code.

5. Acknowledgements

This work is partially supported by the NSF of the Bulgarian Ministry of Education and Science, Project VU-MI-205/2006 and IS-M-4 project of the University of Plovdiv ”Paisii Hilendarski”, Bulgaria.

References

[1] Asman P., Legacy Wrapping, http://www.hillside.net/plop/plop2k/proceedings/Asman/Asman.pdf

[2] Dijk J., A. Malinova, V. Yordanov, J. van der Mulen, New Interfaces for the Plasimo Framework, Proceedings of the the 6th International Conference on Atomic and Molecular Data and Their Applications (ICAMDATA), Beijing, China, October 28-31, 2008 (to appear).

[3] Gamma E., R. Helm, R. Johnson, J. Visslides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[4] Gocheva-Ilieva S. G., I. P. Iliev, Mathematical modeling of the electric field in copper bromide laser, Proceedings of Int. Conf. of Numerical Analysis and Applied Mathematics, ICNAAM 2007, September 16-20, 2007, Conference Proceedings of American Institute of Physics (AIP), vol. CP936, pp. 527-530, 2007. doi:10.1063/1.2790197, http://adsabs.harvard.edu/abs/2007AIPC..936..527G

[5] Gosling J., B. Joy, G. Steele, G. Bracha, Java Language Specification, 3rd ed., http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[6] Iliev I.P., S. G. Gocheva-Ilieva, On the application of the multidimensional statistical techniques for exploring copper bromide vapor laser, CP1067, Applications of Mathematics in Engineering and Economics '34-AMEE '08, American Institute of Physics, 2008, 475-482.

[7] Java Native Interface 5.0 Specification, http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html

[8] Liang S., The Java Native Interface: Programmers Guide and Specification, Addison-Wesley, 1999.

[9] Malinova A., V. Yordanov, J. van Dijk, Leveraging existing plasma simulation codes, International Book Series "Information Science & Computing", Number 5, pp.136-142, Supplement to the International Journal "Information Technologies & Knowledge", Volume 2/2008.

[10] Malinova A. A., S. G. Gocheva-Ilieva, I. P., Iliev, Wrapping legacy codes for Numerical simulation applications, Proceedings of the III International Bulgarian-Turkish Conference Computer science, Istanbul, Turkey, October 12-15, 2006, Part II, pp. 202-207, 2007.

[11] Malinova A. A., S. G. Gocheva-Ilieva, I. P. Iliev, Web Services-based simulation of metal vapour lasers, Proceedings of ILLA '2006 - IX International Conference on Laser and Laser-information Technologies: Fundamental Problems and Applications and LTL '2006 - V Intern. Symp. Laser Technologies and Lasers, Smolyan, Bulgaria, October 4-7, 2006, pp. 315-323, April 2007.

[12] Malinova A. A., S. G. Gocheva-Ilieva, Application of the Business Process Execution Language for building scientific processes for simulation of metal vapor lasers, Proceedings of the 3rd Balkan Conference in Informatics, Sofia, Bulgaria, 27-29 September, 2007, Volume 2, pp.75-86, 2007.

[13] Meyer, Bertrand, Object Oriented Software Construction, 2nd ed., Prentice Hall, New York, 1997.

[14] NAG libraries, Numerical Algorithms Group, http://www.nag.co.uk

[15] Plasimo simulation software, http://plasimo.phys.tue.nl

[16] Sevinc P., Flatin J., Guerraoui R., Patterns in SNMP-Based Network Management, Proc. 17th International Conference on Software and Systems Engineering and their Applications (ICSSEA 2004), vol.1 à 3, pp. 1.1-1.12, Paris, France, November 2004.

[17] Schmidt D., M. Stall, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture — Patterns for concurrent and networked objects, Volume 2, Willey, 2000.

Anna Malinova
Faculty of Mathematics and Informatics
University of Plovdiv
236 Bulgaria Blvd.,
4003 Plovdiv, Bulgaria
e-mail: `malinova@uni-plovdiv.bg`

Received 26 November 2008

ПОДХОДИ ЗА ПРОЕКТИРАНЕ НА ОББИВКИ НА НАТИВНИ НАСЛЕДЕНИ КОДОВЕ

Анна Малинова

Резюме. В статията са описани част от резултатите, получени след прилагането на различни подходи за обвиване на наследен научен софтуер в областта на физиката на плазмата и симулацията на лазери с метални пари. Процесът на обвиване включва създаване на Java-обвивки чрез интерфейса Java Native Interface и прилагането на такива образци за проектиране като Adapter, Proxy и Wrapper Façade. Представено е и реално приложение на използваните техники при обвиване на наследен код за симулация на нискотемпературна плазма.