ROMAI J., **3**, 2(2007), 229–241

FRAMEWORK-ORIENTED PROGRAMMING

Mocanu Mihai Ştefan

University of Piteşti

mocanushtefan@yahoo.com

1. INTRODUCTION

The notion of framework represents a piece of software that solves a particular problem. The text editors and command lines are less and less employed in writing software. Instead, large pieces of code are assembled together, resulting more complex and feature-rich applications. But together with a complex architecture come problems. Firstly, the learning curve is high. Knowing the properties of lots of commonly used frameworks is a difficult process. Secondly, the developer sometime needs to replace an existing piece of software with an updated version, or, more complex, replacing it with a similar one.

A framework and its properties can be best described using design by contract, so the paper uses this concept. We see how it can be extended to be applicable not only to an operation, but to a larger construct, and also how can we express known concepts using design-by-contract specific terms [6], [7], [8].

In order to better illustrate the notions, we take as example the development of a web application. In order to develop a web application, first we need to decide on a technologies use. For didactic purposes, from the multitude of available technologies and libraries, we will take a particular combination of those that best describe the notions in this article.

- For the web technology, we choose a servlet implementation. The number of components that offer a servlet technology is impressive, and we can choose our favorite one from several tens of implementations [3].
- The choice of the web presentation technology, is also varied. New and classic technologies form a range of 10-20 choices.
- Our application makes extensive use of reports, so we need a report generator. Although this is a fairly used feature, we only have around 5 choices.
- We need to export generated reports to a spreadsheet application like Microsoft Excel, so we need such libraries. In this respect, the number of choices is fairly limited (only 2-3 libraries for this task).

The order in which the software components are defined is important. We can see a classification: the first 2 technologies described are very general, while the later ones are more suited for particular problems. Also, the more general ones tend to fall in the "framework" category.

As noted, for the first technologies, we have a larger number of available implementation to choose from, while for more particular problems, this number is reduced. In addition, more particular technologies tend to have a more simple usage.

One big problem arising today is the interoperability of these components. This article attempts to make the following operations easier:

- replacing a software component with a later version;
- replacing a software component with a similar one.

The notion of "replace" has to be understand as replacing without any modification in the existing source code. We also want to define an useful algorithm for determining the circumstances in which we can replace the components.

In this article, the focus will be on object-oriented programming languages, and more specifically on Java, but some principles may be extended to other platforms too.

2. BASIC DEFINITIONS

Definition 2.1. *A **library** is a collection of classes, grouped together to perform a prespecified contract.*

It is a straightforward definition. Moreover, everything that can be packaged in a way usable by another software component can be regarded as a library.

However, just putting together a collection of classes and packaging them only offers an advantage regarding organization of the classes, offering a clue that they are related. There is little information on what that particular collection of classes does. This information is offered separately, and it is offered in various sources. Servlet technology, for example, is widely used and thus lots of documentation exist: books, tutorials, examples etc. This is also because it is a more general-purpose technology. But specialized libraries, like the ones used to export data in particular formats we need in this application, have less such information, although the functionality is more precisely defined and less voluminous.

Let us try to define some properties of a library.

Firstly, a library is a standalone entity and it performs that contract. That is the purpose of the packaging, that several smaller entities to be regarded as a whole, working together to perform a specific task.

Some libraries are specially built to implement a set of specifications. But as noted before, the description of the functionality is separated from the library itself.

Remark that any class has a implicit contract, even if it is not declared: it is what the class actually does. The contract of a library is defined by the union of the contracts of all the contained classes. However, as libraries tend to have large numbers of classes, this implicit contract is unusable with respect

to automated analysis. The purpose of this article is to make the contract of a library suitable to be used by other software components.

A library as defined above lacks the ability to describe what it is doing. We are interested of adding this capability to a library, while still preserving the entity structure.

Definition 2.2. *A **module** is a library that publicly describes its contract.*

Essentially, a module encapsulates a library and the description of its contract; the description of the contract needs to be placed inside the library. We are interested in exposing the contract in such way that it can be processable by another software component, and this is the advantage we are looking when defining the module in this way. We want to be able to better integrate software components, and, by defining in a standard way what a piece of software does, this represents a step forward.

Let us analyze the contracts of our test libraries.

The libraries for spreadsheet exporting have a fairly simple contract. As a minimum, the contract of these libraries can be expressed as the contract of a static method: theoretically, we could extract the functionality of the library and encapsulate it into one class. The reporting library is more complicated. The contract of this library can not be expressed as simple as the previous libraries, although basically it can be regarded as being a library that takes input data and exports them to various formats. Also, its functionality depends a lot on the libraries it uses. As for the servlet container library, it really is a very complex one. Its functionality and specifications, as well as the standards it is based on have evolved over time. To be able to even start it requires knowledge of different technologies; the contract of this library is accordingly complex.

The following definition proposes a way to describe the contract of a module.

Definition 2.3. ***The Public Application Programming Interface (Public API)*** *of a library is the minimal subset of classes that is used by the contractor in order to execute its contract.*

For the spreadsheet libraries, the Public API could consist of only one class responsible of transforming the input data into a spreadsheet. For the reporting library, the public API gets more complex, as it needs lots of classes to perform its contract: specialized types of data sources, different parameters, various types of report objects etc. As for the servlet container, the functionality it exposes is very large. It has hundreds of classes, for various purposes: request-response handling, session management, support for different technologies etc. However, compared to the actual number of classes in the servlet implementation, the Public API contains only a small percent of the classes.

By exposing a Public API, a module actually marks a set of classes that are necessary to perform the contract and are not susceptible to modify over time. This offers a good advantage when it comes to new releases, because the classes outside the Public API can be modified without affecting the contract of the module. In a new release, a module can modify some or many of the private classes, add new ones or remove old ones, but as long as the Public API remains the same, it can be safely used by an existing application.

Definition 2.4. *The* ***functionality contract*** *of a module is the union of all contracts of the classes in the Public API.*

In our examples, we can see that for simple modules, like the spreadsheet libraries, the module contract is exactly the contract of the publicly exposed classes, that is, the module contract is the same as the functionality contract. For more complex modules, like the servlet container, much more of the module contract is contained in classes other than the ones in the Public API. For instance, we can define our own Servlet (by extending the Servlet class). The

Servlet class is in the Public API of the servlet module, but there are other classes that are responsible of processing it, in order to comply with the servlet specification. Actually, this is a common pattern for a more complex type of module:

Definition 2.5. *A **framework** is a module for which the contract is not included in the functionality contract.*

In other words, the contract of the framework is much larger than what is exposed in the Public API; however, we only need the Public API in order to fully make use of the framework.

For instance, the servlet technology defines the Servlet session in the Public API, but the actual class that processes the session is not. Instead, an internal set of classes is responsible for defining the algorithms to process classes from the Public API. Note that these clasees can change over time; however, the only thing a user of the framework needs to know are the classes from the Public API. This is also true for different Servlet technology implementations: the Servlet session is accessed in the same way disregarding the library we are using. In other words, it may be implemented differently, but the usage is the same over all implementations.

Definition 2.6. *Let PA(A) be the Public API of module A, let PA(B) be the Public API of module B. We say that PA(B) **extends** PA(A) if* $\forall X \in PA(A),\ \exists\, Y \in PA(B)$ *such that* $(X = Y) \vee (Y \text{ extends } X)$

Using this definition, we can now give an answer to the two questions in the beginning of the article regarding replaceability:

Theorem 2.1. *(Simple Replaceability) We can replace a module A by module B if PA(B) extends PA(A).*

Proof. Any software component that uses module A uses it through the Public API, by instantiating and running the contracts of classes from PA(A). When we replace module A by module B, we are also using it through its Public API.

But for any class X ∈ PA(A) we can find a class Y ∈ PA(B) to substitute it. If Y = X, the substitution is obvious. Also, if Y extends X, by the fundamental properties of object-oriented programming, we can use Y in any operations that use X. ∎

Note that this definition does not make any statement regarding the contract: we can replace module A by module B, but we have no guarantee that module B is able to fulfill the A's contract. In general, we should replace module A by module B if B can fulfill the A's contract and when the B's public API extends A's the public API.

This theorem defines the easiest way a module can bring new functionality: based on the inheritance of the classes from the Public API. But this is not the only possibility. For instance, a new version can just add a new method to a class without extending the old class. In the particular case of Java, there is a strict specification when a set of classes can be replaced by another one: binary compatibility, defined in the "Java language specification second edition" [1].

This tells us that there is already a way to determine if we can replace a library by a new one, by means of testing binary compatibility. When we want to replace a module by a similar one, the binary compatibility test fails for the module as a whole. However, if the Public API passes the binary compatibility test, the module is replaceable, thus broadening the area of modules that can replace an existing one. If classes outside the public API have been refactored without breaking binary compatibility, it is still possible to replace a module by another one because we only use the classes from the Public API to run the contract with the module. This is the informal proof for the following theorem of binary replaceability.

Theorem 2.2. *We can replace a Java module A by a Java module B if PA(B) is binary compatible with PA(A), as defined by the "Java language specification second edition"*

So far, we could only replace a Java module A by another Java module B only if B was binary compatible with module A as a whole. Now we identify a series of steps for a framework:

- **initialization** is the step where the framework is set up, preparing it for the execution of contract;
- **contract** is the step where the contract is taking place;
- **dispose.** A framework should dispose any resources used.

An important part in the lifecycle of a framework is represented by the configuration.

Definition 2.7. *The **configuration** of a framework is the contract of the initialization step.*

The lifecycle of a framework is the succession of stages a framework goes through. Let us define it based on the steps defined previously.

Definition 2.8. *The **lifecycle** of a framework is the succession of initialization and contract steps in an indefinite number, followed by the dispose.*

The lifecycle is actually the common usage pattern of any module. In order for a module to function, it needs a valid context - the configuration. After a valid context has been defined, we need to make use of the contract of the module. In some cases, the module needs to be reinitialized with another context, and so on. After the module has finished its contract, it can be disposed by the user.

3. SUGGESTIONS

Based on the basic definitions in the previous sections, several suggestions for improving the software process are made in the following.

Firstly, regarding the public exposal of the contract, this has to be done in such way that it should not interfere much with the development process and should not take a considerable amount of time. Also, it has to be easy understandable by the software developer.

Suggestion 1. *A module can define its Public API inside a prespecified file (called manifest) using standard declarations.*

For a simple module, a simple solution can be developed. Let us take the example of the spreadsheet exporter. These libraries come packaged in a standard Java Archive file (JAR). The JAR standard provides a manifest file, which can contain standard information usable by the Java Virtual Machine (JVM) [2]. The idea is to extend the format of this file in order to include the declaration of the classes from the Public API. Besides usual declarations, like *Main-Class* or *Manifest-Version*, a new declaration can be introduced: *Public-API*, followed by the fully qualified class name. This change can be used by the JVM or by other tools to determine the properties of the module. Considering that simple modules have their contract the same as the Public API contract, we can state that the contract defined by Public-API classes is the same as the framework contract.

There are several advantages of using the Public-API declaration inside the manifest file.

Firstly, an automated tool could determine the module which can be replaced by another, using the replaceability theorems, Theorem 2.1 and Theorem 2.2. By analyzing the classes in the Public API, one can determine the changes in the Public API and determine if the module can be replaced by a new version. For instance, if one class in the new Public API extends one in the old one, the old module can be safely replaced. But if one class or one method is removed, then the new module can not safely replace the old one.

Of course, an automated tool can be used today to check all the classes inside a library to detect any changes and to determine if it can be replaced by a new version. But this would be highly inefficient, because some classes

can be modified or removed, without affecting the overall functionality. This is one of the purposes of the Public API: to describe the essential classes in a module.

Also, by declaring the Public API inside the manifest file, we can give the software developer a better entry point inside the module, by allowing him/her to concentrate only on a few classes. Java classes usually contain API documentation, that can be used by various tools (such as *javadoc* [4]) to generate documentation in various formats. For a library that contains hundreads of classes, just by looking at the API documentation can not give too much information. But by selecting from that number of classes only the ones in the Public API, we can have a better insight in the module.

One idea widely used today is the separation of the Public API from the actual implementation. For instance, in the case of the JavaServer Faces (JSF) technology ([5]), there is a set of classes (JSF-API) (publicly downloadable) that represent the core structure of the framework. The interfaces and the abstract classes must be implemented, resulting a new library that represents the implementation of the standards described in the JSF API. Also, when an implementation of the standard is used, it can not be used separately, but only together with the JSF API library it has been compiled against.

The solution to this problem is to declare the Public API in the implementation library in the same way we declare it for a simple module, even though the classes are not present in the implementation, but in the library that defines the standard.

In order for a framework to function, it needs a configuration. Traditional methods include configuration files, but as we move towards accessing a framework in a standard way, we also need to provide a standard way to specify the configuration of a framework.

Suggestion 2. *The configuration of a framework should be contained in the Public API.*

Classical approaches separate the configuration in the form of another file, which is then placed in a prespecified location. Then specialized classes take the configuration file, parse it and then perform the initialization based on it. One flaw of this approach is that in this way, it is hard to use a framework in a programatic way, thus limiting the interoperability. By using a framework in a programatic way, we highly increase the component aggregation level. For instance, Servlet containers are meant to be standalone applications, as this is also a tendency for frameworks in general. But if we want to embed a servlet container in another application, configuration problems appear. We would normally want to modify different parameters in the configuration file, and then make use of the framework. But if the framework can only be configured by means of a configuration file, then this whole process is considerably hard. We can eliminate this inconvenience by having a class in the public API that can handle the configuration process. By having such a class, we can make sure that the process of integrating a framework in another software component is easy.

Suggestion 3. *It should be possible to express the configuration of a framework by a single class.*

What does this suggestion of simplified configuration say is that there should be one object that encapsulates all data necessary for the configuration. One advantage of using this approach is that the object reflection can be used to determine the properties of the object configuration, thus significantly simplifying the configuration task. Moreover, a specialized tool can extract the properties of the object and present a graphical user interface to interactively build such an object, thus further simplifying the component integration process, or we could serialize this object to send it over a network or to store it in a convenient format. Also, we have only one object we have to concentrate on when reading the documentation in order to understand the configuration needs of a framework.

Other advantage is that in the case of a new framework, if the new framework needs a new configuration object, it can extend the old one to offer new configuration functionality. In this way, the old framework can use the new configuration file, because of the parametric polymorphism property of a class.

Suggestion 4. *Standard programmatic access should be provided for a framework life cycle.*

More precisely, this framework's skeleton means that a framework should be expressed by implementing a standard interface:

```
public interface Framework{
        public void configure(Obect configuration) throws Exception;
        public void start() throws Exception;
        public void dispose() throws Exception;
}
```

By having a framework implement the standard lifecycle steps defined previously, we make sure that a framework can be seamlessly integrated with another software component. Unfortunately, there are many software components that do not even allow programatic access to their functionality, making the integration task very difficult.

Such a declaration of a framework shows its potential when we want to change a framework implementation by another.

4. COMPARISON TO OOP

The modules as defined above have strong similarities with the classical object-oriented programming.

Encapsulation. The same way an object encapsulates data and methods to process the data, a module encapsulates the description of a contract and the classes that perform the contract.

Inheritance.

Definition 4.1. *We say that module B* ***inherits*** *from module A when the conditions from Definition 2.6 are met.*

The same way a class can inherit another one, a module inherits another one if there are classes its Public API inherits the old ones. The same way objects deliver new functionality by inheritance, a module offers new functionality by extending the Public API. The Public API can be compared to a method signature, while the rest of the classes (private classes) could be interpreted as the actual body of the method.

Polymorphysm of a module is based on the polymorphism of the classes in the Public API.

Module inheritance is strongly related to the notion of replaceability: if a module A inherits module B, it can safely replace it in a software component. This feature, combined with using a Java ClassLoader, yields the possibility to replace a module at runtime, that is without stopping the application.

References

[1] Gosling, J., Joy, B., Steele, G., Bracha, G., *Java(TM) language specification (2nd ed.)*, Prentice Hall, PTR, June 5, 2000.

[2] Sun Microsystems *JAR File Specification*, http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html

[3] Sun Microsystems *Java Servlet Technology*, http://java.sun.com/products/servlet/

[4] Sun Microsystems *Javadoc Tool*, http://java.sun.com/j2se/javadoc/

[5] Sun Microsystems *JavaServer faces technology*, http://java.sun.com/javaee/javaserverfaces/

[6] Meyer, B., *Design by contract*, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[7] Meyer, B., *Design by contract*, Advances in Object-Oriented Software Engineering, D. Mandrioli, B. Meyer (eds.), Prentice Hall, Englewood Cliffs, 1991, 1-50.

[8] Meyer, B., *Applying design by contract*, Computer (IEEE), **25**, 10(1992), 40-51.