Stabilizing Component APIs with Meta Tags

Stefan Roock

Roock@informatik.uni-hamburg.de

University of Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg
Germany

http://swt-www.informatik.uni-hamburg.de

APCON Workplace Solutions GmbH
Friedrich-Ebert-Damm 143
22047 Hamburg
Germany

ABSTRACT

Large modern application systems should be organized as a set of components. That way the components are rather independent from each other and may therefore evolve independently. This goal is seldom achieved in practice. One reason is the usage of frameworks which have a complex API to the application code. Therefore modifications to frameworks often affect the framework API.

This paper discusses formal concepts to capture framework modifications and relates the introduced formal notions to practical issues like Java deprecated tags. I claim that framework APIs can be stabilized with meta tags without degenerating the architecture caused by backward compatibility.

Keywords: framework, component, versioning, migration, base modification, composite modification, history, object orientation, sigma calculus

MOTIVATION AND OVERVIEW

Large application systems should be organized as a set of components. That way the components are rather independent from each other and may therefore evolve independently. With the same technique it is possible to use commercial available components to build the application system. If the interfaces between the components are clearly defined it is even possible to change your mind later on and replace buyed components by self developed ones or vice versa.

These goals are seldom achieved in practice. The reasons range from weak architectures to bad programming habits of the developers. In this paper I focus on one special reason: Frameworks provide much more support for application development than components. In addition to components, frameworks define an architecture for whole application systems. This is achieved by the definition of general super classes for the application classes. The result is a complex API between the application system and the framework which makes large portions of the framework public to the application system.

The source of the versioning problem is the otherwise extremely useful open-closed principle introduced by Meyer (cf. [Meyer1997]). The open-closed principle means that the interface of a class can be fixed (i.e. closed) for re-use by other clients; at the same time the interface can be evolved (i.e. open) through subclassing for further development. This well-established principle must be revisited for framework development.

Applications using frameworks usually have to subclass several classes of the (so-called white-box) framework. If the framework evolves, the superclasses of the application classes may change, thus invalidating the application program. This problem is also known as the „fragile base class problem“ (cf. [Szyperski1997]).

If the framework evolves it is often necessary to modify its API. Therefore the applications have to migrate to new framework versions. We face this problem in our daily work on the JWAM framework (cf. [JWAM], [Lippert et al. 2000], [Roock2000]).

This may be a major effort. This paper deals with formal foundations of the sketched problem and ways to solve it. The Java meta tag concept is sufficient to stabilize a component's or framework's API.

FORMAL FOUNDATIONS

Components and Frameworks

I use the sigma calculus (cf. [AbadiCardelli1998]) as a formal base for the concepts of components and frameworks. The sigma calculus is suitable to describe object oriented systems written in the common object oriented programming languages like Java, C++ or Smalltalk. The basic ideas of the sigma calculus are:

- No explicit notion of *class*. A class is simply an object which generates objects.
- *Methods* are first class citizens: A method has no name but can be assigned to a name (cf. lambda calculus, [Curry et al. 1968], [Curry et al. 1972]).
- No explicit notion of *field*. A field in an object is simply a method with a constant result. This results in the ability to update method implementations at runtime. In the case of a field, the method implementation is updated with a new constant implementation returning the new value of the field.
- No explicit notion of *inheritance*. Inheritance may be simulated by copying methods from other objects.
- Objects and types are strictly separated. An object does not define a type by itself.
- Structural matching of types: A type is a subtype if it has all methods of the super type. An object implements a type if the object implements all methods of the type.
- The typical control structures like alternative (if) and loop (for, while) can be simulated by object methods. For example the for-loop may be supported by a method *for* at object *IntegerClass*.
- Functional or imperative evaluation: The sigma calculus supports a functional as well as an imperative evaluation strategy. For the rest of the paper I assume the imperative evaluation strategy since the widespread OO languages are imperative.

The sigma calculus defines *types*, *objects*, *method signatures*, *method implementations*, *method calls* and *names* (to which a type, object or method may be assigned) as explicit concepts. It doesn't define explicit concepts for *classes*, *inheritance* or *control structures*.

The Concepts of History and Base Modification

A history is simply an ordered collection of modifications. It is suitable to define a minimal set of modifications. I call these *base modifications*. I use the concepts of the sigma calculus as the ground to define the base modifications.

Element	Create	Delete	Update
Type	possible	possible	unneccessary
Object	possible	possible	unneccessary
Method	possible	possible	unneccessary
Name	senseless	senseless	possible
Signature	senseless	senseless	possible
Implementation	senseless	senseless	possible

This results in the following set of base modifications:

- For types: createEmptyType, removeType, createTypeMethod, removeTypeMethod, updateTypeName, updateTypeMethodName, updateTypeMethodSignature
- For objects: createEmptyObject, removeObject, createObjectMethod, removeObjectMethod, updateObjectName, updateObjectMethodName, updateObjectMethodSignature, updateObjectMethodImplementation

For every of these 15 base modifications the preconditions and postconditions may be defined quite easily. What matters here additionally is the compatibility of the base modifications: Some are *compatible*, some are *incompatible* and others are *automatable*. With the help of the sigma calculus it is possible to proof the compatibility of each modification. We then can describe every base modification with the following scheme (here shown with the modification *createTypeMethod* as an example):

Name: createTypeOperation

Parameter: Typename T, Methodname O, Method signature S

Preconditions: T exists, O doesn't exist in T

$$\exists a \in TypeCollection : a.Name = T$$

$$\neg\exists b \in a.Def : b.Name = O$$

Postconditions: T has method O with signature S

$$\exists a \in TypeCollection : a.Name = T$$

$$\exists b \in a.Def : b.Name = O \wedge b.Signature = S$$

Compatibility: incompatible (since objects of the applications which conformed to the type may no longer conform to the type)

$$isIncompatible((createTypeMethod, T, (T, O, S)) =$$
$$\neg isCompatible((createTypeMethod, T, (T, O, S)) \wedge$$
$$\neg isAutomatable((createTypeMethod, T, (T, O, S)) \rightarrow$$
$$\neg false \wedge \neg false \rightarrow$$
$$true$$

$$isCompatible((createTypeMethod, T, (T, O, S)) =$$
$$\neg\exists a : isDependent(a, T) \rightarrow$$
$$false$$

$$isAutomatable((createTypeMethod, T, (T, O, S)) =$$
$$\exists f : f(a) = b : T \in a \wedge (T, O, S) \in b \wedge$$
$$isComplete(a) \wedge isComplete(b) \rightarrow$$
$$false$$

The following table shows the compatibility of the base modifications.

Modification	Compatibility
createType	isCompatible
deleteType	isAutomatable
createTypeMethod	isIncompatible
deleteTypeMethod	isIncompatible
updateTypeName	isAutomatable
updateTypeMethodName	isAutomatable
updateTypeMethodSignature	isIncompatible
createObject	isCompatible
deleteObject	isAutomatable
createObjectMethod	isCompatible
deleteObjectMethod	isAutomatable
updateObjectName	isAutomatable
updateObjectMethodName	isIncompatible
updateObjectMethodSignature	isIncompatible
updateObjectMethodImplementation	isCompatible

Composite Modifications

Additionally to the concept of base modifications a concept of *composite modification* is required. This is due to the fact that the described base modifications seldom occur as such in the widespread OO programming languages. For example the creation of a public class in Java will consist of the two base modifications *createType* and *createObject*.

I define the notion of composite modification as a sequence of modifications. For example the modification *createJavaClass* is the sequence *createObject;createType* which means that first the object is created and then the type. To ease the handling of composite modifications I assume that every composite modification is composed from exactly two modifications. Since every modification may be a composite modification itself, this constraint does not restrict the concept.

It is relativly easy to define the pre- and postconditions of composite modifications:

- The precondition of the composite modification is the precondition of the first modification in the sequence plus the parts of the precondition of the second modification which are not automatically fulfilled by the postcondition of the first modification.
- The postcondition of the composite modification is the postcondition of the second modification in the sequence plus the parts of the postconditions of the first modification which are not directly related to the postcondition of the second modification.

The compatibility of composite modifications is not that easy. If two modifications have the same compatibility the resulting composite modification should have this compatibility also. Therefore the modification *createJavaClass* is compatible.

What happens if the compatibilities of the two modifications differ, is not totally clear until now and subject to future research.

Meta Tags

In some programming languages (Java [ArnoldGosling1998], Eiffel [Meyer1997]), it is possible to tag interfaces, classes and operations as deprecated (in Java with the deprecated meta tag; in Eiffel, with the obsolete keyword). If a class uses a deprecated interface, class or operation, compilation and execution of the program is possible but the compiler issues warnings. This way, the application developers may use the warnings to migrate to the new framework version in small steps.

If an operation in the framework becomes superfluous, it may be marked as deprecated and removed in a later version. If an operation is renamed, a new operation with the new name may be introduced and the old one may be marked as deprecated.

This mechanism is powerful and easy to use, but it has its problems. One problem is caused by moving classes between packages or renaming packages. In this case, all classes within the package will become deprecated. Since these deprecated classes have no inheritance relationship with the new classes, typing problems will occur.

Another problem is that deprecated tags do not cover all possible refactorings (cf. [Fowler1999], [Opdyke1992]). Renaming of classes or modifications of the signature of an operation cannot be easily covered with the deprecation mechanism.

But all these problems can be solved with the introduction of three additional meta tags: the default tag, the ID tag and the deprecated-inheritance tag. Every abstract method may have a default tag which specifies a default implementation. If a new abstract method is introduced into the framework, inheriting application classes may no longer be concrete classes. With the help of the default tag it is easy to develop a tool which generates the new method with a default implementation into the application classes.

The ID tags assigns an ID to every interface, class, method and field. It is easy to create a small tool which generates ID tags into new or modified source code. With the ID tag it is easy to detect renamings of interfaces, classes, methods and fields as well as movements of interfaces and classes between packages. That way is addresses one of the main weaknesses of the deprecated tag. A tool which automatically does the necessary renamings in the application is easy to develop.

Sometimes it is necessary to tag an inheritance relationship as deprecated (for example to transform an inheritance to a delegated). This isn't covered by the deprecated tag. Therefore I introduced the deprecated-inheritance tag. This tag allows the usage of both the sup and the super class both warns of the application codes makes use of polymorphic assignments.

With the help of the sigma calculus it can be shown that the described meta tags are sufficient to stabilize the framework's API.

DISCUSSION AND FUTURE WORK

The general aspects of versioning frameworks and migrating application projects have been discussed. A formal approach to the problem based on the sigma calculus has been sketched. The notions of history, base modification and composite modification were introduced.

The formal concepts were related to practical issues like meta tags.

REFERENCES

[AbadiCardelli1998] M. Abadi, L. Cardelli: *A Theory of Objects*. 2nd Edition, New York, Springer, 1998.

[ArnoldGosling1998] Ken Arnold, James Gosling. The Java Programming Language. 2nd Edition. Addison Wesley. 1998.

[Curry et al. 1968] H.B. Curry, R. Feys, W. Craig: *Combinatory Logic - Volume I*. Amsterdam: North-Holland Publishing Company, 1968.

[Curry et al. 1972] H.B. Curry, R. Feys, W. Craig: *Combinatory Logic - Volume II*. Amsterdam: North-Holland Publishing Company, 1972.

[Fowler1999] Martin Fowler. Refactoring. Improving the Design of existing Code. Addison-Wesley. Reading Massachusetts. 1999.

[Java1998]. Java Product Versioning Specification. http://www.javasoft.com. February 10, 1998.

[JWAM] The JWAM framework. http://www.jwam.de. 1999.

[Lippert et al. 2000] M. Lippert, S. Roock, H. Wolf, H. Züllighoven: JWAM and XP - Using XP for framework development. Proceedings of the XP2000 conference. Cagliari, Sardinia, Italy, 2000.

[Meyer1997] Bertrand Meyer. Object-Oriented Software Construction. Second Edition. Prentice Hall. New Jersey. 1997.

[Opdyke1992] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD Thesis. University of Illinois at Urbana-Champaign. 1992.

[Roock2000] S. Roock: eXtreme Frameworking - How to aim applications at evolving frameworks. Proceedings of the XP2000 conference. Cagliari, Sardinia, Italy, 2000.

[Szyperski1997] Clemens Szyperski. Component Software. Addison-Wesley. Harlow, England. 1997.

ACKNOWLEDGMENTS

I wish to thank the following colleagues at the University of Hamburg for supporting my work, namely: Heinz Züllighoven, Guido Gryczan, Henning Wolf and Martin Lippert.