Refactoring Tags for automatic refactoring of framework dependent applications

Stefan Roock
Apcon Workplace Solutions &
University of Hamburg
Vogt-Kölln-Str. 30
Hamburg, Germany
+49 40 42883 2302
roock@jwam.de

Andreas Havenstein
Apcon Workplace Solutions

Friedrich-Ebert-Damm 143
Hamburg, Germany

havenstein@jwam.de

ABSTRACT
We describe the concept of *refactoring tags* which supports XP for framework development – especially simple design, refactoring and short releases.

A set of four refactoring tags (similar to Java meta tags) reify modifications done to the framework in its source code. Migration tools interpret the refactoring tags and support application developers when migrating to a new framework version with a changed API.

Keywords
Refactoring, framework, refactoring tag, meta tag.

1 MOTIVATION AND INTRODUCTION

Refactoring frameworks puts extra load on refactorings, (cf. [2]) since most refactorings change the framework API. Framework dependent applications have to migrate to new framework versions (cf. [4]). The caused efforts hinder framework refactorings. Therefore frameworks are often designed up front and then their API is maintained stable. This is in contrast with XP techniques of simple design, merciless refactoring and short releases. We experienced problems with these XP techniques for the JWAM (see [1]) framework development.

Based on our experience we developed the concept of *refactoring tags* which supports XP for framework development. Refactoring tags reify modifications done to the framework in the framework source code. Migration tools interpret the refactoring tags and support application developers when migrating to a new framework version with changed API.

The migration tools are implemented for Java on the base of Java meta tags and can be transferred to other programming languages easily.

2 MODIFICATIONS

Refactorings create modifications like the following:

- Move class or interface to another package
- Create, remove, rename class or interface
- Create, remove, rename method or attribute
- Change modifiers of class, method or attribute
- Create or remove inheritance between classes and interfaces
- Change method return type or parameter list
- Change method semantics (contract of the method, cf. [3]).

These modifications can be assessed by their compatibility:

Compatible	No changes to framework API.
Incompatible	The application has to be migrated manually.

A modification is *compatible* if it does not change the API of the framework and doesn't therefore cause any migration effort for the application. An *incompatible* modification changes the API of the framework and causes migration efforts for applications. Nearly all of the above modifications are incompatible.

3 REPRESENTING MODIFICATIONS IN SOURCE CODE

The main idea of our approach is to represent modifications in source code in an abstract way. For Java we use *meta tags* for this purpose. We define the following meta tags which we call *framework tags*:

- **Past:** Denotes the previous version of the signature of a class, interface, method or attribute. The signature of a class or interface is defined by its modifiers, name and super classes and super interfaces.
- **Future:** Denotes the coming version of the signature of a class, interface, method or attribute.
- **Paramdef:** Defines default values for parameters.
- **Default:** Defines a default implementation of abstract methods.

Past Tag
The past tag denotes the previous version of an element. Consider the class *Customer* which inherits from *BusinessObject*.

```
/**
 * @past public class Client
 *         extends BusinessObject
 */
public class Customer
  extends BusinessObject [...]
```

It is obvious that the class was renamed from *Client* to *Customer* and that the modification is automatable.

Future Tag
The future tag is similar to the past tag but directed into the future. Let's assume that the framework developers want to remove the inheritance relation between *Cus-*

tomer and *BusinessObject*. In this case they don't perform the modification directly but announce it with the future tag.

```
/**
 * @future public class Customer
 */
public class Customer
  extends BusinessObject
```

Since the future tag does not denote the inheritance it is clear that the inheritance relation will be removed. Now the application developers have to remove all polymorphic assignments of customers to business objects. The advantage is that the application developers have at least one framework version cycle to adapt the application. Therefore migration is much more smoother since the application is compilable during the whole migration process.

Paramdef Tag

The paramdef tag denotes default values for methods. If the parameter list changes as a result of a refactoring, the corresponding method calls or overwriting methods can be migrated automatically. Changes of a parameter list are recognizable for a tool via the past tag described above. The default values are needed in two cases:

- **A parameter is added to a framework method.** For migrating method calls within application classes to the new method signature, a migration tool can automatically add the given default parameter in calls of framework methods. Consider the following refactored framework method:

```
/**
 * @past public boolean comp(int
c)
 * @paramdef delta = 0.001
 */
public boolean comp(int c,
  float delta)
{
  return abs(_value - c) < delta;
}
```

In application classes calls to this method can be automatically completed with the default value for the new parameter.

Call with old signature: `obj.comp(42);`
Call after migration: `obj.comp(42, 0.001);`

- **A parameter is removed from a framework method**
  In this case the default parameter values are used to keep the code consistent inside of overwritten methods in application classes. For demonstration we reverse the refactoring of the example above. The comparison method in the framework is now reduced to a single parameter method:

```
/**
 * @past public boolean comp(int
c,
 *          float delta)
 * @paramdef delta = 0.001
 */
public boolean comp(int c) {[..]}
```

An overwriting method in an application class derived from that framework class has now to be migrated to a single parameter method. This can be done by moving the former parameter to a local variable with the same name and the value of the default parameter.

The old implementation of the derived application method has two parameters:

```
public boolean comp(int c, float
delta)
{
  return abs(_value - c) < delta;
}
```

After the migration it is reduced to a single parameter method. The removed parameter is replaced by a local variable initialized with the given default value:

```
public boolean comp(int c)
{
  float delta = 0.001;
  return abs(_value - c) < delta;
}
```

Default Tag

If new framework methods are defined in an interface or an abstract class, the default tag defines a default implementation for these methods. This is important for the migration of derived or implementing application classes. New framework methods can be detected via the since tag. These new defined methods with the default implementation can be automatically inserted into the application classes. Consider the *customer* interface with the new method *getName* inserted.

```
public interface Customer
{
  /**
   * @since 1.2
   * @default return "no name";
   */
  public String getName();
}
```

In all implementing application classes the new *getName* method can be inserted automatically with the default implementation:

```
public String getName()
{
  return ”no name”;
}
```

4 ADDITIONAL EXAMPLES

In Section 3 we described some basic modifications applied to the framework and how the tags provide a means to support the application classes migration. There are some cases which seem to be more difficult but which can also be handled by analyzing the framework tags.

Examples for non-automatic-migratable changes to the framework are

- **Change of the return type of methods**
  If the return type of a method changes, all occurrences of method calls had to be adapted to that new return type. This is not automatable and the application code would not be compilable any more due to these incompatible changes.
- **Change of contracts that specify the semantics of a method**
  Application code based on methods with the old semantics would be compilable but could lead to runtime problems and exceptions due to the changed contracts.

To avoid the semantic problems and keep the application code compilable a combination of copying and renaming framework methods is a solution.

Lets assume we want to change the semantics of a framework method. In the former framework version the *description* method provided access to some informations with an index starting at 0. In the new version the first element is accessed by an index starting at 1.

The new version of the method denotes the old precondition (*require*) of the method’s contract:

```
/**
 * @contract require 1<=index<=count
 * @past require 0<=index<count
 */
public String description(int index)
{
  assert 1 <= index && index
<=_count;
  return _entries[index-1];
}
```

The application code relies on the old contract with indices starting at 0 and is not automatic migratable to the new semantics.
To keep the existing applications consistent with the new framework version, the old method with the old semantics is copied and renamed:

```
/**
 * @contract require 0<=index<count
 * @past public String
 *         description(int index)
 * @future #undefined
 */
public String
  deprecated_description(int index)
{
  assert index 0 <= index < _count;
  return _entries[index];
}
```

Detecting the past tag a migration tool can change automatically the application classes to use the *deprecated_description* method which provides the proper semantics for the old application classes.

The future tag with the *#undefined* value indicates that this method will be removed in future versions of the framework. The future tag with value *#undefined* is equivalent to the Java deprecated tag.

A similar procedure is applicable to methods with a changed return type.

The resulting application code is compilable and has the same behaviour as the old application code with the old framework. Again a smooth migration is possible.

5 COMPATIBILITY WITH REFACTORING TAGS

With these meta tags the compatibility classes increases by three. Now we have:

Compatible	No changes to framework API.
Automatable	A software program can migrate the application to the new framework version.
Semi-Automatable	A software program can migrate the application. Application developer has to make some choices from a limited set.
Deferred-Incompatible	An incompatible modification is announced but not done yet. Application developers have at least one version cycle to migrate the application.
Incompatible	The application has to be migrated manually.

When the framework tags are used, most modifications change their compatibility from incompatible up to ”better” compatibilities.

The following table shows for some framework modifications the compatibility classes reached by the usage of our tags:

Modification	**Tags used**	**Compatibility class**
rename method	past	automatable
rename class	past	automatable
add interface method	since, default	automatable
change method signature	past, paramdef	automatable
change method	past, future	deferred-

return type		Incompatible
change method semantics	past, future	deferred-Incompatible

6 TOOL SUPPORT

For the framework tags to work properly in professional contexts tool support is necessary:

Past Tag Generator	Generates past tags for all elements with their current signature.
Migrator	Performs automatable migrations and creates todo lists from deferred-incompatible modifications.

The *Past Tag Generator* is used by framework developers whenever the development of a new framework version is started. Then it replaces all existing past tags with new ones. The new past tags first refer to the current version of the annotated element. When framework developers modify an element the past tag denotes the previous version of the element.

The *Migrator* is used by application developers to migrate applications to new framework versions. The migrator can be configured with a framework fitted with tags and the application to be migrated. The migrator tool then generates a todo list. A todo list entry indicates the position in the application source which has to be adapted due to the changes in the framework. The entry describes the framework modification and the compatibility class. All automatable refactorings can be performed by the migrator tool automatically. All semi-automatable refactorings can be interactively performed. The application developer has to choose between the several alternatives presented by the migrator tool. Then the tool performs the choosen adaption. Occurrences of incompatible modifications are also indicated in the todo list but have to be performed and checked out manually by the application developer.

7 STATE OF WORK

The described concept is implemented as a prototype for the JWAM framework (cf. [1]). The described concepts should be usable for components, class libraries and sub systems as well.

One experience from first usages is that refactoring tags avoid using the ”change comments” feature of modern refactoring browser. This feature does not only rename a class or method and all references but is able to guess which comments have to be changed as well. The guessing is done based on string matching and finds the refactoring tags also.

8 ACKNOWLEDGEMENTS

We thank the collegues at Apcon Workplace Solutions GmbH for their support in testing the described ideas and concepts. Also we owe thanks to the reviewers for substantial enhancements for this paper.

REFERENCES

1. JWAM framework. http://www.jwam.org
2. M. Fowler: Refactoring: *Improving the Design of Existing Code*. Reading, Massachusetts, Addison-Wesley, 1999.
3. B. Meyer: *Design by Contract*. In: D. Mandrioli, B. Meyer (Eds.): *Advances in Object-Oriented Software Engineering*. New York, London: Prentice-Hall, 1991, pp. 1-50.
4. S. Roock: *eXtreme Frameworking - How to aim applications at evolving frameworks*. In: [5]. pp. 71-82.
5. G. Succi, M. Marchesi (Eds.): *Extreme Programming Examined*. Reading, Massachusetts, Addison-Wesley, 2001.