Upgrading of component-based application

Raphael Gfeller, Raphael Gfeller@sunrise.ch
Software and Systems
University of Applied Sciences Rapperswil, Switzerland
www.hsr.ch/mse

Change history

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10.2008</td>
<td>Before 0.1</td>
<td>Before “Milestone 1: outline and concept of seminar article reviewed”</td>
</tr>
<tr>
<td>16.11.2008</td>
<td>0.1</td>
<td>Most of the content has been done</td>
</tr>
<tr>
<td>17.11.2008</td>
<td>0.2</td>
<td>Internal reviewed</td>
</tr>
<tr>
<td>19.11.2008</td>
<td>0.3</td>
<td>After reviewed by Pascal Hauser</td>
</tr>
<tr>
<td>14.12.2008</td>
<td>1.0</td>
<td>Reviewed by J. Joller</td>
</tr>
</tbody>
</table>

Abstract
History, goals and problems for Component-based Software Engineering (CBSE) are described. Different available approaches are described and compared to handle change management on CBSE. Based on the approach of “automatically find applied refactorings between two versions of a component” new possible fields of application are given.
1. Table of Contents

Abstract ......................................................................................................................................... 1

1. Table of Contents .................................................................................................................. 2

2. Motivation ........................................................................................................................... 3

3. Component-based Software Engineering (CBSE) ............................................................... 4
   3.1. History ............................................................................................................................ 4
   3.2. Benefits for the software producer ................................................................................. 4
   3.3. Goals of the Component-based Software Engineering .................................................... 4
   3.4. Complexity of change management for building components ...................................... 5
   3.5. Approaches for realizing change management for building components .................... 7

4. Discussions .......................................................................................................................... 12
   4.1. Comparing of the approaches for realizing change management for building components ......................................................................................................................... 12
   4.2. New fields of application ............................................................................................... 13

5. Literature Cited ...................................................................................................................... 16
2. Motivation

Normally, software is based on different libraries, frameworks and an underlying operating system. Libraries, frameworks and the operating systems expose their interfaces that describe the way how another program can interact with them. This is known as API (Application Programming Interface).

Software’s normally have strong references to the API of some components. This results in the problem, that, if the API changes of a component, it becomes maybe incompatible with the existing already written software. For example, the name of the function “getPersons” changes to “getClients”.

New versions of libraries result in the positive way normally in more functionality, less bugs and performance improvements. At the other hand, changes of the API are often done by new versions of libraries and these changes are often not backward-compatible with the old version of the API.

Common is, that the programmer has to upgrade the software manually to the new API. This is an expensive and critical work which results in a dilemma for the software producer. He is interested in the new features and performance improvements of the component. On the other side, the cost for upgrading and the risk of the new behavior from the new version of the component exists.

Is there a way to reduce the costs and the risk of upgrading to a new version of a component? In chapter “Approaches for realizing change management for building components”, approaches are described that try to find an optimal way of dealing with this dilemma. But first, the engineering discipline is explained that focus on component based software development called Component-based Software Engineering (CBSE).
3. **Component-based Software Engineering (CBSE)**

This section will give an overview about the Component-based Software Engineering (CBSE) also known as Component-Based Development, (Christian Wachsmann, 2002 S. 7-8)

### 3.1. History

The idea that software should be componentized (like prefabricated things in other part of the industry) was first published at 1968 by Douglas McIlroy at the NATO conference on software engineering in Garmisch, Germany. (McIlroy, 1968 S. 88-98). Based on his ideas, pipes and filters were included within the UNIX operating system.

The modern concept of a software component was defined by Brad Cox of Stepstone who coined the term “Software ICs”. He published his ideas in (Cox, et al., 1986); based on them he invented the Objective-C language that adds Smalltalk-style messaging to C. (Cox, 1995)

IBM led the path with their System Object Model (Castellano, 1996) in the early 1990s. As reaction of this, Microsoft invented Object Linking and Embedding in 1990, known as OLE (Skolob Matrix, 2007) and Component Object Model (Microsoft, 2008), COM.

### 3.2. Benefits for the software producer

As listed in (Mohagheghi, et al., 2003 S. 2) the following benefits exits for the software producer that use components:

- Shortened development time, and reduced total cost, since systems are not developed from scratch.
- Facilitation of more standard and reusable architectures, with a potential for learning.
- Separation of skills, since much complexity is packaged into specific frameworks.
- Fast access to new technology, since we can acquire components instead of developing them in house.
- Improved reliability by shared components

### 3.3. Goals of the Component-based Software Engineering

Based on (Crnkovic, et al.), (Radinger, et al., 2003) and (Garlan, et al., 2001) the following goals for the CBSE method are identified: 1. increase the productivity, quality, and time-to-market in software development thanks to the deployment of both standard component and production automation. 2. Build software systems from standard components rather than "reinventing the wheel" each time, thinking in terms of system families rather than single systems. 3. Being able to find, adapt, and incorporate disparate components to form working, reliable applications.
### 3.4. Complexity of change management for building components

Because there are so many stakeholders invited during change management for components, it becomes a very complex topic. The following common basic steps describe the challenges of developing a component.

<table>
<thead>
<tr>
<th>Change management step</th>
<th>Event</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>First release of component</td>
<td>The component C Version 1 ([C_1]) is released by the producer with the following signature.</td>
<td>Interesting stakeholders writes software (A_1)-(A_n) that are based on (C_1).</td>
</tr>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Bug fixing with no API changes</td>
<td>Based on feedback of the stackholders or internal testings, bugs of the current release are removed. The API of the component does not changed. This results into (C_1').</td>
<td>Most of the notified stakeholders are using the corrected version (C_1'). Cause no adapton has to be made by the stakeholders.</td>
</tr>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Bug fixing with API changes</td>
<td>Based on feedback of the stackholders or internal testings, bugs of the current release are removed. The API of the component has changed. This results into (C_1'').</td>
<td>Only few notified stakeholders are using the corrected version (C_1''). Cause they have to addapt their existing software. Some still using the old version.</td>
</tr>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Change management step</td>
<td>Event</td>
<td>Response</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>New features</td>
<td>Based on feedback of the stackholders or internal feedback, new features have been added to the current version. The existing API of the component does not changed. This results into $C_1'''$.</td>
<td>Notified stakeholders that are interested into the added feature are using the modified version $C_1'''$.</td>
</tr>
<tr>
<td>Remove feature</td>
<td>Based on feedback of the stackholders (legal issues, feature does not the right thing, ...) or internal feedback, existing features have been removed from the current version. This results into $C_1''''$.</td>
<td>Very few notified stakeholders will using the modified version $C_1''''$.</td>
</tr>
</tbody>
</table>
| Release of new version of the component | If there are      
- enough bugs solved  
- enough feature added  
- any internal structural changes to the component affected  
    the component developer will release a new version $C_n$. This step will include the following sub steps defined above:  
- Bug fixing with no API changes  
- Bug fixing with API changes  
- New features  
- Removed features | Depending on the changes, some notified stakeholders are upgrading to the new version $C_n$. |
### Change management step

<table>
<thead>
<tr>
<th>Event</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disposal</td>
<td>The component developer stops the developing on the component. The developer can extend the lifetime of his component by:</td>
</tr>
<tr>
<td></td>
<td>• Publishing the source code</td>
</tr>
<tr>
<td></td>
<td>• Still providing technical support</td>
</tr>
<tr>
<td></td>
<td>• Selling the source code</td>
</tr>
<tr>
<td></td>
<td>Based on the quality of the component and the preventive measure of the component developer the component is still using by application developer or not.</td>
</tr>
</tbody>
</table>

The goals of the producer of a component are that:

- As many stakeholders as possible are using his component
- The stakeholders are using always the most actual version (also if there are only removed features included). Because of the following reasons: 1. reduces the cost of supporting and handling multiple different versions, 2. for legal reasons, 3. will earn money by selling the new version of the component.

The producer has to be sure that the effort for stakeholders is minimal and the profit is maximized for the stakeholders. To ensure this, he can group some change management steps together. The most effort for stakeholders results by adapting their existing application to a new API of the component. Unfortunately API changes occur very often. The target will be to reduce the effort for the stakeholders if the API of the component changes. The next topic will describes the possibilities to do that.

### 3.5. Approaches for realizing change management for building components

A new component is shipped to the application developer. He has the following possibilities to handle the new version: 1. Ignore the new version, 2. Adapt his software for using the new component. That is the most done approach, 3. Adapt the shipped component to make it compatible with the exiting application, only available if the source code for the component is available, 4. Create a compatibility wrapper for the new component, so that the new component can be used as the old component. That is often done if the source code for the application is not available or the application cannot be modified.

For handling the new version of a component, the application developer has to know the changes between the old the new version. These following approaches will give an overview of finding changes between two versions of a component.

**Find changes that change the behavior**

Detecting changes that are changing the behavior of an application can only be detected by testing the interface of the component by the application developer. Anyway, the component developer should communicate these changes in the release notes. (Samw, 2004). For example, a method behavior is changed from version 1 to 2: `CalcSomethingVersion1(int value){ return value*2;}`, `CalcSomethingVersion2(int value){return value /2;}`.
**Find changes that are not changing the behavior**

Non behavior changes are often done by refactorings. For example most API changes in five widely used components were caused by refactorings (Dig, 2007 S. 36). They only improve reusability and maintainability and affect only the structure of a component and are meant to preserve the functional behavior of the component. (Dig, 2007 S. 24). For example, a methods behavior that is not changed from version 1 to version 2 can be realized by applying the refactoring “Rename”. For example: before refactoring: method CalcSomething(int value){return value*2;}; after refactoring: method CalcItNow(int value){return value*2;} which was renamed from CalcSomething to CalcItNow.

Refactorings can be defined as follow (Dig, 2007 S. 36), any change to programs can be regarded as a function program to program (p — p’) or as a program transformation. Program transformations are composed of primitive program transaction, which are called operations. There are two types of operations: those that change the semantics of a program, and those that do not. Refactorings are defined as a subset of operations that preserve the semantics of a program.

**Manual approaches**

**Approach 1, No communication**
The component developer does not communicate any API changes,

**Approach 2, Document API, Release Notes**
Release notes are documents that are normally shipped with a software program. They contain product state, product changes, contact information and so on.

Using the definition of release notes by (Samw, 2004), the software programmer writes the API Changes in the issues summary, enhancement section.

**Approach 3, Insert helper keyword into the programming language**
Almost all modern languages support a way to mark a program entity that is no longer recommended for use.

For example, the obsolete attribute in C#, see (Microsoft, 2005). The programmer can decorate a program entity with the obsolete attribute. Use of an entity that is marked obsolete will subsequently generate a compiler warning or an compiler error, depending on how the attribute is configured.

The following codes shows that a call to OldMethod terminates the compilation

```csharp
class B
{
    [System.Obsolete("use NewMethod", true)]
    public void OldMethod() {}
    public void NewMethod() {}
}

// Generates an error, terminating compilation:
b.OldMethod();
```

Another example is the @Deprecated annotation in java, see (Sun, 2004). As in C#, the @Deprecated annotation indicates that the marked element is deprecated and should no longer
be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation.

Using the definition of release notes by (Samw, 2004), a tool can generate an API-Changes-Documentation of obsolete or deprecated entities that can be placed in the release notes of the sipped component.

**Approach 4, Naming conventions**
For example, Eclipse (Eclipse, license@eclipse.org, 2008) developers use this approach when they extend an interface. (Dig, 2007 S. 19). For example, assume an interface IPerson {GetFirstName, GetSurname} on component version 1. In the next version of the component version 2 the IPerson interface is extend with the method GetName which returns GetFirstName + “ “ + GetSurname. A new interface IPerson2:extends IPerson {GetName} is created. New clients can used the new Version IPerson2 whereas old clients still using the old version IPerson.

**Approach 5, Coexistent**
Based on the approach above, if an interface has fewer methods than the old version, it will make the thing a bit trickier. A solution is that the two software versions will be coexistent. Calls from an application that is adapted to version 2 uses the component version 2, calls from older clients are still using version 1.

**Approach 6, COM-Style interface query**
Similar to the approach that has been mention above. A component provides multiply versions for example of an interface. The component provides the possibility to query for a well known interface of the client. (Dig, 2007 S. 19-20).

**Automatic approaches**

**Approach 7, Refactoring of the target application based on approach 3**
Jeff H. Perkins evaluates in his work “Automatically Generating Refactorings to Support API Evolution (Jeff H. Perkins, 2005)” a mechanism that finds calls to deprecated code within the application. Calls to deprecated code are then replaced by their bodies. Finding deprecated code is done with the in approach 3 described keywords.

For this method, the developer does not need to change his development practice; no additional artifacts or tools are needed for the component writer. No additional artifact has to be provided by the component writer. There is only a lightweight tool for the user that is using the component. Because of the limitation of this technique, only about 80% of all deprecated code can be replaced automatically, (Discussed in detail on section 3.2 (Jeff H. Perkins, 2005)). The limitations are:

- It only handles classes, methods and static final fields that are explicitly marked as deprecated.
- Calls to private methods within the component are limited
- If there is no replacement method. We have to keep the call and / or generate a warning. For example, Java’s Thread.suspend()
Approach 8, Record and replay refactoring
An extension to the refactoring engine records the refactorings that are applied on the component. These recorded refactorings shipped with the component. The application developer can replay this refactorings on its source code. This scenario was recently demonstrated in CatchUp (Henkel, et al., 2005 S. 274-283).

Approach 9, Detect refactorings automatic
While replaying refactorings as mentioned above, shows great promise, it depends on the existing refactoring logs. Unfortunately these logs are not available for existing legacy versions of components. Also logs will not be available for further versions because some developer will not use refactoring engines with recording capabilities, or other developer will do refactorings manually, or because that existing refactoring logs will not be shipped with a new version of the component. Daniel Dig proposes in his dissertation (Dig, 2007 S. 51-75) a novel algorithm that can detect sequences of refactorings between two versions of a component. An implementation of this new algorithm has been realized as an Eclipse plugin that is called RefactoringCrawler. This plugin is able to detect up to 85% of applied refactorings (evaluated on the following three components: EclipseUI 2.1 - 0, Struts 1.2.1 - 1.2.4 and JHotDraw 5.2 - 5.3). Detection is done by two analysis steps, a fast syntactic analysis that finds candidates for refactoring and a precise semantic analysis step that finds the actual refactorings.

The following section will give an overview of the algorithm that is used by (Dig, 2007 S. 54-63). Inputs are the two components C₁ and C₂. A tree for each component is generated while the nodes of the graph represent a source-level entity like a package, class, method or a field. Each node stores the full qualified name of the entity. The nodes are arranged hierarchically by their full quality names. A special fingerprint for each node is generated by a method that is called shingles. The fingerprint is added to the node.

Shingles, coined by (Andrei Z. Broder, 1987), have the characteristics that small permutation of an input string result in a small permutation of it shingles. Therefore, shingles enable detection of string with similar fragments much more robustly than traditional string matching techniques. Shingles are described in detail by (Andrei Z. Broder, 1987), shingling is internally based on the Rabin fingerprint algorithm (Michael O. Rabin, Harvard University, 1981). The goal of shingling is that if two documents that diver only in formatting or other information that we chose to ignore, for instance punctuation, formatting commands, capitalization, and so on, will be reduced to the same sequence.
Based on the generated shingles, refactoring candidates are found. Each candidate is a pair of similar entities from the two versions of the component. Similarity is defined as the normalized average (0..1) of the similarity of M1 to M2 and M2 to M1, where M1, M2 are multisets of shingles for two methods, classes, or packages. If the similarity value is above the user-specified threshold, the pair is deemed similar and is passed to the semantic analysis. This is an effective way to reduce the candidates that are progressed by the expensive semantic analysis. The figure above shows an example of the calculated shingles for two versions of the same method; remark that the multisets are nearly similar.

As the pseudo code shows, each detection strategy is applied until a fixed point is reached and that all strategies share the same log of detected refactorings. The following strategies are implemented so far and they are applied in the following order: 1. RenamePackage, 2. Rename Class, 3. RenameMethod, 4. PullUpMethod, 5. PushDownMethod, 6. MoveMethod, 7. ChangeMethodSignatures. Sharing of this log is important for successful detection of refactoring when multiple types of refactoring are happened to the same entity. For example, a method was renamed and the signature of the method has also changed.

The strategies compare whether an entity in one graph corresponds to an entity in another graph in respect to the already detected refactorings. The following renaming function is introduced: \( p(“\text{qualified name”}, \log) \) that maps a fully qualified name to the renaming in \( \log \) as one can see in (Dig, 2007 S. 59).

By (Dig, 2007 S. 59-61), the strategies compute the likelihood based on the references among the source code entities in each of two version of the component. The following rules are given for adding a reference between two nodes: 1. there is a reference from a node [method] \( m’ \) to a node[method] \( m \), 2. there is a reference from a node \( n’ \) to a node/class \( C \) if: \( n’ \) is a method that has (i) an argument or return of type \( C \), or (ii) an instantiation of class \( C \), or (iii) a local variable of class \( C \). or \( n’ \) is a class that (i) has a field whose type is \( C \) or (ii) is a subclass of \( C \). 3., there is a reference from a node \( n’ \) to a node/package \( p \) if \( n’ \) is a class that imports some class from the package \( p \). A reference has a multiple attribute which is increased if the same reference is added again. For example, if a method \( m \) calls a method \( m’ \) twice. Define \( \mu(n’,n) \) for the multiplicity from the node \( n’ \) to the node \( n \).

The directed similarity between two nodes is defined as, whereas \( rlog \) is equal to \( log \).

The overall similarity between two nodes is defined as:

\[
\delta(n, n’, rlog) = \sum_{n_1} \min(\varphi(n_1, n), \varphi(r(n_1, xlog), n’)) \\
\sum_{n_1} \varphi(n_1, n)
\]

\[
\sigma(n_1, n_2, rlog) = \frac{\delta(n_1, n_2, rlog) + \delta(n_2, n_1, rlog^{-1})}{2}
\]
Each strategy checks first if a pair is a likely refactoring. This is done via a fast syntactic check whether the pair is relevant for the refactoring and then performs a semantic check that determines the likelihood of the refactoring. The following table shows the condition for the RenamePackage method.

If the likelihood value is above the user-specified threshold, the refactoring is remembered.

The described algorithm overcomes three main challenges, as opposed to some previous algorithms that are mentioned in (Dig, 2007 S. 64). 1. It will scale up to code sizes that are used for real-world components as for example: Eclipse (Eclipse, license@eclipse.org, 2008). 2. It could handle noise that is introduced by backward compatibility handling in the components. As mentioned in “Approach 3, Insert helper keyword into the programming language”. 3. Can handle multiple refactorings happening to a single entity by using shared refactoring logs.

4. Discussions

4.1. Comparing of the approaches for realizing change management for building components

This section compares the approaches that are listed in the topic “Approaches for realizing change management for building components”.

Based on the comparison table bellow, the best method to use is discussed in Approach 8, Record and replay refactoring. Unfortunately the tool chain for using this method is not existent or not a standard fitting in the actual modern development environment. Further work has to be done in this direction. This work includes: 1. making the idea of the method public to the developers, 2. realizing a tool chain than can be used by everyone, 3. Make these tools as a standard fitting for modern development environment. 4. Shipping refactoring logs should be a part of new setup and deployment assistants.

Additionally to Approach 8, Record and replay refactoring release note should be shipped that include all behavior changes. Because of the behavior changes between two components and anyway it is very important that the application developer writes rigorously unit tests for validating the used component.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary target</td>
<td>To throw the application developer into the cold water.</td>
<td>Give an advice what has changed in the new component.</td>
<td>Supports co-existence in a way of adding functiona lity to given entities. Does not support removing functionality.</td>
<td>Supports co-existence of different version of a component by using switch statements.</td>
<td>Supports co-existence of different version of a component by querying the component for a well known interface.</td>
<td>Automatically adopts the component by the given advices of the approach 3.</td>
<td>Records the changes that are done by the component developer automatically. These are used by approach 8 to replay them on the application.</td>
<td>Finds the changes that are done by the component developer automatically.</td>
<td></td>
</tr>
</tbody>
</table>
### Attribute | Approach 1 | Approach 2 | Approach 3 | Approach 4 | Approach 5 | Approach 6 | Approach 7 | Approach 8 | Approach 9
---|---|---|---|---|---|---|---|---|---
Can handle behavior changes | -- | + | -- | -- | -- | -- | -- | -- | --
Can handle multiple version of a component | -- | -- | + | ++ | + | ++ | -- | -- | --
Can handle the following refactorings | None | All | All | All | All | All | All | All | Rename-Package Rename-Class Rename-Method PullUp-Method PushDown-Method Move-Method Change-Method Signatures so far
Work needed for the component developer | No | Yes, Writing the release note | Yes | Insert deprecated keywords into the code. | Yes | Support different versions. | Yes | Support different versions. | Yes | Insert deprecated keywords into the code. | No | Just do refactoring with a tool that can record refactoring and ship the refactorings with the component
Help provided for the application developer | - - Has to find all changes itself | - Advice to changes. (Accuracy 1%-100%) | + Advice to changes supported by the environment (Accuracy 1%-100%) | + can still using the old version. For using the new features, clear introduction are given. (Accuracy 100%) | + can still using the old version. For using the new features, clear introduction are given. (Accuracy 100%) | + Following the advice to changes are followed automatically (up to 75% accuracy) | + Component is updated two the new version with only few user inputs. (up to 100% accuracy) | ++ Component is updated two the new version with only few user inputs. (up to 85% accuracy)

### 4.2. New fields of application
Possible new practical applications based on the above approaches we discuss in this section.

**Detect an illegal usage of a component**
This application idea is explained by the following use case:

Company A has written a well tested component $C_A$ with its test result $R$ based on the test $T$. It sells them for using, but not for redistributing.

Company B has written another component, $C_B$. Company B has integrated $C_A$ in $C_B$ in an illegal way. For hiding the origin of $C_A$, they will apply some refactorings on it.

$C_B = \text{some functions} + C_A + \text{refactorings}$.

For a normal person, it is not obvious that $C_A$ has been integrated into $C_B$. 
Company A smelled a rat that Company B has illegally used $C_A$ in $C_B$. To validate its suspicion that $C_A$ is integrated into $C_B$, the company A can find the Refactoring used in $C_B$ based on the approach 9. Applying these refactorings on $T$, that results in $T'$. Rerun $T'$ for $C_B$, that results in $R'$. If $R$ and $R'$ are equal means that the chance that $C_A$ is integrated into $C_B$ is very high.

**Detect cheating of students**

This possible new field of application is based on a similarly approach that is used for “Detect an illegal usage of a component”. The idea is also described as a use case.

A teacher gives its students a task for solving a given problem in Java include write unit tests $T$ for it. The teacher will receive $n$ solutions $S$. If a student A cheats, he is going to take the solution $S_B$ from student B. He will fog $S_B$ so the origin of $S_B$ isn’t clear. During fogging of $S_B$ the behavior of $S_B$ does not change. Applying refactorings or do some edits that does not change the behavior of the application are the only possibilities to fog $S_B$.

$$S_B = \text{some edits} + S_A + \text{refactorings}.$$  

As in the application approach above, the teacher can find possible cheats of the students.

Remark: this application approach is only valid if the given problem will result in different solutions.

**Reduce the transferred data in a CVS (CVS) system by transferring refactoring**

In normal CVS (CVS) system, the changed file is transferred from the client to the server.

Let’s assume that the user changes the name of an often used class within a large project. There would be a large amount of data to be transferred from the client to the server.

My approach to reduce the network traffic for that scenario is, that refactorings which has been recorded based on “Approach 8, Record and replay refactoring, page 10”, are transferred from the client to the server. On the server they are executed again. This would decrease the data to be transferred noticeable. Another positive effect is that the refactorings are applied to the full solution (I guess some of you have already unloaded a unit test project to save compilation time, applied some refactorings, have reloaded the unit test project which results in some compile errors). For normal textual changes, the normal transmission mechanism is used.
Requirements for realizing this method are:

- Client has to be able to record refactoring, for example based on “Approach 8, Record and replay refactoring, page 10”
- Server has to be able to execute refactoring and he has to have knowledge about the project structure like for example Microsoft team foundation server does, (Microsoft, 2007).

**Extend change management within a small company**

Assume that a company owns some common code. This common code realizes basic functionalities, for example logging, basic business functions and so on. Based on the approach from above, extend the logic in the server part that refactorings are forwarded to all stakeholders. For example, if function doLogging() is renamed to doLog() on a given project, this refactoring will forwarded and applied on the common code and each project within the company that has a reference to the common code.

**Component Evaluation**

Approach 9 shows that it is possible to extract the refactoring steps between two different versions of the same component if version 1 and version 2 are known.

In the real-world, there is normally more than one component that does nearly the same job. That implies that the API of these components have to be similar. The behaviors of these components have to be similar too.

That result in the hypothesis, that it could be possible to find refactoring that reduces the manual work for adapting the component B if component A has used before.

If this hypothesis could be proved, the following positive effects for software writers would result: 1. Reduces the dependency on a concrete component vendor. 2. Adapting to a new component will also adapt the tests automatically. 3. Tools for automatic evaluation of components could be written. Steps of the tool; a. [Precondition: Program is using component A. Unit tests are written for component A.] b. Test the program with component A. c. Adapt the program for the usage with component B. d. Tests the program with component B. e. [for each component] do step c and d again. f. Displays the result to the software engineer. g. Converts the program for the usage with the best component.
5. Literature Cited


