Bachelor's Thesis

Using Version Control Systems for Tracking Adaptation-relevant Software Changes

submitted by

Ulf Wemmie

born 02.04.1981 in Oldenburg(Oldb.)

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: MSc Ilie Savga
Professor: Dr. rer. nat. habil. Uwe Aßmann

Submitted November 1, 2006

Contents

List of Figures

List of Tables

Glossary

AST	Abstract Syntax Tree	26
CVFV	Constant values in field variables	17
IDE	Integrated Development Environments	14
IS	Inheritance structure	17
SMC	Sequence of method calls	17
SVN	Subversion	22
UC	User classes	17
VCS	Version Control System	11
VSS	Visual SourceSafe	22

Chapter 1

Introduction

This is the first meaningful chapter, but all it contains is an introduction.

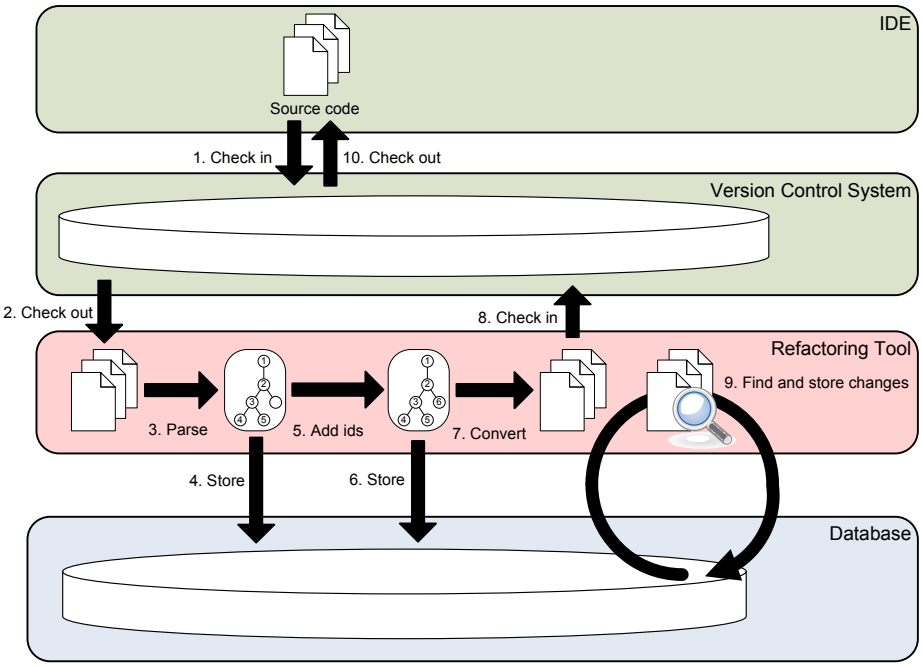

Figure 1.1: Tool architecture

1.1 Motivation

Refactoring is a standard means in software evolution. To explore low-level refactorings two snapshots of source code are required that should be obtained using Version Control Systems (VCS). Due to flexibility issues, a wide range of VCS' ought to be supported. It is expected that even a chain of mutually dependent refactorings can be explored. This at least semi-automatically data

mining must not lack one of the supported refactorings and has to be unambiguous. Furthermore, the work is limited to changes on the interface level. Changes to the methods' bodys, for instance, are not to be captured.

Change tracking processes can lead to two main errors: result sets can be wrong or incomplete. This can be avoided through the development and use of an annotation language that supports the identification of fine-grained entities independently of their names and properties. Every program entity such as class, interface, method, field, and parameter is assigned a global unambiguous number, the entity id:

```
1 [Id(3452345)]
2 public void setName(String name);
```

The annotation language should be human and machine readable allowing the developer to support the identification of refactorings.

Identifying refactorings can lead to two main errors: They can be discovered wrong or missed.

1. Discovering refactorings wrong can be avoided by adding global unambiguous numbers in form of annotations to the fine-grained entities on the interface level (Interfaces, Classes, Methods, Fields). By identifying entities independently of matching properties even a chain of mutually dependent refactorings mostly can be identified without the developer's help.

2. Refactorings will not be missed. Only in special circumstances, such as merging and splitting of methods, the knowledge of the developer is requested to classify details of the refactoring.

Version Control Systems often do not offer high performance and lack of transactions or configurations. A fast and easy access is reached by preprocessing and storing the information to a relational database. This process is partitioned into Data Extraction and Restoring Transactions:

The former extracts the interface-informations of a subset of files stored in the repository. This is done by parsing the obtained source code to an Abstract Syntax Tree (AST) which can be traversed to store the interface informations in the database. The latter restores transactions using Sliding Time Windows [ZW04]. Unambiguous transactions are required to identify refactorings faultlessly.

Unlike most related tracking strategies annotation-based tracking is not intended to track changes in the history of applications. Because

1.1.1 Goals

1.2 Methodology

Chapter 2

Background

2.1 Refactorings

As software grows old, the source code does, too. Software has to be maintained by changing and extending source code. By way of examining the alternation made in the evolution of different open source projects [DJ06] discovered that "over 80 % of these changes could be considered refactorings". Refactorings are changes to the structure of source code that preserve its behavior. In the context of binary-to-binary-compatibility only changes that alter the interface, particularly types, fields and method-signatures, are important. Other interface-preserving changes, for instance the low-level refactoring substitute algorithm introduced by [Fow99], which only alters the statements of a method body without changing the interface, are not covered. Most of the following strategies only consider refactorings. However, some could be extended to be capable of trachking other changes. Also, chains of refactorings, respectively refactorings composed of low-level refactorings, can only be discovered by a few strategies.

2.2 Related Tracking-Strategies

Below, different refactoring tracking strategies are discussed. Depending on the automation level they can be classified into automatic, semi-automatic and manual strategies. Human interaction is solely avoided in the former. Particularly semi-automatic strategies are not capable of avoiding the identification of false positives and negatives. A false positive is a wrong identified refactoring, whereas a false negative is a missed one. If they can potentially occur, the developer has to validate the results. Manual strategies need the developer's interaction, too. Instead of validating the results, changes or previous states have to be annotated to every program entity. In the following various different automatic, semi-automatic and manual strategies are presented. Finally, a short comparison is given.

2.2.1 Automatic

Many Integrated Development Environments (IDEs) contain refactoring tools, which allow users to apply refactorings faultlessly. Refactorings can be obtained

automatically by adding a logging mechanism to these tools. Thus, the problematical tracking of false positives and negatives is avoided. Besides, the developer does not have to validate the found changes.

Henkel and Diwan extend the refactoring engine of Eclipse [ECL] to capture refactorings [HD05]. These logged changes can be replayed later on other machines to maintain binary-to-binary-compatibility. Extending Microsoft Visual Studio's refactoring engine to convert this concept to the .Net environment seems to be disproportionally harder, because its code is not open source. Moreover, the developer is limited to IDE-supported refactorings and cannot manually refactor the code.

2.2.2 Semi-Automatic

Unlike the previous tracking strategy, refactorings are not logged in semi-automatic tracking, but tool-supportively discovered. Depending on the approach the identification is more or less error-prone. As "it is not always possible to unambiguously identify all refactorings" [GW05], semi-automatic tracking strategies are characterized by the developer's subsequent review in form of identifying false positives and negatives. Particularly with regard to the use of the results to automatically generate adapters between two successive versions, undiscovered and even wrong discovered changes are very problematic. They can lead to binary incompatibility in form of runtime-errors or even unperceived malfunction.

Algorithm-based Comparison

In algorithm-based approaches similar code fragments of two successive versions are expected to be equal. Their differences may have resulted from changes. As no annotations are required, its strength lies in the analysis of preexisting source-code. Nevertheless, refactorings are only identified in case of small deviations. [GW05] realized, that "often the refactorings are impure: the developer has not only performed the refactoring, but has changed other things at the same location at the same time". Particularly tracking of a chain of refactorings applied to the same entity is almost impossible. The amount of comparisons necessary for identifying refactorings is reduced by classifying entities into different sets, for instance into sets of methods and classes. Depending on the refactoring only a subset has to be analyzed.

Metric-based Heuristics

Current refactoring tracking research on metric-based heuristics restricts itself to refactorings that change the inheritance tree's structure. Standard-metrics in the field of inheritance and method- and class-size are used by [DDN00] to detect the following high-level refactorings:

- Split into superclass / merge with superclass
- Split into subclass / merge with subclass
- Move to superclass, subclass or sibling class

These refactorings are composed of Fowler's low-level refactorings up/push down method, extract sub-/superclass, collapse hierarchy. Software metrics characterize the source code. Combined and used as heuristics, they can analyse which methods and classes have been grown or shrinked in comparison to the previous version and whether the inheritance tree changed. This information is used to derivate the refactorings applied.

Fingerprints

Fingerprints can be seen as source-code characteristics allowing the identification of similar entities. Advanced analysis can easily detect which changes occurred between these entities. In the following only the techniques of Shingles and Birthmarks are taken into account.

Shingles Shingles in their main purpose are a means to detect the resemblance and containment of documents [Bro97]. This string matching technique is used by Danny Dig to identify refactoring candidates [DCMJ06, DJ06].

In detail, a shingle is a contiguous sliding-time subsequence. All shingles of size ω contained in a text, the so-called ω-shingling, is defined as a multiset. The 2-shingling of

```
String username = user.getName();
```

is the multiset of shingles of size 2 contained in (String, username, user, getName):

```
{(String, username), (username, user), (user, getName)}
```

For every method shingles are computed out of the tokens of the particular entity's (Javadoc-)comments and body. The sequence of tokens is divided into all subsequences of a given length by using a sliding window. To increase the performance of the ensuring comparison, all shingles are encoded with the help of Rabin's hash function[Rab81]. Furthermore, the amount of subsequences can be restricted using a heuristic. By comparing the shingle's hash-values of different versions, the containment and resemblance of entities can be detected. Every entity-pair of two successive versions that have a similar shingle encoding are expected as the same entity that may has changed over time. If and which refactoring has been applied to the entity is obtained by advanced analysis.

Birthmarks Birthmarks are characteristics of source code [TNMiM05]. It's original purpose is the identification of software being stolen and reused. There are four types of Birthmarks

- Constant values in field variables (CVFV)
- Sequence of method calls (SMC)
- Inheritance structure (IS)
- User classes (UC)

```
1  /**
2   * @past public class Client
3   *          extends BusinessObject
4   *
5   */
6  public class Customer
7    extends BusinessObject [...]
```

Figure 2.1: Refactoring tags describing the prior state of the method.

Sometimes method have fields with constant initial values. These CVFV can be used to identify a class. SMC are uses of types and methods of well-known classes, such as J2SDK. They can signature a method. To identify a class sometimes the inheritance structure and the used classes can be used. The same classes of successive versions may use the same well-known classes. Compared with the other techniques these insufficient identification mechanisms can only be used to support another approach.

2.2.3 Manual

As the name suggests in manual refactoring tracking user-support is needed. Changes or previous and future states are noted in source-code or other documents. Two approaches that annotate advanced information in the code are taken into account.

Annotations

Despite of the previous detection strategies in annotation-based approaches the source-code does not remain unchanged. Changes made to the interface can be described or retrieved by meta-information. A human readable approach is followed by [RH02]. Intuitive meta-tags are used to denote previous and future versions of elements (Fig. 2.1). As the state prior to the second last change is not provided, the whole evolutionary life cycle of the application is not covered. The application developer is forced to adapt his application to the framework very soon. Besides, this strategy is only applicable to changes within the same type-definition. Changes to the inheritance tree can not be identified.

[CN96] use annotations to denote all changes made to the interface, which leads to large change specifications in a cryptic and counterintuitive manner.

Advantages	Disadvantages
No false positives and negatives	Tool-Support required
More refactorings can be found	Application developer has to learn annotation language

Table 2.1: Advantages and disadvantages of annotation-based approaches.

2.2.4 Strategy Comparison

The strategies presented before have to be estimated with reference to their suitability for binary-to-binary-compatibility. A prerequisite is the lack of false positives and negatives. [DCMJ06] use precision and recall, two standard metrics, to measure the accuracy of their shingle-based tool RefactoringCrawler.

$$PRECISION = \frac{GoodResults}{GoodResults+FalsePositives}$$

$$RECALL = \frac{GoodResults}{GoodResults+FalseNegatives}$$

As no tracking of false positives and negatives is allowed, only a precision and recall of 1 is sufficient for binary-to-binary-compatibility. Besides the strategy introduced in section 4.1, this can only be achieved by automated and manual approaches. Unfortunately everyone of the remained strategies has its disadvantages: An automated approach may not be integrated into Visual Studio and limits itself to the refactorings supported by the IDE. Manual strategies require the developer to learn an annotation language and to take down every change applied. The approach introduced in section 4.1 can be classified between manual and semi-automated. It is semi-automated because of its automatic assignment of ids and it needs, like manual strategies, user-support for a few refactorings. In total compared with a manual approach the latter restricts the amount of user-interaction. A complete automatic approach integrated into Visual Studio fails unless it does succeed to get detailed information about the refactoring engine. Furthermore it is ambiguous whether the security concept prevents the access to the this engine.

	Automatic	Semi-automatic			Manual		
Refactoring	Capturing	Algorithm	Metric	Shingles	State	Changes	New
Add Parameter	++	+	-	+	o	+	++
Extract Class	++	++	++	+	o	+	++
Extract Hierarchy	++	++	++	+	o	+	++
Extract Interface	++	++	++	+	o	+	++
Extract Method	++	++	++	+	o	+	++
Extract Subclass	++	++	++	+	o	+	++
Extract Superclass	++	++	++	+	o	+	++
Move Class*	++	++	+	+	o	+	++
Move Field	++	++	+	+	o	+	++
Move Interface*	++	++	+	+	o	+	++
Move Method	++	++	+	+	o	+	++
Pull Up Constructor[1]	++	++	++	+	o	+	++
Pull Up Field	++	++	++	+	o	+	++
Pull Up Method	++	++	++	+	o	+	++
Push Down Field	++	++	++	+	o	+	++
Push Down Method	++	++	++	+	o	+	++
Rename Class*	++	+	-	+	o	+	++
Rename Field*	++	+	-	+	o	+	++
Rename Interface*	++	+	-	+	o	+	++
Rename Method*	++	+	-	+	o	+	++

Table 2.2: Comparison - Tracking of refactorings

Chapter 3

Data Preprocessing

Efficient tracking of refactorings always requires preparation of data. As direct access to repositories is very slow, and parsing each version in every comparison step needs a lot of computing time, all necessary information is stored to a database. Preprocessing of data consists of various steps described below. At first, in section 3.1, the characteristics of Version Control Systems are treated. Then, in section 3.2 a fundamental subset of the C# language specification is described followed by the syntactic analysis of data. Finally, section 3.4 depicts the data model used.

3.1 Version Control Systems

Version Control Systems in their original purpose allow teams of developers to work independently of each other on the same piece of code. In addition, they allow them to trace back or undo changes. Mainly, ASCII coded files are archived in repositories. Changes of the source code are made to local copies and checked in into the repository, afterwards. To not completely store every version, only differences are stored. Therefore, two successive versions of the same file are compared line by line. This comparison originates from diff, a file comparison tool for Unix. Its output in general only shows added, deleted or replaced lines. Below, information available in Version Control Systems are depicted. Afterwards differences and commonalities of different implementations are treated. Finally, transaction restoring techniques are introduced.

3.1.1 Versioning Information

Depending on the Version Control System, different amount of meta information is available. Especially details of versions, transaction and configurations are very important.

Versions

Version Control Systems work file based. Every revision of a single file is stored to the repository. These revisions are described by versions that in general contain at least the meta-information name, path, committer, timestamp and version number. Often a required or optional comment is assigned.

Transactions

A transaction is the set of all versions that have been added to the repository in conjunction. Every version is part of one single transaction. Many Version Control Systems such as Visual SourceSafe (VSS) or CVS[CVS] do not store information about transactions. A technique to retrieve these transactions is described in 3.1.3.

Configurations

The set of the newest versions available after a transaction is called a configuration. Transactions only include the versions that describe files which have been changed or deleted. A configuration contains both changed and unchanged files available after the corresponding transaction. Nevertheless, deleted files are not part of configurations.

3.1.2 Differences and Commonalities

Although more VCS' exist, only Visual SourceSafe (VSS) [VSS], CVS and Subversion (SVN) [SVN] are discussed in more detail. At first, the distinct models, Lock-Modify-Unlock and Copy-Modify-Merge, which are used in version control, are compared.

	Visual SourceSafe	CVS	Subversion
Atomic transactions	-	-	x
Change set support	-	-	o
Version number based on	file	file	transaction
Commit message per	file	change	?

Table 3.1: Different Version Control Systems in comparison

Lock-Modify-Unlock vs. Copy-Modify-Merge

The latter is supported by all of the researched VCS'. Although VSS enables the use of Lock-Modify-Unlock instead, it should be avoided in the context of change tracking. Lock-Modify-Unlock is characterized by a file locking mechanism, which allows single users to gain exclusive access to a resource. Others have to wait until the resource is unlocked. Particularly change tracking tools need access to all files that have been modified in conjunction. Only one locked file can avoid the preprocessing at a certain time, as individual source code changes may affect various files. Thus, a more suitable model, such as Copy-Modify-Merge, is needed. It allows multiple checkouts of resources at the same time, which avoids the problem of change tracking tools discussed before. However, merging conflicts, which have to be corrected by the user, may occur. Transactions containing unparseable files have to be corrected or discarded by a tracking tool, as they cannot be processed.

Atomic Transactions

Moreover, incomplete check ins may lead to incomplete transactions, too. Subversion avoids this lack of files by using atomic transactions. Either all changes

are checked in together or the transaction is rolled back leading to an unchanged repository. Because transactions are rarely supported by VCS', transaction restoring algorithms are discussed below.

Deleting Files

3.1.3 Restoring Transactions

As many Version Control Systems do not observe the checkins done together, a technique that restores transactions is needed. All versions that are not assigned to a transaction have to get sorted in ascending order depending on their time stamp. Then, those versions, that have been checked in within a specified interval, are attached to the same transaction. [ZW04] distinguish Fixed Time Windows and Sliding Time Windows.

In Fixed Time Windows all versions checked in within a fixed interval are assumed to belong to the same transaction. The starting time is always obtained from the first version. However, the check in of very big or many files may exceed the interval leading to incomplete transactions.

This is avoided by using a Sliding Time Window heuristic, which appoints the maximum temporal gap between two successive versions. Two successive versions are in the same transaction if the temporal gap between them does not exceed a given value. Otherwise, a new transaction is started.

3.2 C# Language Specification

The section's intention is not to give an introduction to C#. Therefore previous knowledge about the language is implied. Furthermore, the description of parts of the language has not the pretension of being complete. Only a small subset important for binary-to-binary-compatibility is taken into account. Moreover, due to the complexity only the most important language constructs are treated. Tracking changes requires information about the logical organization of the source-code. As assemblies are a means for physical packaging and deployment, they are out of scope. The outer closure of the logical organization of C#-entities are namespaces [Cor01]. Their members can be another namespace or types. Every type declared belongs to a particular namespace or implicitly to the global one.

3.2.1 Types

A type can either be a value type, a reference type or a type parameter. The variables of the former directly contain data, whereas reference types' variables contain references to their data. Value types are either struct types or enumeration types. Some predefined struct types, which are called simple types, are provided in C# by reserved words such as bool, byte, int, long, double or char. Reference types can be class types, interface types, array types or delegate types. Only class and interface types, which are probably the most important ones, are regarded in more detail. Type parameters are part of generics. Their treatment is designed to be part of future work.

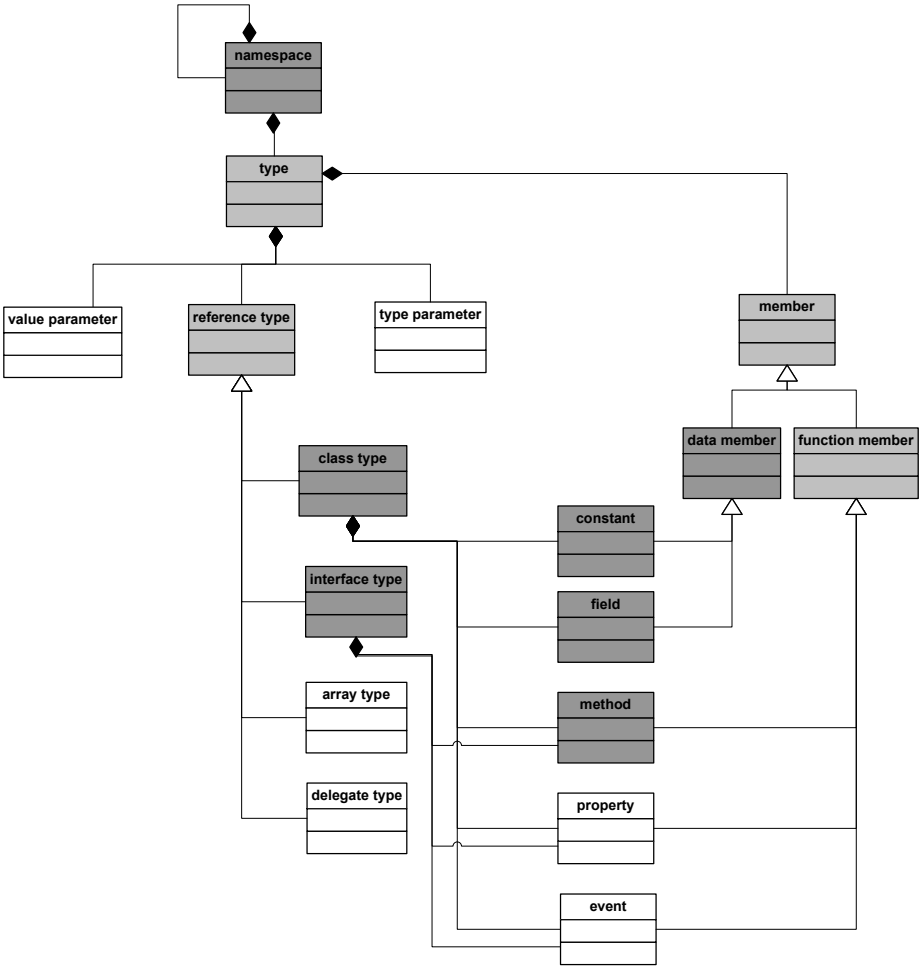

Figure 3.1: Covered parts of the C# language specification

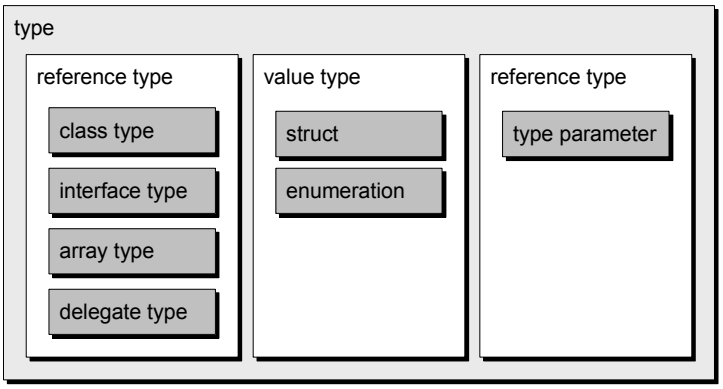

Figure 3.2: Types

Class

All classes by default are private, but the use of public, protected, internal, or private access modifiers is possible, too. Classes can implement multiple interfaces, but only extend a single class. Classes can be nested.

By using the abstract modifier, classes can be declared abstract. In contrast to general classes, abstract classes are incomplete classes that cannot be instantiates. They can only be used as base classes. Members of abstract classes are permitted, but not required to be abstract.

.Net 2.0 supports partial classes, a mechanism to split the declaration of a class to various files. Nested and partial classes are considered to be part of future change tracking..

Reference type	Possible function member
class	method
	property
	event
	index
	operator
	instance constructor
	destructor
	static constructor
interface	method
	property
	event
	index

Table 3.2: Allowed function members by reference type

Interface

Interfaces define contracts that are implemented by classes or structs. Like classes they can have a partial modifier. Referring to table 3.2 their members only can be methods, properties, events or indexes. An interface is public by default, but the access modifiers public, protected, internal or private can be assigned as well.

3.2.2 Members

C# distinguishes between the four access modifiers public, protected, internal and private to declare the accessibility of members or types. All members by default have an implicit accessibility level. Their default values and the allowed modifiers depend on the type they are declared in. As stated in table 3.3 class members have a private and interface members a public one.

Members are separated in two parts: data members and function members. The data members constants and fields, as well as the function members methods and constructors are described below. Further function members such as properties, events, indexers, operators, and destructors are omitted due to complexity reasons.

Members of	Default accessibility	Allowed declared accessibility
class	private	public
		protected
		internal
		private
		protected internal
interface	public	none

Table 3.3: Accessibility levels[Cor01]

Constants and Fields

Constants are constant variables computed at compile-time. As these class-members are implicitly static, the appropriate modifier is not required and not even allowed.

Fields on the other hand can be either associated with a class or an object. If the field is assigned a static modifier, it is a static variable associated with a type. Otherwise it is an instance variable associated with an object.

Methods

Methods are parts of objects, classes or interfaces. They are characterized by their signature. "The signature of a method consists of the name of the method, the number of type parameters, and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right"[Cor01]. In C# the parameter names and the return value of the method are not part of the signature. As signatures have to be unique within the same type, it is impossible to declare two methods that only differ in their return-values or parameter-names. Although out and ref parameter modifiers are part of the signature, two methods may not differ only in these values.

Constructors

Constructors can be either instance or static constructors. The syntax is similar to the method's one, but in contrast constructors lack of a return value.

3.2.3 Parameters

3.2.4 Attributes

Attributes are declarative elements that attach information to various program entities. They consist of a name and optional arguments. This information is accessible at runtime. In contrast to ordinary comments the attribute's syntax is checked at compile-time. Attributes cannot be assigned to every entity, but most of the entities, such as classes, interfaces, structs, enums, delegates, or function members, are supported. However, a namespace cannot be attached an annotation. After parsing the source code, every attached attribute is represented by a node of its own in the Abstract Syntax Tree (AST).

3.2.5 Accessibility Levels

3.2.6 Inheritance

3.3 Syntactic analysis

3.4 Data Model

```
1  CREATE TABLE [dbo].[SeedGenerator](
2     [SeedGenerator] [int] IDENTITY(1,1) NOT NULL
3  ) ON [PRIMARY]
```

Chapter 4

Tracking Changes

4.1 Annotation Language

As learning and utilizing a difficult and complex change description language is very cumbersome, it is advantageous to keep such a language as simple as possible. Few keywords in conjunction with computer-aided support lowers the previous knowledge and mostly supersedes user interaction. Anyway, change tracking shall not discover false positives and negatives. In place of adding change descriptions or previous states, like done in the manual approaches discussed in 2.2.3, only a global unique identifier is assigned to every program entity on the interface level:

```
[ID(987)]
public void createBill(Date invoiceDate)
```

Depending on the applied changes, the developer has to change the attribute type from `ID` to `MergeSequence`, `MergeSimilar` or `Split`. Unlike `ID`, the former two attributes require more than one parameter.

4.1.1 Attribute Keywords

The annotation language contains the keywords `ID`, `MergeSequence`, `MergeSimilar` and `Split`. In form of attributes they are annotated to program entities, such as reference types or members. The purpose of the `ID` attribute is only the identification of entities, whereas the other describe special changes to method bodies in a coarse grained way. Attribute keywords and refactoring names cannot be used interchangeable, because only the environment of each entity, such as its position in the inheritance hierarchy, in combination with the keywords assigned lead to an unambiguously identified refactoring.

ID

The `ID` attribute only has one mandatory parameter, a global ambiguous identifier which facilitates the recognition of the program entity it is assigned to. In the actual context, identifiers are always the starting point for change tracking. After each check in to the VCS, declarations on the interface level in conjunction with the change information in form of annotations are recorded. This

information is the basis for change tracking. To initialize a new position for further change tracking, the tracking tool replaces the attributes of those program entities, which contain a different annotation than `ID`, with a new unambiguous identifier. Subsequently, the changed source code is checked in by the tracking tool. Prior to subsequent code modifications, the developer has to check out this new initial state.

MergeSequence

`MergeSequence` introduces a new method, which merges the bodies of several methods in a sequential order. The statements of the former bodies may be adapted, but their order shall stay unchanged. In the strict sense, the bodies of the old methods are moved to a new one, and the previous `ID`s are appended to the parameter list of the new method's `MergeSequence` attribute. The order of the attribute's parameter list fixes the sequence, in which the bodies have been merged. Finally, the remaining parts of the old method, such as its signature and return type, have to be removed.

MergeSimilar

Methods with similar behavior are merged with use of the `MergeSimilar` attribute. Through this, duplicated methods can be removed. All prior methods are assumed to have the same signature. This attribute not necessarily represents a merge method refactoring. A pull up refactoring is likewise possible. To achieve an unambiguous identification, the method declaration additionally has to be taken into account. Section 4.2 depicts, on what terms a certain refactoring is tracked.

Split

4.1.2 Single and Multiple Changes

ToDo: Adapt text to subsection
Depending on the applied changes, the id remains the same or a new one is assigned. Unless methods are merged or split, the id stays unchanged. Otherwise, changes of two subsequent configurations are discoverable by comparing their entities of the same identifier. As transactions contain only changed versions, the problem can be reduced to a comparison of the previous active configuration with the ensuring transaction. Then, all changes are compared to the total set of former active versions. In order to minimize the user interaction, ids are computed and assigned computer-aided. A tool parses the very first version and adds ids to all entities. Ids are encapsulated into C# attributes. After parsing, their representing nodes are already assigned to the exact entity node. Comments, in contrast, lack of this assignment. Another advantage of annotations is the automatic syntax validation, which takes place every time the developer compiles his code. Even refactorings applied within different classes of the inheritance hierarchy are supported, as overwritten method of the same signature have an id of their own. Below, different refactorings are taken into account. They are exemplified by reinterpreting examples of Fowler's refactoring catalog [Fow99] in the context of annotation based change tracking.

4.2 Using of Annotations to Identify Refactorings

The following chapter covers chains of refactorings.

4.2.1 Rename Method

If the name of a method does not offer information about its intention, a rename method refactoring can be applied to change the method name to a better one.

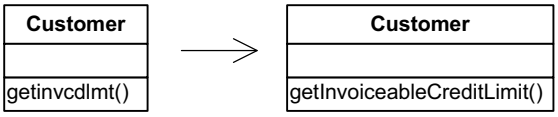

Figure 4.1: Rename method

Consider a method, which consists of the obfuscated name `getinvcdtlmt`. Rename method is used to replace the old name with `getInvoiceableCreditLimit`, a name, that is more comprehensible by humans (Figure 4.1). As the method is unambiguously identified by an identifier, only the names of two successive versions' names have to be compared. The identifier has been assigned to an annotation of the type `ID` or `MergeSimilar`. In the former case its value remains the same (Figure 4.2), whereas the latter leads to a new assigned `ID` due to the merge operation. `MergeSequence` and `Split` do not indicate a rename method refactoring, because they implicitly introduce one or more new methods.

```
1 [ID(1000)]
2 public double getinvcdlmt()
```

(a) Version n

```
1 [ID(1000)]
2 public double getInvoiceableCreditLimit()
```

(b) Version n + 1

Figure 4.2: Rename method - The method name is changed from `getinvcdlmt` (a) to `getInvoiceableCreditLimit` (b). Nevertheless, the identifier remains the same.

4.2.2 Merge Method

ToDo: Rewrite/adapt this subsection
Certainly associating global ids to entities is sufficient to automatically identify most of the low-level refactorings. Splitting or merging methods especially in chains of refactorings are hardly to identify. But which id is to be assigned to a merged method? As the merged method's body consists of the statements of

```
1 [ID(1000)]
2 public void setName()
3
4 [ID(10001)]
5 public void setSurname()
```

(a)

```
1 [MergeSequence(1000, 1001)]
2 public void setName()
```

(b)

```
1 [ID(1005)]
2 public void setName()
```

(c)

Figure 4.3: Merge methods - two methods with ids 1000 and 1001 (a) are merged to one method (b). After every check-in a tool replaces sequences of annotations through a new id (c)

both methods, their ids have to be assigned together with an merge-attribute. Even a sequence of merged methods is supported. To enhance the readability after every check-in, sequences of annotations are replaced by a new id (Figure 4.3).

4.2.3 Split Method

ToDo: Rewrite/adapt this subsection
Like merging splitting has an annotation of its own: split. Its argument contains a character indicating the part of the split method: a for the first part, b for the second and so on. In addition the old annotation with id remains in the code until it is checked-in and the sequence of annotations is replaced by one annotation with a new id.

4.2.4 Move Method

By applying a move method refactoring, a method is moved to another reference type, for example, because the features of the target are more often used by the method than the ones of the source.

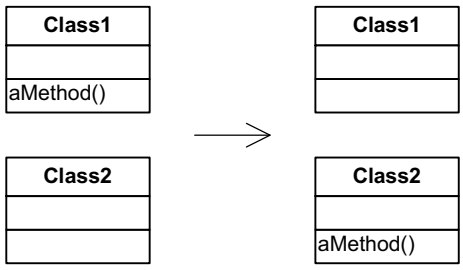

Figure 4.4: Move method

Nevertheless, on condition that the target type is part of the inheritance

hierarchy of the source type, in place of move method, a pull up or push down refactoring has been applied. A move method refactoring is characterized by the following conditions:

$$\begin{aligned} &\forall method_k \in Version_i \\ &\forall method_l \in Version_{i+1} \\ \exists hierarchyType_m \in SuperType(method_k) \cup SubType(method_k) : \\ &ID(method_k) = ID(method_l) \wedge \\ &Type(method_k) \neq Type(method_l) \wedge \\ &hierachyType_m \neq Type(method_l) \end{aligned}$$

4.2.5 Pull Up/Push Down Method

$$\begin{aligned} &\forall method_k \in Version_i \\ &\forall method_l \in Version_{i+1} \\ \exists superType_m \in SuperType(method_k) : \\ &ID(method_k) = ID(method_l) \wedge \\ &Type(method_k) \neq Type(method_l) \wedge \\ &superType_m = Type(method_l) \end{aligned}$$

Chapter 5

Evaluation

User has to add semantic informations Provide Refactorings-proposal Furthermore are all Refactorings selectable

Chapter 6

Related Work

Heuristics and Metrics

Chapter 7

Future Work

.

Chapter 8

Conclusion

Bibliography

[Bro97] Andrei Z. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, page 21, Washington, DC, USA, 1997. IEEE Computer Society.

[CN96] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 359–368, Washington, DC, USA, 1996. IEEE Computer Society.

[Cor01] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[CVS] Concurrent Versions System.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson:. Automated detection of refactorings in evolving components. In *Proceedings of European Conference on OO Programming (ECOOP'06)*, Nantes, France, 2006.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM Press.

[DJ06] Danny Dig and Ralph Johnson. Toward automatic upgrade of component-based applications, May 2006.

[ECL] Eclipse.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Longman, Inc., 1999.

[GW05] Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE*, pages 274–283, 2005.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[RH02] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework dependent applications, 2002.

[SVN] Subversion.

[TNMiM05] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Java birthmarks —detecting the software theft —. *IEICE Transactions on Information and Systems*, E88-D(9):2148–2158, September 2005.

[VSS] Visual SourceSafe.

[ZW04] Thomas Zimmermann and Peter Weigerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 1, 2006