MolhadoRef: A Refactoring-aware Infrastructure for OO Programs

Danny Dig, Kashif Manzoor
University of Illinois at Urbana-Champaign
{dig,manzoor2}@uiuc.edu

Tien N. Nguyen
Iowa State University
tien@iastate.edu

Ralph Johnson
University of Illinois at Urbana-Champaign
johnson@cs.uiuc.edu

ABSTRACT

Refactoring tools allow programmers to change source code much quicker than before. However, the complexity of these changes cause versioning tools that operate at a file level to lose the history of components. This problem can be solved by semantic, operation-based SCM with persistent IDs. We propose that versioning tools be aware of the program entities and the refactoring operations. MolhadoRef uses these techniques to ensure that it never loses history.

1. INTRODUCTION

Refactorings [12] are program transformations that change the structure of a program without changing its external behavior. In recent years, tools like Eclipse [7] and IntelliJ IDEA [13] have made refactoring tools a standard for most Java programmers. The wide-spread use of a new kind of software tool often forces other tools to adjust to it. Refactoring tools make particular demands on software configuration management (SCM) tools. A refactoring tool allows a programmer to quickly make changes that potentially affect all parts of a system. Some refactorings are local in scope, such as extracting a function. However, changing the name or interface of a global function can have global scope, since every part of the system that uses the function will have to change. Changes that seem simple from a refactoring point of view can be complex from an SCM point of view unless the SCM tools can treat refactorings intelligently.

Most SCM systems are based on files, not on program entities. They model changes in terms of the lines of a file that have changed, instead of program entities that changed. In contrast, a semantics-based SCM is tailored to a particular programming language and so can represent individual program entities such as classes and functions. Further, most SCM systems identify entities by name, whether it is file name or the name of a program entity. Renaming or moving an entity will cause the system to lose track of it. In contrast, Molhado [19] is an SCM infrastructure that creates unique, persistent IDs for each entity, and treats the name as an attribute that can change. Thus, it can track the history of an entity in spite of it being refactored.

An SCM system needs to deal with branches; versions derived from a common base but not from each other. Making branches is easy, but merging them can be hard. File-based SCM systems can merge changes automatically if they are to different parts of a file, but if two branches change the same part of a file then the merge fails and must be done manually. If each branch renames a function and there is a line in the program that calls both functions then a conventional SCM system cannot merge the changes automatically. This demonstration will show that those changes can be merged by a semantics-based SCM that gives program entities permanent IDs. It helps track the history of refactored, fine-grained program entities, eliminates many conflicts when merging refactored versions in multi-user environments, and represents the history at a higher level.

2. OUR APPROACH

2.1 Semantics-based, Operation-based SCM

We developed *MolhadoRef*, a semantics-based SCM system, which is able to capture and version the underlying semantics of Java programs. It also maintains *persistent identifiers* for all program entities in its repository. It uses the *operation-based* SCM approach [15] to represent and record refactoring operations as *first-class* entities in the repository. In the operation-based approach, an SCM tool records the operations that were performed to transform one version into another and replays them when updating to that version. The operation-based approach gives a precise way to integrate changes caused by editing operations from different lines of parallel development [15]. As for refactoring, recent extensions to refactoring engines [12, 7] allow to record and replay refactoring operations.

MolhadoRef is based on Molhado object-oriented SCM infrastructure [19], which was developed for creating SCM tools. Unlike the file-based SCM approach, Molhado allows an SCM system to model and capture the structure of logical entities within a file and the operations on them. Molhado has a flexible data model that allows it to represent programs in any language. A program consists of a set of *nodes*, each of which has a set of *slots* that are attached to it by means of *attributes*. Nodes are the units of identity, while slots hold values and attributes map nodes to slots. Nodes, slots and attributes that are related to each other form *attribute tables*, whose rows correspond to nodes and columns to attributes. The cells of attribute tables are slots.

The Molhado data model can be specialized for a particular application. MolhadoRef specializes it to represent Java programs, so nodes are used to represent program entities. The unique identifiers of nodes facilitate the management of entities' histories, especially when they are refactored. MolhadoRef captures the semantics of a program with a Molhado component type named *CompilationUnit*, which has a tree-based structure representing the program's Abstract

Syntax Tree (AST). An AST node is represented as a Molhado node. The class name is one of its properties.

MolhadoRef assumes that the front-end editor recognizes refactoring operations that were performed on program entities. When a user checks in changes to the current version, refactoring operations are recorded along with other changes. Refactoring operations are represented as Molhado components and recorded as attribute values associated with AST nodes. Parameters of an operation are recorded as attribute values associated with the operation.

Version control is added into the data model by a third dimension in attribute tables. That is, slots in any data type can be versioned. Molhado's version model is called *product versioning* in which a version is *global* across entire system [19]. It allows branching and merging of versions as well. Molhado stores and retrieves versioned attribute tables. No file versioning is involved. A novel structure-oriented versioning algorithm for attributed trees was developed to provide the fine-grained content change and version management for ASTs.

2.2 Composition of ID-based Refactorings

Most refactoring engines assume a single user. Refactoring engines make it easy to change a lot of code, but when multiple developers refactor the same code, it is likely that they will create conflicts. The refactorings that one user performed during a programming session might be perfect alone, but are invalid when they are combined with the refactorings that another user performed on the same code. Current refactoring engines are purely based on the names of the source code entities, therefore we call them *name-based refactorings*. Even though in a single user environment name-based refactorings work fine, they fail in environments where multiple users refactor the source code in parallel. Therefore, a theory for composing refactorings is needed to accommodate multi-user environments.

To overcome the shortcomings of *name-based refactoring*, we propose an extension to the refactoring engines called *ID-based refactoring*. We were inspired from the real life of human beings. The citizens in most countries hold a unique ID (e.g., in U.S. this is the Social Security Number) that allows the person's identity to remain the same even though the person might change her name (e.g. through marriage) or relocate to a different part of the country. We assign to each source code entity a unique ID which remains the same even when the entity is refactored. New IDs get created when new source code entities are added to the program, and IDs get deleted when their corresponding entities get deleted. IDs are stored in the SCM system along with the source code entities when the source code is checked in.

The presence of persistent IDs can solve several types of conflicts in multi-user environments that are unsolvable within the *name-based refactoring* paradigm. More details on our ID-based refactoring composition theory are in [5].

3. TOOL IMPLEMENTATION

Programming tools are more likely to be used in practice when they are conveniently incorporated into an Integrated Development Environment (IDE). We implemented a semantic, operation-based SCM as an Eclipse plugin, MolhadoRef. MolhadoRef uses the Eclipse Java programming editor as the front end and the Molhado framework as the SCM back end. MolhadoRef connects two systems that work in different paradigms. Eclipse editors operate at the file level granularity. Molhado models source code entities at a finer level of granularity than file-based systems. Also, Eclipse offers a name-based refactoring engine whereas MolhadoRef requires an ID-based refactoring engine.

First, we extended the Molhado framework with support for the program elements found in Java programs as described earlier. Molhado offers two types of components: *composite* components that can contain other composites and *atomic* components (the lowest level of granularity). Java packages and compilation units (Java source files) are modeled as Molhado composite components. Program elements within a Java class (e.g., methods, fields) are modeled as atomic components. Although Molhado affords modeling at even finer level of granularity (e.g., program statements and expressions), for efficiency reasons we stop at the method and field declaration level. The name and signature of the method are attributes that allow for distinguishing between possible overloaded methods in the same Java class. Along with methods and fields, inner classes are modeled as entries in the table associated with the main class of a compilation unit; inner classes can expand into new tables when they contain fields and methods.

The interaction with the Eclipse front-end is triggered when a user wants to check in the code. The first time an Eclipse project is checked into in the repository, MolhadoRef does a *lightweight* parsing of the source code. We call it lightweight parsing because it stops at the level of method and field declaration; the parser stores the program statements and expressions within the method bodies or field initializers as a single string. For each Java program entity, MolhadoRef creates its Molhado counterpart. These Molhado entities are connected to form trees mirroring the lightweight Java ASTs. Java source files contain other information like copyright notice and documentation embedded in javadoc comments. Even though technically the documentation is not part of the compiler's AST nodes, we save this information as "documentation" attributes of the corresponding Molhado entities.

After code is checked in for the first time, subsequent 'check-in's need to store only the changes from last check-in. In a pure operation-based SCM, all the changes are recorded at the time when they happen and are stored as operations in the SCM system. These operations are then replayed on the source code of a user who wants to update to the latest version of the code. This operation-based approach can be very accurate in recording the exact type of change, but a large number of change operations introduce overhead both for recording and replaying. At the other end of the spectrum, in the state-based approach, the deltas are computed just before the user commits the code by comparing the two versions. This is more efficient (since the changes are computed only once per programming session) but it cannot recover the semantics of the changes (it gathers all changes in a large pile of seemingly unrelated changes). For instance, a method rename can result in a lot of changes: changing the declaration of the method, updating the method callers as well as the transitive closure of all declarations and call sites of overridden methods.

MolhadoRef uses a mixture of both paradigms to maximize efficiency and accuracy. MolhadoRef uses the Eclipse compare engine to learn the individual deltas (e.g., changes within a method body or addition/removal of classes, meth-

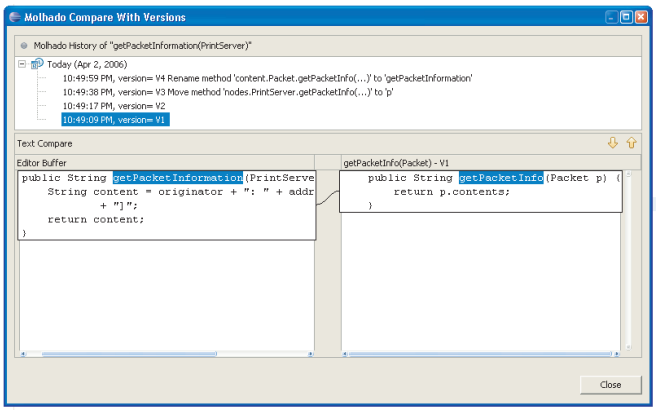

Figure 1: History of Refactored and Modified Entity

ods, and fields) and it captures the refactorings performed by the Eclipse refactoring engine to record the semantics of refactoring operations. The changes caused by refactoring operations are reported by both the compare engine and the refactoring engine. For instance, renaming method `LANPacket.getPacketInfo` to `getPacketInformation`, causes the compare engine to report deletion of `getPacketInfo` and addition of `getPacketInformation`. However, since the refactoring engine also reports the renaming, the change reported by the compare engine is overruled, and the name of the method is updated in tables, thus avoiding loss of history.

The Eclipse compare engine offers several APIs for reporting changes at different levels of granularity. MolhadoRef uses `Differencer` to find changes at the directory or file level. Once it learns the Java files that changed, it uses `JavaStructureComparator` to report the changes in terms of Java program elements (e.g., classes, methods and fields). From the program elements, the `RangeDifferencer` finds the low level changes (e.g., changes inside a method body). All the differencers report their results as a tree of `DiffNodes`, which serve as inputs to `JavaStructureDiffViewer` that displays graphically the changed elements.

The Eclipse refactoring engine was recently extended (starting with Eclipse 3.2M4 of December 2005) to record refactoring operations. MolhadoRef uses the new refactoring engine to record and store the performed refactorings. The representation for refactorings in MolhadoRef, which is based on attribute tables as described, is compatible with the XML format that Eclipse refactoring engine uses. Therefore, the refactoring operations can be resuscitated and replayed back by the refactoring engine during an update operation.

When the user invokes a checkout operations, MolhadoRef reconstructs (from its internal representation) the Java compilation units and packages and invokes the Eclipse code formatter on the files. After MolhadRef brings the classes and packages into a project in the current Eclipse workspace, the user can resume her programming session.

The snapshot in Figure 1 illustrates the history of a renamed method. The reader can find more screen shots and download MolhadoRef at: `netfiles.uiuc.edu/dig/MolhadoRef`

4. EVALUATION

Modeling source code entities and refactoring operations requires extra space when compared with file-based SCM systems. We compare the space required by MolhadoRef with the space used by CVS to keep track of source code.

Table 1: Evolution of Eclipse's core.refactoring

Version	LOC	Changed LOC	#Pack	#Classes	#Methods
01/31	19933	-	14	114	868
02/28	19993	1786	13	114	871
03/29	20405	526	13	114	875

We checked out of Eclipse CVS repository three revisions of `org.eclipse.ltk.core.refactoring`. This subcomponent is the core of the refactoring engine in Eclipse. These revisions are tagged in the Eclipse repository at 01/31, 02/28/ and 03/29 2006. Table 1 shows how the source code evolved along this time interval. Even though the total number of lines of code does not reveal a great number of changes, the component passed through a great deal of changes revealed by the number of individual lines of code changed. Between versions 01/31 and 02/28, our refactoring-inference engine reveals several structural changes: four classes moved to other packages, one class was renamed, five classes were deleted and five new unrelated classes were added, four methods were renamed and four changed their signatures, one method moved to another class. Between versions 02/28 and 03/29, most changes are edits, e.g., all the classes changed their copyright notice.

Table 2 shows the space taken by the source code relying on the local disk, compared with the space taken by CVS and Molhado. We used the Unix 'disk usage' (du) utility to calculate the total space. We give the size in bytes (as returned by 'du -abc') and in kilobytes ('du -akc'). The Linux machine uses a block size of 1kB for files and 4kB for directories. To calculate the space taken by CVS we checked into our own CVS server the three versions of the source code. As for Molhado, we checked in the three versions. After each check-in we executed a check-out. We used Eclipse comparison utility to verify that our Molhado implementation did not lose/add any source code along the three revisions.

The size in bytes for MolhadoRef degrades gracefully; it is respectively 2.21, 2.69, and 2.80 times larger than the size of initial source code. However, Molhado adds a large number of small files (between 20 and 80 bytes) to represent internally the versioning information. Since the operating system on the machine where we ran the evaluation allocates 1kB for any of these small files, the ratio between the actual files size used to store the source code in Molhado and the initial source code is respectively 3.55, 5.52, and 7.04. On a Windows system that allocated less space for small files, the space on disk for MolhadoRef repository was 2.39, 3.23, and 3.66 times larger than the original source code.

However, given the current trend that disk space is getting cheaper, we believe that the benefits gained by being able to track structural changes far outweigh the extra space.

MolhadoRef correctly retrieves the history of classes and methods renamed or moved in the 02/28 version, while CVS loses their history. In addition, browsing through the history with MolhadoRef reveals the refactoring operations, thus offering a higher-level understanding of code evolution. CVS shows a lot of changes scattered throughout the source code, with no connection between them.

In the future, when open-source projects will store logs of refactorings performed by different developers, we plan to estimate how many refactorings could be merged within

Table 2: Comparison among spaces required for storing the source code on local disk (no version control), CVS and Molhado. For each system, space size in bytes (B) is the sum of all bytes in every file, while the space used by the operating system to store the files is given in kiloBytes(kB).

Version	Local[B]	Local[kB]	CVS[B]	CVS[kB]	Molhado[B]	Molhado[kB]
01/31	722596	984	724717	968	1602252	3500
02/28	718026	968	832425	1104	1934408	5344
03/29	735525	988	869721	1136	2063352	6964

an ID-based environment. We plan to implement a merging algorithm based on the conflict tables and evaluate what is the time saved when merging with our tool. Since there are currently no logs of refactorings for open-source projects, this must wait for the future.

Checking in the source code for the three versions using MolhadoRef took respectively 11, 9 and 7 seconds, while checking out each version took respectively 10, 14, and 14 seconds (check out time includes the compilation of the whole Eclipse project). Using CVS, check-ins took respectively 2, 3, and 2 seconds and each check-out took 8 seconds.

5. RELATED WORK

5.1 SCM Systems

Many sources can be served as excellent surveys on SCM [4, 27]. Early SCM systems (e.g. CVS [18]) provided versioning support for individual files and directories. In addition to version control, advanced SCM systems also provide more powerful configuration management services. Subversion [24] provides more powerful features such as versioning for meta-data, properties of files, renamed or copied files/directories, and cheaper version branching. Similarly, commercial SCM tools still focus on files [27].

Several SCM systems have recognized the importance of managing the version history of program entities. Similar to our approach, Gandalf [11] works at the AST level. In Gandalf, each module has a unique interface and potentially multiple realization variants, each of which evolves into versions. In DAMOKLES [6], which is heavily based EER database, leaves of the object composition hierarchy may be as small as statements. POEM [14] provides version control in terms of functions and classes in C++ programs, which are interrelated via a dependency graph that is partitioned into work areas. The unique identifiers in MolhadoRef is similar in spirit to unique tags for AST nodes in Westfechtel's system [26]. In that system, tags are used in incremental updating of revisions of dependent documents.

The tree-based versioning framework in COOP/Orm [16] works not only for programs but also for hierarchically structured documents. The principles of the framework include sharing unchanged nodes among versions and change propagation. The Unified Extensional Versioning Model [1] supports fine-grained versioning for a tree-structured document by composite, atomic, and link nodes. Each atomic node is represented via a textual file. Since their focus is on collaborative and interactive editing tasks, fine-grained changes are not persistent. In Coven [3], the exact size of a versioned fragment depends on the supported document: entire method and field declaration for C++ and Java programs, or paragraph of text in LaTeX documents. A project in Coven is a composition of *compound artifacts*, which are sets of other artifacts and/or program fragments. In Ohst's fine-grained SCM model [20], changes are managed within contexts of UML *tools* and *design transactions*.

The operation-based approach has been used in software merging [15]. It is a particular flavor of change-based merging that models changes as explicit operations or transformations. Operation-based merge approach can improve conflict detection [17]. Lippe *et al* [15] described a theoretical framework for conflict detection with respect to general transformations. No concrete application or tool for refactorings was presented. Edwards' operation-based framework [8] detects and resolves semantic conflicts from application supplied semantics of operations. However, no existing SCM tool is able to manage versions of fine-grained program entities and refactoring operations performed on those entities in a tightly connected manner as in MolhadoRef.

5.2 Refactoring

Programmers have been cleaning up their code for decades, though the term *refactoring* was coined much later [22]. Opdyke [21] wrote the first catalog of refactorings while Roberts and Brant [23] were the first to implement a refactoring engine. The refactoring field gained much popularity with the catalog of refactorings written by Fowler et al. [10]. Soon after this, IDEs began to incorporate refactoring engines. Tokuda and Batory [25] describe the large architectural changes in two frameworks as a large sequence of small refactorings. They estimate that automated refactorings are 10 times quicker to perform than manual ones. Record-and-replay of refactorings was recently demonstrated in Catch-Up [12] and JBuilder2005 [2] and is planned to be a standard part of Eclipse 3.2. Our methodology uses the record-and-replay of refactorings, although there are many more components needed to build a refactoring-aware SCM.

Ekman and Asklund [9] insightfully present the benefits of refactoring-aware versioning systems: the ability to track the history of refactored program entities, better merging in the presence of well defined semantics of refactoring operations, and better human understanding of the code evolution. They too present a programming model that affords refactoring-aware versioning system for Eclipse. Our approaches are different in many ways: their model heavily relies on modifications to the Eclipse *front-end* (e.g., changing the Eclipse Java Model class hierarchy to support IDs for program elements), whereas we rely on a powerful *back-end* SCM to model program entities with unique IDs. Since we do not impose any changes to the development environment, our approach can be smoothly integrated with other IDEs (in fact we had another implementation with a standalone front-end editor). Their approach is more lightweight since it keeps the program elements and their IDs in volatile memory, thus allowing for a short-lived history of refactored program entities. Our approach is more heavyweight, program elements and their IDs are modeled in the SCM and stored throughout the lifecycle of the software project allowing for a global history tracking of refactored entities.

6. CONCLUSION & FUTURE WORK

Refactoring tools have become popular because they allow programmers to safely make structural changes in large systems. However, such changes create problems for the current SCM tools that operate at the file level. As a result, the history of refactored entities is lost. We propose a novel SCM system, MolhadoRef, that is aware of program entities and the refactoring operations that change them. Because MolhadoRef uses a unique identifier for each program element, it can track the history of refactored program elements. In addition, we introduce the notion of *ID-based refactoring* and we show how unique identifiers allow for many more merging scenarios in multiuser environments than traditional name-based refactoring.

We implemented a refactoring-aware SCM as MolhadoRef, an Eclipse plugin. We extended the Molhado framework to model Java program elements and store refactoring information. By evaluating the extra space required to model program elements and refactoring operations, we learned that the extra space is around three times larger than the original source code. We believe the benefits of tracking refactored entities far outweigh the extra space cost.

In the future, we plan to focus on algorithms for semantic merging of refactoring operations and regular edit operations, so that the level of user involvement is minimal. Because we operate at the semantical level of changes, the merging is going to be much more powerful than traditional line-based merging.

We believe that the availability of such semantics-aware, refactoring-tolerant SCM tools is going to encourage programmers to be even bolder when refactoring. Without the fear that refactorings are going to cause conflicts with others' changes, software developers will have the freedom to make their designs easier to understand and reuse.

7. REFERENCES

[1] U. Asklund, L. Bendix, H. Christensen, and B. Magnusson. The unified extensional versioning model. In *Proceedings of the 9th Software Configuration Management Workshop*. Springer Verlag, 1999.

[2] www.borland.com/resources/en/pdf/white_papers/jb2005_whats_new.pdf.

[3] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of the tenth Foundations of software engineering symposium*, pages 99–108. ACM Press, 2002.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[5] D. Dig, T. Nguyen, and R. Johnson. Refactoring-aware software configuration management. Technical Report UIUCDCS-R-2006-2710, UIUC, April 2006.

[6] K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES: a database sytem for software engineering environments. In *Proceedings of the International Workshop on Advanced Programming Environments*. Springer Verlag, 1986.

[7] Eclipse Foundation. http://eclipse.org.

[8] W. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of Symposium User Interface Software Technology*, 1997.

[9] T. Ekman and U. Asklund. Refactoring-aware versioning in eclipse. *Electr. Notes Theor. Comput. Sci.*, 107:57–69, 2004.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[11] N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, Dec 1986.

[12] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 274–283, 2005.

[13] JetBrains Corp. http://www.jetbrains.com/idea.

[14] Y. Lin and S. Reiss. Configuration management with logical structures. In *ICSE'96: Proceedings of International Conference on Software Engineering*, pages 298–307, 1996.

[15] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE5: Proceedings of Symposium on Software Development Environments*, pages 78–87. ACM Press, 1992.

[16] B. Magnusson and U. Asklund. Fine-grained revision control of Configurations in COOP/Orm. In *Proceedings of the 6th Software Configuration Management Workshop*. Springer Verlag, 1996.

[17] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[18] T. Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[19] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 215–224. ACM Press, 2005.

[20] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *FSE'03: Proceedings of the Foundations of software engineering*, pages 227–236. ACM Press, 2003.

[21] B. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, U of Illinois at Urbana-Champaign, 1992.

[22] B. Opdyke and R. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA'90: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[23] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *TAPOS*, 3(4):253–263, 1997.

[24] Subversion.tigris.org. http://subversion.tigris.org/.

[25] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[26] B. Westfechtel. Revision Control in an Integrated Software Development Environment. In *Proceedings of the 2nd Software Configuration Management Workshop*, pages 96–105. ACM Press, 1989.

[27] CM Yellow Pages. http://www.cmcrossroads.com/.