Source Code Rejuvenation is not Refactoring

Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup

Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77843-3112
{peter.pirkelbauer, dechev}@tamu.edu bs@cse.tamu.edu

Abstract. Programmers rely on programming idioms, design patterns, and workaround techniques to make up for missing programming language support. Evolving languages often address frequently encountered problems by adding language and library support to subsequent releases. By using new features, programmers can express their intent more directly. As new concerns, such as parallelism or security, arise, early idioms and language facilities can become serious liabilities. Modern code sometimes benefits from optimization techniques not feasible for code that uses less expressive constructs. Manual source code migration is expensive, time-consuming, and prone to errors.
In this paper, we present the notion of *source code rejuvenation*, the automated migration of legacy code and very briefly mention the tools we use to achieve that. While *refactoring* improves structurally inadequate source code, source code rejuvenation leverages enhanced program language and library facilities by finding and replacing coding patterns that can be expressed through higher-level software abstractions. Raising the level of abstraction benefits software maintainability, security, and performance.

1 Introduction

Popular programming languages evolve over time. One driver of evolution is a desire to simplify the use of these languages in real life projects. For example, in the early 1970ies the C programming language was developed as a system programming language for the UNIX operating system on the PDP-11 [1]. With the proliferation of UNIX to other platforms, C evolved to reflect concerns such as portability and type safety. Later Stroustrup enhanced C with higher level abstractions to simplify the development of a distributed and modularized UNIX kernel [2]. Compared to C, ISO C++ [3] directly supports the program design with classes, dynamic dispatch, templates, exception handling, and more [2]. Abstraction mechanisms present in C++ can be compiled to efficient machine code on many architectures. This makes C++ suitable for software development of embedded systems, desktop computers, and mainframe architectures. C++'s proliferation and success is a constant source of ideas for enhancements and extensions. The ISO C++ standards committee has released a draft of the next revision of the C++ standard, commonly referred to as C++0x [4] [5]. C++0x will address a number of modeling problems (e.g., object initialization) by providing better language and library support. Until compiler and library implementations of C++0x become widely available, C++ programmers solve these problems through programming idioms and workaround techniques. These "solutions" are typically more involved than they would be in C++0x and can easily become another source of errors and maintenance problems.

This paper draws its examples from C++0x's proposed extensions to the language and its standardized libraries [6], but the topic is equally valid for other widely used and evolving languages, such as Python [7], C# [8], and Java [9]. For example, Java is currently undergoing its sixth major revision since its first release in 1995. Extensions under consideration for Java 7 include support for closures, null safe method invocations, extended catch clauses to catch and rethrow groups of exceptions, type inference for generics and others [10].
The contributions of this paper are:

- We define the term *source code rejuvenation* and delineate it from related fields.
- We demonstrate source code rejuvenation with examples that migrate code from C++03 to C++0x.

The rest of this paper is outlined as follows: In §2, we define the term source code rejuvenation. In §3, we demonstrate source code rejuvenation based on C++0x language features. In §4, we put our work in context to refactoring. In §5, we describe the tool we use to implement source code rejuvenation tools. In §6, we summarize.

2 What is Source Code Rejuvenation?

Source code rejuvenation is a source-to-source transformation that replaces deprecated language features and idioms with modern code. Old code typically contains outdated idioms, elaborate and often complex coding patterns, deprecated language features (or data structures). The rejuvenated code is more concise, safer, and uses higher level abstractions. What we call outdated idioms (and patterns) are techniques often developed in response to the lack of direct language support. When programming languages and techniques evolve, these coding styles become legacy code, as programmers will express new code in terms of new language features. This leads to a mix of coding styles, which complicates a programmer's understanding of source code and can cause maintenance problems. Furthermore, the teaching and learning can be greatly simplified by eliminating outdated language features and idioms.

Source code rejuvenation is a *unidirectional process* that detects coding techniques expressed in terms of lower-level language and converts them into code using higher-level abstractions. High-level abstractions make information explicit to programmers and compilers that would otherwise remain buried in more involved code. We aim to automate many forms of code rejuvenation and to provide program assistance for cases where human intervention is necessary. In other words, our aim is nothing less than to reverse (some forms of) (software) entropy!

Preserving behavioral equivalence between code transformations is necessary to claim correctness. In the context of source code rejuvenation, a strict interpretation of behavior preservation would disallow meaningful transformations (e.g., see the initializer list example §3.1). We therefore argue that a valid source code rejuvenation preserves or improves a program's behavior. In addition, when a rejuvenation tool detects a potential problem but does not have sufficient information to gurantee a correct code transformation, it can point the programmer to potential trouble spots and suggest rejuvenation. For example, a tool can propose the use of the C++0x's array class instead of C style arrays. A C++0x array object passed as function argument does not decay to a pointer. The argument retains its size information, which allows a rejuvenation tool to suggest bounds checking of data accesses in functions that take arrays as parameters.

2.1 Applications

Source code rejuvenation is an enabling technology and tool support for source code rejuvenation leverages the new languages capabilities in several aspects:

Source Code Migration: Upgrading to the next iteration of a language can invalidate existing code. For example, a language can choose to improve static type safety by tightening the type checking rules. As result formerly valid code produces pesty error or warning messages. An example is Java's introduction of generics. Starting with Java 5 the compiler warns about the unparametrized use of Java's container classes.

Even with source code remaining valid, automated source code migration makes the transition to new language versions smoother. For example, programmers would not need to understand and maintain source files that use various workaround techniques instead of (later added) language constructs. For example, a project might use template based libraries (e.g., Standard Template Library (STL) [11], STAPL [12]) where some were developed for C++03 and others for C++0x. In such a situation, programmers are required to understand both.

Education: Integration with a smart IDE enables "live" suggestions that can replace workarounds/idioms with new language constructs, thereby educating programmers on how to better use available language and library constructs.

Optimization: The detection of workarounds and idioms can contribute a significant factor to both the development cost of a compiler and the runtime, as the detection and transformation requires time. Compiler vendors are often reluctant to add optimizations for every single scenario. The introduction of new language constructs can enable more and better optimizations (e.g., const_expr lets the compiler evaluate expressions at compile time [5]). Automated source code migration that performs a one-time source code transformation to utilize the new language support enables optimizations that might be forgone otherwise.

3 Case Studies

In this section, we demonstrate source code rejuvenation with examples taken from C++0x, namely initializer lists and concept extraction.

3.1 Initializer lists

In current C++, the initialization of a container (or any other object) with an arbitrary number of different values is cumbersome. When needed, programmers deal with the problem by employing different initialization idioms.

Consider initializing a vector of **int** with three constant elements (e.g., 1, 2, 3). Techniques to achieve this include writing three consecutive push_back operations, and copying constants from an array of **int**. We can "initialize" through a series of push_back()s:

```
// using namespace std;
vector<int> vec;

// three consecutive push_backs
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
```

Alternatively, we can initialize an array and use that to initialize the vector:

```
// copying from an array
int a[] = {1, 2, 3};
vector<int> vec(a,a+sizeof(a)/sizeof(int));
```

These are just the two simplest examples of such workarounds observed in real code. Although the described initialization techniques look trivial, it is easy to accidentally write erroneous or non-optimal code. For example, in the first example the vector resizes its internal data structure whenever the allocated memory capacity is insufficient to store a new value; in some situations that may be a performance problem. The second example is simply a technique that people often get wrong (e.g. by using the wrong array type or by specifying the wrong size for the vector). Other workarounds tend to be longer, more complicated, and more-error prone. Rejuvenating the code to use C++0x's initializer list construction [13] automatically remedies this problem.

```
// rejuvenated source code in C++0x
vector<int> vec = {1, 2, 3};
```

In C++0x, the list of values (1, 2, 3) becomes an initializer list. Initializer list constructors take the list of values as argument and construct the initial object state. As a result, the rejuvenated source code is more concise – needs only one line of code (LOC) when compared to two and four LOC needed by the workaround techniques. The rejuvenated source code becomes more uniform: every workaround is replaced by the same construct. In this particular case, we gain the additional effect that the rejuvenated code code style is analogous to C style array initialization (compare the definition of the array a in the code snippet with the workaround examples). Thus, the reader does not have to wonder about the irregularities of C++98 initialization.

3.2 Partial order of templated functions in a generic function family

Current C++ supports generic programming with its template mechanism. Templates are a compile time mechanism that parametrize functions or classes over types. With current C++, the requirements that make the instantiation of template bodies succeed cannot be explicitly stated. To type check a template, the compiler needs to instantiate the template body with concrete types. Programmers have to look up the requirements in documentation or infer them from the template body. Attempts to instantiate templates with types that do not meet all requirements fail with often hard to comprehend error messages [14]. Concepts [14] [15] is a mechanism designed for C++0x to make these requirements explicit in source code. A concept constrains one or more template arguments and provides for the separation of template type checking from template instantiation.

Concept extraction: In [16], we discuss tool support for extracting syntactic concept requirements from templated source code. For example, consider the STL [11] function advance that is defined over input-iterator:

```
template <class Iter, class Dist>
void advance(Iter& iterator, Dist dist) {
  while (dist−−) ++iterator;
}
```

Our tool extracts the following concept requirements:

```
concept AdvInputIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&); // to copy construct arguments
  void operator++(Iter&); // advance the iterator by one
  bool operator−−(Dist&, int); // decrement the distance
}
```

Likewise, our tool extracts the following requirements from the advance implementations for bidirectional-iterators:

```
// for Bidirectional−Iterators
template <class Iter, class Dist>
void advance(Iter& iterator, Dist dist) {
  if (dist > 0)
    while (dist−−) ++iterator;
  else
    while (dist++) −−iterator;
}

concept AdvBidirectIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&); // move the iterator forward
  void operator−−(Iter&); // move the iterator backward
  bool operator++(Dist&, int); // post-increment
  bool operator--(Dist&, int); // post-decrement
}
```

and random access-iterators:

```
// for RandomAccess-Iterators
template<class RandomAccessIterator, class Distance>
void advance(RandomAccessIterator& i, Distance n) {
  i += n;
}

concept AdvRandomAccessIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator+=(Iter&, Dist&); // constant time positioning operation
}
```

Callers of a generic function, such as the advance family, are required to incorporate the minimal concept requirements in its concept specification. Consider a function random_elem, that moves the iterator to a random position and returns the underlying value:

```
template <class Iter>
typename Iter::value_type random_elem(Iter iter, size_t maxdist) {
  advance(iter, rand(maxdist));
  return *iter;
}
```

The concept requirements on Iter depend on the minimal concept requirements of the generic function advance. From the concept requirements that were extracted for the advance family, the hierarchical concept relationship (or a base implementation) cannot be inferred. The sets of requirements extracted for input- and bidirectional-iterator can be ordered by inclusion. However, the set of requirements for randomaccess-iterator is disjoint from the other two sets. The minimal set of requirements cannot be automatically determined.

The lack of explicit information on the hierarchical order of templated function declarations is not only a problem for a concept extraction tool, but also for programmers. Without more information compilers cannot discern overloaded template functions for a given set of argument types. To overcome this problem, programmers have invented idioms, such as tag dispatching [17] and techniques that utilize the substitution failure is not an error mechanism [18]. This section of the paper demonstrates that a rejuvenation tool can recover the concept hierarchy by identifying the tag dispatching idiom in legacy code.

Tag dispatching: The tag dispatching idiom adds an unnamed non template parameter to the signature of each function-template in a generic function family (e.g., inputiterator_tag, bidirectional_iterator_tag, ...).

```
template<class InputIterator, class Distance>
void advance(InputIterator& iter, Distance dist, inputiterator_tag);

template<class RandomAccessIterator, class Distance>
void advance(RandomAccessIterator& iter, Distance dist, randomaccess_iterator_tag);
```

With the extra argument, the compiler can discriminate the tagged functions based on the non template argument dependent parameter type. A templated access function uses the class family iterator_traits [19] to construct an object of the proper tag type. An iterator tag is an associated type (i.e., iterator_category) of the actual iterator (or its iterator_traits).

```
template<class InputIterator, class Distance>
void advance(InputIterator& iter, Distance dist) {
  advance(i, dist, iterator_traits<InputIterator>::iterator_category());
}
```

Recovering structural information from tags: To distinguish tags from regular classes, we require parameters used as tags to possess the following properties.

– all template functions of a generic function family have unnamed parameter(s) at the same argument positions(s).
– the type tuple of tag parameters is unique for each function template within a generic function family.

In addition, we require that for each generic function family exist an access function that has the same number of non-tag arguments (and types). Tag classes are not allowed to have non-static members.

By identifying tag classes, our tool can deduce the refinement relationship of template functions from the inheritance relationship of the tag classes. Consider, the hierarchy of the iterator classes:

```
struct input_iterator_tag {};
struct forward_iterator_tag : input_iterator_tag {};
struct bidirectional_iterator_tag : forward_iterator_tag {};
struct randomaccess_tag : bidirectional_iterator_tag {};
```

By knowing that input_iterator is the base of the tag hierarchy, we can propagate the requirements of the corresponding template function advance to the requirements of its callers. For example, the requirements of function random_elem are:

```
concept RandomElem <typename Iter, typename Dist> {
  // requirements propagated from advance
  Dist::Dist(const Dist&); // to copy construct arguments
  void operator++(Iter&); // advance the iterator by one
  bool operator−−(Dist&, int); // decrement the distance

  // additional requirements from random_elem
  Iter::Iter(const Iter&); // to copy construct arguments
}
```

4 Refactoring

The term refactoring is derived from the mathematical term “factoring” and refers to finding multiple occurrences of similar code and factoring it into a single reusable function, thereby simplifying code comprehension and future maintenance tasks [20]. The meaning of refactoring has evolved and broadened. In [21], Opdyke and Johnson define refactoring as an automatic and behavior preserving code transformation that improves source code that was subject to gradual structural deterioration over its life time. Essentially, refactorings improve the design of existing code [22] [23].

Traditionally, refactoring techniques have been applied in the context of object-oriented software development. Automated refactoring simplifies modifications of a class, a class hierarchy, or several interacting classes [21]. More recently, refactoring techniques have been developed to support programs written in other programming styles (i.e., functional programming [24]).

Refactorings capture maintenance tasks that occur repeatedly. Opdyke [25] studied recurring design changes (e.g., component extraction, class (interface) unification). Refactoring is a computer assisted process that guarantees correctness, thereby enabling programmers to maintain and develop software more efficiently. In particular, evolutionary (or agile) software development methodologies [26], where rewriting and restructuring source code frequently is an inherent part of the development process of feature extensions, benefit from refactoring tools.

"Anti-patterns" [27] and "code smells" [22] are indicators of design deficiencies. Anti-patterns are initially structured solutions that turn out to be more troublesome than anticipated. Examples for anti-patterns include the use of exception-handling for normal control-flow transfer, ignoring exceptions and errors, magic strings, and classes that require their client-interaction occur in a particular sequence. Source code that is considered structurally inadequate is said to suffer from code smell. Examples for "code smell" include repeated similar code, long and confusing functions (or methods), overuse of type tests and type casts. The detection of code smell can be partially automated [28] and assists programmers in finding potentially troublesome source locations. Refactoring of anti-patterns and "code smells" to more structured solutions improves safety and maintainability.

Refactoring does not emphasize a particular goal or direction of source code modification - e.g., refactoring supports class generalization and class specification [25], refactoring can reorganize source code towards patterns and away from patterns (in case a pattern is unsuitable) [23].

Refactoring strictly preserves the observable behavior of the program. The term "observable behavior", however, is not well defined [20]. What observable behavior exactly requires (e.g., function call trace, performance, ...) remains unclear. Refactoring does not eliminate bugs, but can make bugs easier to spot and fix.

4.1 Source Code Rejuvenation and Refactoring

Table 1 summarizes characteristics of source code rejuvenation and refactoring. Both are examples of source code analysis and transformations that operate on the source level of applications. Refactoring is concerned to support software development with tools that simplify routine tasks, while source code rejuvenation is concerned with a one-time software migration. Both are examples of source code analysis and transformation. Source code rejuvenation gathers information that might be dispersed in the source of involved workaround techniques and makes the information explicit to compilers and programmers. Refactoring emphasizes the preservation of behavior, while source code rejuvenation allows for and encourages behavior improving modifications.

	Source Code Rejuvenation	**Refactoring**
Transformation	Source-to-source	Source-to-source
Behavior preserving	Behavior *improving*	Behavior preserving
Directed	yes Raises the level of abstraction	no
Drivers	Language / library evolution	Feature extensions Design changes
Indicators	Workaround techniques / idioms	Code smells Anti-patterns
Applications	One-time source code migration	Recurring maintenance tasks

Table 1. Source Code Rejuvenation vs. Refactoring

We might consider code rejuvenation a "subspecies" of refactoring (or vise versa), but that would miss an important point. The driving motivation or code rejuvenation is language and library evolution rather than the gradual improvement of design within a program. Once a rejuvenation tool has been configured, it can be applied to a wide range of programs with no other similarities than they were written in an earlier language dialect or style.

5 Tool Support for Source Code Rejuvenation

For our implementation of a source code rejuvenation tools, we utilize the Pivot source-to-source transformation framework [29]. The Pivot's internal program representation (IPR) allows for representing a superset of C++ including some programs written in the next generation of C++. IPR can be conceived as a fully typed abstract syntax tree. IPR represents C++ programs at a level that preserves most information present in the the source code. For example, IPR preserves uninstantiated template code. This allows us to analyse template and improve template definitions, for example, by deducing concepts or rejuvenating template function bodies.

We stress that the IPR is fully typed. Type information enables the implementation of source code rejuvenation that is type sensitive. Most of the potential rejuvenation analysis and transformations depend on type information. For example, concept extraction distinguishes operations that are template argument dependent from operations that are not. Likewise, the implementation to migrate source code to use initializer lists depends on whether the container type supports initializer-list constructors. This is the case for standard STL containers.

Related work includes systems for source code evolution and transformations. MoDisco [30], which is part of Eclipse, provides a model driven framework for source code modernization. The Design Maintenance System (DMS) [31] is an industrial project that provides a transformation framework. DMS supports the evolution of large scale software written in multiple languages. Stratego/XT [32] is a generic transformation framework that operates on an annotated term (ATerm) representation. Rose [33] provides a source-to-source translation framework for C++ and Fortran programs.

6 Conclusion

In this paper, we have discussed source code rejuvenation, a process that automates and assists source code changes to take advantage of improvements to programming languages and its libraries. We have supported our arguments with two specific examples from the migration from C++03 to C++0x.

We are aware that refactoring has been used to describe semantic preserving code transformations that migrate code to use new frameworks (e.g., Tip et al. [34], Tansey and Tilevich [35]). In this paper, we have demonstrated with examples that the difference between language evolution related code transformations and refactoring is subtle but important. We prefer and suggest the term "source code rejuvenation" for describing one-time and directed source code transformations that discover and eliminate outdated workaround techniques and idioms.

References

1. Ritchie, D.M.: The development of the c language. In: HOPL-II: The second ACM SIGPLAN conference on History of programming languages, New York, NY, USA, ACM (1993) 201–208
2. Stroustrup, B.: The design and evolution of C++. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994)
3. ISO/IEC 14882 International Standard: Programming languages: C++. American National Standards Institute (September 1998)
4. Stroustrup, B.: The design of c++0x. C/C++ Users Journal (2005)
5. Becker, P.: Working draft, standard for programming language c++. Technical Report N2914 (June 2009)
6. Becker, P.: The C++ Standard Library Extensions: A Tutorial and Reference. 1st edn. Addison-Wesley Professional, Boston, MA, USA (2006)
7. van Rossum, G.: The Python Language Reference Manual. Network Theory Ltd. (September 2003) Paperback.
8. ECMA: The C# language specification. Technical report, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland (June 2006)

9. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. 4th edn. Prentice Hall, PTR (August 2005)
10. Miller, A. http://tech.puredanger.com/java7 retrieved on July 6th, 2009.
11. Austern, M.H.: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)
12. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: Stapl: A standard template adaptive parallel C++ library. In: LCPC '01, Cumberland Falls, Kentucky (Aug 2001) 193–208
13. Merrill, J., Vandevoorde, D.: Initializer lists — alternative mechanism and rationale. Technical Report N2640, JTC1/SC22/WG21 C++ Standards Committee (2008)
14. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 291–310
15. Gregor, D., Stroustrup, B., Siek, J., Widman, J.: Proposed wording for concepts (revision 4). Technical Report N2501, JTC1/SC22/WG21 C++ Standards Committee (February 2008)
16. Pirkelbauer, P., Dechev, D., Stroustrup, B.: Extracting concepts from C++ generic functions. Technical report, Dept. of Computer Science and Engineering, Texas A&M (October 2009)
17. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
18. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal (21(6)) (June 2003) 25–32
19. Myers, N.C.: Traits: a new and useful template technique. C++ Report (1995)
20. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2) (2004) 126–139
21. Opdyke, W.F., Johnson, R.E.: Creating abstract superclasses by refactoring. In: CSC '93: Proceedings of the 1993 ACM conference on Computer science, New York, NY, USA, ACM (1993) 66–73
22. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
23. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
24. Lämmel, R.: Towards generic refactoring. In: RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming, New York, NY, USA, ACM (2002) 15–28
25. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992) UMI Order No. GAX93-05645.
26. Abrahamsson, P., Salo, O., Ronkainen, J.: Agile software development methods: Review and analysis. Technical report, VTT Electronics (2002)
27. Brown, W.J., Malveau, R.C., McCormick, III, H.W., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc., New York, NY, USA (1998)

28. Parnin, C., Görg, C., Nnadi, O.: A catalogue of lightweight visualizations to support code smell inspection. In: SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization, New York, NY, USA, ACM (2008) 77–86
29. Stroustrup, B., Dos Reis, G.: Supporting SELL for high-performance computing. In: 18th International Workshop on Languages and Compilers for Parallel Computing. Volume 4339 of LNCS., Springer-Verlag (October 2005) 458–465
30. : Eclipse MoDisco project. http://www.eclipse.org/gmt/modisco/ retrieved on September 20th, 2009.
31. Baxter, I.D.: Dms: program transformations for practical scalable software evolution. In: IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, New York, NY, USA, ACM (2002) 48–51
32. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.16: components for transformation systems. In: PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, New York, NY, USA, ACM (2006) 95–99
33. Quinlan, D., Schordan, M., Yi, Q., Supinski, B.R.d.: Semantic-driven parallelization of loops operating on user-defined containers. In: Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'03), Springer-Verlag Lecture Notes in Computer Science (2003)
34. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 265–279
35. Tansey, W., Tilevich, E.: Annotation refactoring: inferring upgrade transformations for legacy applications. In: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. Volume 43., New York, NY, USA, ACM (2008) 295–312