Author manuscript, published in "16th European Conference on Software Maintenance and Reengineering (CSMR 2012), Szeged : Hungary (2012)"
DOI : 10.1109/CSMR.2012.42

hal-00662777, version 1 - 25 Jan 2012

Invertible Program Restructurings for Continuing Modular Maintenance

Julien Cohen
ASCOLA team (EMN, INRIA, LINA)
University of Nantes
Nantes, France
Email: Julien.Cohen@univ-nantes.fr

Rémi Douence
ASCOLA team (EMN, INRIA, LINA)
INRIA
Nantes, France
Email: Remi.Douence@inria.fr

Akram Ajouli
ASCOLA team (EMN, INRIA, LINA)
EMN
Nantes, France
Email: Akram.Ajouli@mines-nantes.fr

***Abstract*—When one chooses a main axis of structural decompostion for a software, such as function- or data-oriented decompositions, the other axes become secondary, which can be harmful when one of these secondary axes becomes of main importance. This is called the tyranny of the dominant decomposition.**

In the context of modular extension, this problem is known as the Expression Problem and has found many solutions, but few solutions have been proposed in a larger context of modular maintenance.

We solve the tyranny of the dominant decomposition in maintenance with invertible program transformations. We illustrate this on the typical Expression Problem example. We also report our experiments with Java and Haskell programs and discuss the open problems with our approach.

***Keywords*-modular maintenance; restructuring; invertible program transformations; tyranny of the dominant decomposition;**

I. Introduction

Evolvability is a major criteria of quality for enterprise software. Evolvability is directly impacted by the design choices on the software architectures [1]. However, it is generally impossible to find software architectures that are evolvable with respect to all concerns. So, one of these concerns has to be privileged at the expense of other ones. This is sometimes called the *tyranny of the dominant decomposition* [2]. At the micro-architecture level, there are many ways to provide modular extensions which are orthogonal to the main axis of decomposition of a code structure, such as using open classes [3] in which one can add methods without modifying the source code of those classes (see a review of several solutions in [4]). However, these solutions generally break the regularity of the initial architecture (architectural degeneration), which results in a decrease in the maintainability (Sec. II). This reveals a tension between modular extension and modular maintenance.

In this paper, we use invertible program transformations between pairs of "dual" code structures to solve the tyranny of the dominant decomposition. We illustrate this with two code structures, data- and operation-oriented, for which we have built transformations with refactoring tools (Sec. III and V). We also give the challenges to be solved to make this approach fully automatic and scalable (Sec. VI), based on our experience with Java and Haskell program transformations (Sec. IV).

II. The Modular Maintenance Problem

In this section, we illustrate the fact that with fixed code structures, maintenance cannot be modular with respect to independent features (for instance, the set of operations on a data type is independent of the set of possible cases in that data type). We illustrate this in an object oriented setting on a Java program, but the problem is not restricted to object oriented architectures.

A. Each Architecture Privileges Modular Maintenance on a Given Axis

When choosing a class structure (or more generally a module structure) for a given program, one has to choose between several possibilities with different advantages and disadvantages [1]. We illustrate this with two possible class structures for a simple evaluator which have dual advantages and disadvantages : Composite (or Interpreter) and Visitor design patterns (Figs. 1 and 2). This program is the same that is often used to illustrate the expression problem [5], here given in Java.

The data type `Expr` represents the expression language to be evaluated. It is represented by an abstract class. The type `Expr` has a subtype for literals (`Num` for integers) and another for an operator (`Add` for additions). Two operations (methods) are defined on the type `Expr` : `eval` to evaluate expressions and `show` to transform them into strings. Their behavior is defined by case on subtypes. We call the code that defines the behavior of these two operations the *business code*. In the following, we are interested in the location of the business code in the class structure (which determines the modularity of maintenance tasks).

In the Composite architecture (Fig. 1), the business code which deals with a given subtype is delimited by the corresponding class. The diagram in Fig. 3(a) shows a matrix indexed on subtypes and operations. The concrete classes form a partition of the matrix according to the subtypes covered by the business code they contain. For instance, the class `Add` contains the business code for the two operations but only the part which concerns the subtype `Add`.

In this architecture, the maintenance concerning a given subtype is modular: when the requirements or the internal representation of a subtype changes, all the changes in the

hal-00662777, version 1 - 25 Jan 2012

```
abstract class Expr {
  abstract Integer eval ();
  abstract String show ();
}
```

```
class Num extends Expr {
  int n;
  Num (int n){ this.n=n ;};

  Integer eval() { return n; }

  String  show() { return Integer.toString(n); }
}
```

```
class Add extends Expr {
  Expr e1,e2 ;

  Add (Expr e1, Expr e2){
    this.e1 = e1 ;
    this.e2 = e2 ;
  }

  Integer eval() { return e1.eval() + e2.eval() ; }

  String  show() {
    return "(" + e1.show() + "+" + e2.show() + ")" ;}
}
```

Fig. 1. Data decomposition (Composite/Interpreter pattern) in Java – program P_{data}.

business code are located in the corresponding class. On the other hand, the maintenance of a given operation is not modular: when the requirements for an operation changes, the changes in the business code can be spread over the subclasses.

The program with the Visitor architecture (Fig. 2) has *dual properties* with respect to modularity. Its class structure makes that all the business code related to a given operation are located in a single class. For instance, the class EvalVisitor contains all the business code for the method eval. The matrix of Fig. 3(b) pictures that the classes with the business code do not cover subtypes anymore but operations.

In this architecture, the maintenance of a given operation is modular: when the requirements for an operation changes, all the changes in the business code are located in a single class. On the other hand, the maintenance of a given subtype is not modular: when a subtype changes, the changes in the business code can be spread over the visitor classes.

This duality illustrates the tyranny of the dominant decomposition in action: whatever program structure is chosen, some maintenance will be non modular. In the following, we call this the *modular maintenance problem*.

B. The Modular Maintenance Problem: Functional Programming Style

The opposition between data oriented architectures and operation oriented architectures is not specific to object oriented programs. In functional languages, functions are frequently defined by pattern matching on the structure of data. This corresponds to an operation oriented architecture: maintaining

```
abstract class Expr {
  Integer eval(){return(accept(new EvalVisitor()));}

  String  show(){return(accept(new ShowVisitor()));}

  abstract <T> T accept (Visitor <T> v) ;
}
```

```
class Num extends Expr {
  int n;
  Num (int n){ this.n=n ;};

  <T> T accept(Visitor <T> v){return v.visit(this);}
}
```

```
class Add extends Expr {
  Expr e1,e2 ;

  Add (Expr e1, Expr e2){
    this.e1 = e1 ;
    this.e2 = e2 ;
  }
  <T> T accept(Visitor <T> v){return v.visit(this);}
}
```

```
abstract class Visitor <T> {
  abstract T visit(Num n);
  abstract T visit(Add a);
}
```

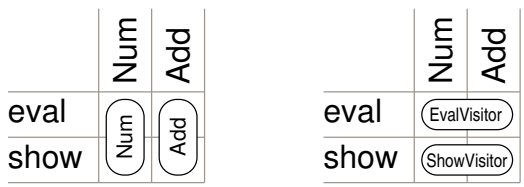
```
class EvalVisitor extends Visitor <Integer> {
  Integer visit(Num a){ return a.n; }

  Integer visit(Add a) {
    return a.e1.accept(this) + a.e2.accept(this) ; }
 }
```

```
class ShowVisitor extends Visitor <String> {
  String visit(Num a){return Integer.toString(a.n);}

  String visit(Add a) {
    return "(" + a.e1.accept(this) +
           "+" + a.e2.accept(this) + ")" ; }
}
```

Fig. 2. Functional decomposition (Visitor pattern) in Java – program P_{fun}.



(a) Data decomposition (program P_{data}).

(b) Functional decomposition (program P_{fun}).

Fig. 3. Coverage of classes with respect to operations and data type.

hal-00662777, version 1 - 25 Jan 2012

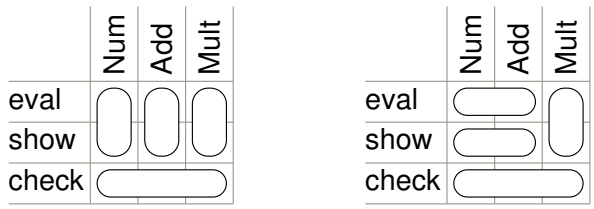
existing functions is modular but maintaining an existing case in the data type is not modular (the changes in business code can be spread over several functions).

An alternative way to define functions is to use traversal operators (*fold* catamorphisms) which take as parameter one function for each case in the data type. Since these parameter functions are specialized for given cases, it is relevant to group them into modules containing business code for specific cases of the data type. This corresponds to a data oriented architecture: maintaining a case in the data type is modular but maintaining a function is not modular (the changes in the business code are spread over several modules) [6].

C. Modular Extensibility (The Expression Problem)

A problem closely related to the modular maintenance problem exists with extensions: in the Composite architecture (we return to an object oriented setting), *adding* a new subtype is modular (the business code is added in the new class) but adding a new operation is not (the business code is spread over several classes), and inversely in the Visitor architecture. This is known as the *Expression Problem* [5].

There are many ways to extend the data-type or the set of operations indifferently in a modular way (see [4] for a review of some solutions). However, after the modular addition of an operation, the code is not modular anymore with respect to subtypes (see Fig 4), and after the modular addition of a subtype, the code is not modular anymore with respect to operations. For this reason, (language-based) solutions for modular extension conflict with modular maintenance.



(a) Data oriented initial decomposition.

(b) Operation oriented initial decomposition.

Fig. 4. Architecture after two modular extensions. We consider the two initial architectures described before (Fig. 3), extended with a subtype named Mult, then extended with an operation named check.

III. Invertible Program Transformations to solve the Modular Maintenance Problem

Chains of refactoring operations can be used to change the structure of programs while preserving their external behavior [7]. We propose to use *invertible* chains of refactoring operations to solve the problems of modular maintenance and modular extension.

First, the two programs P_{data} and P_{fun} of the previous section can be transformed one into the other by a behavior preserving program transformation, and inversely (we have implemented such invertible transformations for Java and for its Haskell functional counterpart, see Sec. IV).

Such transformations solve the problem of modular maintenance: when one faces an evolution task (which requires either to add a new subtype/operation or to modify an existing subtype/operation) to be performed which is not modular in the available form of the program, he applies the convenient transformation to get the program into the convenient form, then he implements the evolution in a modular way (see Fig 5). In the case of an extension, the resulting architecture is not degenerated.

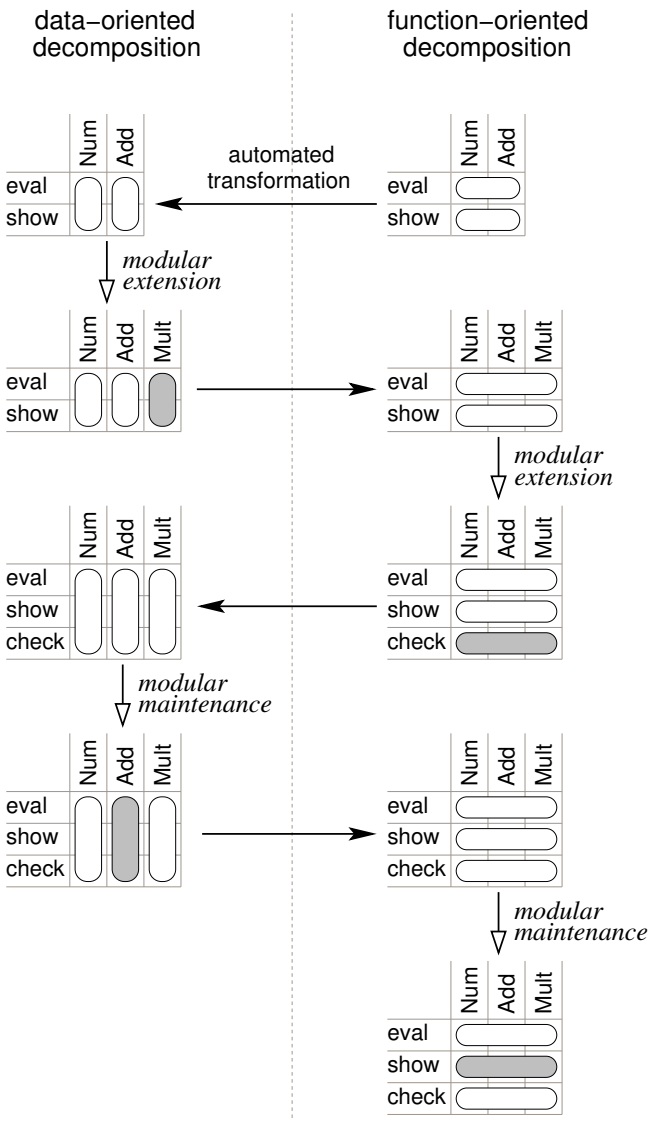


Fig. 5. Scenario for 4 evolutions with architecture transformations. The initial code is extended with the subtype Mult and with the operation check, then maintenance tasks are performed on the subtype Add and on the operation show. Structure transformations are performed so that all the evolutions are modular.

Once the evolution is implemented, one can either leave the program in the last form, or apply the inverse transformation to recover the initial structure with the implemented changes propagated.

hal-00662777, version 1 - 25 Jan 2012

IV. Experiment: Implementation of Architecture Transformations with Refactoring tools

We have made conclusive experiments with Java and Haskell.

In Java, we have put to test Composite ↔ Visitor transformations with Eclipse [8] and IntelliJ IDEA [9] refactoring tools. We describe in [10] the abstract algorithms we use, some variants we propose and the specificities due to the use of these tools. The whole transformation is not automated yet (we plan to automate this algorithm by using the tools API in conjunction with pattern detection tools such as [11]).

In Haskell, we have performed transformations between function oriented and data oriented architectures with the Haskell Refactorer [12]. We describe in [6] the abstract algorithms we have designed. The transformations are automated for several examples of programs. They are concretely defined by scripts much of which is reusable for other programs. We have customized the API of the Haskell Refactorer to be able to automate the transformation steps (see [6], [13]).

A. Results

Here is what we have observed from our experiments:

- The external behavior is preserved by transformations, as well as type safety.
- We find back the initial source code after performing a transformation and its inverse, except for the layout and the comments which have been disturbed.
- In the Java experiment, the visibility for the composite class elements has to change when passing from Composite to Visitor structures. This is not related to the transformation but rather to the nature of the Visitor pattern.
- On small/medium-size programs (we used programs with 6 subtypes and 6 operations), Java refactoring tools were fast enough, while the Haskell Refactorer was very slow: the Composite→Visitor takes about 3 minutes (plus several hours to chain the operations manually) while the Haskell Refactorer could take 30 seconds for an elementary renaming (but transformations are automated).
- Our algorithms are sensitive to variations in the initial structure.
- A few refactoring operations were needed but not covered by the tools. For Java, we have made some refactoring steps manually to validate the transformation algorithms. For Haskell, we have added five operations into the tool to be able to automate the full transformations.

V. Assessment

The results above show that our proposal is workable only with efficient tools. We expose the challenges to be solved to provide such tools in Section VI. In the rest of this section, we discuss more generally the pros and cons of our proposal.

Our approach does not rely on a particular programming language (we have dealt with two different languages). It applies as soon as two alternative programming structures can be expressed in a language. It results that:

- Our solution can be applied to legacy systems.
- The programmer's skill in the programming language is sufficient to implement modular evolutions. Our approach does not require that the programmer should master specific composition mechanisms such as aspects, mixins, open classes, or hyper-slices.
- Our solution does not induce runtime overhead.

On the other hand, a transformation tool capable of performing the architecture transformation must be available for the considered language (see Sec. VI).

Our solution is not limited to the data-centered *versus* the function-centered structures (see Sec. VII-C). It is not even limited to two structures (with the limitation that for each new structure to be considered, a pair of transformations must be available or defined).

Last, programmers already familiar with the initial program structure may lose their marks in a second structure.

VI. Challenges for Tool Support

Using refactoring tools to implement architectures transformations make transformations easy to design and tune since refactoring operations are rather high-level transformations. Refactoring operations are also easily composed to make more complex operations that can be used as components for building our transformations. Moreover, chains of refactoring operations are already used to describe the introduction of design patterns into existing code [14].

On the other hand, other aspects of refactoring tools make their use not entirely satisfactory in our context. We now discuss the challenges to get over in order to make our solution of industrial strength.

A. Soundness.

Using refactoring tools to implement architectures transformations has the advantage that the soundness of the transformation relies on the refactoring tool. However, it is frequent to face bugs in refactoring tools (we have faced several bugs in refactoring tools during our experiments). A single bug in a chain of elementary transformations make all the process fail.

Proofs of correctness of refactoring operations exist [15], [16], but we cannot expect refactoring tools to be proven correct in near future. However, we can expect that popular refactoring tools progressively become safer when bugs are reported.

B. Layout Preservation, Invertibility

With current refactoring tools, it seems impossible to design invertible architecture transformations that take layout and comments into account. A solution is to provide invertible versions of refactoring operations within the meaning of Bohannon *et al.* [17]: non invertible operations, such as deletion, can become invertible by keeping a trace of the program before transformation. This suggests that it could be useful to keep a reference architecture and to use alternate ones only temporarily. That would also allow several maintainers to share a common reference model in the case of teamwork.

hal-00662777, version 1 - 25 Jan 2012

C. Speed and Flexibility

To be workable, our proposal must be automated with convenient tools. The underlying refactoring tool must be sufficiently fast (which is the case for popular tools such as Eclipse but not for academic, prototype tools such as the Haskell Refactorer).

Moreover, to avoid time-consuming user interactions, transformation tools must be capable of detecting structures in programs and of adapting the chain of refactoring operations to these structures. We can consider using pattern detection tools bearing variations in pattern instances (such as [11]) and either to adapt the chain of refactoring operations to these variations upstream or to use tools that infer such chains of refactoring operations. For instance, in [18], the target structure is described by logic constraints.

D. Failures and Pre-Conditions

Since each operation of the chain of refactorings requires some preconditions to be satisfied, it may occur that the user is advised that the transformation cannot be achieved only during the transformation process. For this reason, providing pre-conditions for our transformations is desirable (pre-conditions for chains of refactoring operations are explored in Kniesel and Koch [19]).

E. Macro-Architectures

In this paper, we have dealt with source-code level architectures (micro-architectures). But alternate structures are also useful at the system level (macro-architectures) [20]. This suggests that transformations between dual macro-architectures as well as refactoring tools for composition/coordination languages should be explored.

VII. Related Work

A. Program Restructuring and Refactoring to Patterns

Work on refactoring have always considered that the aim of refactoring is to improve code structure (and so evolvability) [21], [22]. Since most of that work takes place in an object-oriented context, it is natural that design patterns have been considered as target code structures [23], [14]. Switching to alternate patterns has also been considered recently [24].

All that work is a basis for our proposal, but we are more demanding: we need invertible transformations, full automation, etc. (see Sec. VI).

B. Views

Offering alternate views of software artifacts is not a new idea and is useful in practice [25].

Wadler proposes a concept of views that allows to handle datatypes with several interfaces for pattern matching [26]. This permits the programmer to use the more convenient interface to implement an algorithm so that its design and evolvability are improved. However, extension of the data-type still requires cross-cutting changes in the algorithms. Also, the underlying mechanisms can introduce a run-time overhead.

Tarr *et al.* [2] propose to construct programs by composing possibly overlapping compilation units (*hyperslices*), each describing a concern. Hyperslices are useful for program comprehension since the concerns are clearly separated, but since they can be overlapping, evolutions can be difficult to implement.

Mens *et al.* [27] propose a system where concerns are described by a set of properties (a *view*). As for Tarr *et al.* [2], these views help for program comprehension and help to check that an evolution does not violate the properties of a concern, but it does not make the evolution modular.

Shonle *et al.* [28] also allow to define patterns (views) describing crosscutting parts of code of interest, but in addition the programmer can implement concern-specific evolutions based on these patterns.

We share with Black and Jones [29] a same theoretical concept of views: alternate forms of a program which are computed from that program, which external behavior are equivalent, with different structural properties and that can be transformed back to the initial structure. However, whereas we defend the use of "dual" code structures expressible in a same language, they propose to use language extensions to support alternate code structures. For instance, whereas we propose the Visitor code structure as a function oriented alternate view for the data-oriented code structure, they prefer to use a flattened class hierarchy (expressed in an extension of the initial language) so that all the business code for a given operation is grouped.

The number of proposals for concepts of views shows that there is an inclination to provide multiple views of software artifacts to improve separation of concerns. However, the work cited in this section have a common property: they are built *on top* on existing languages (language extensions, pattern languages, additional composition mechanisms...). This means that the programmer must be skilled not only in the base programming language, but also in the technology that provides views (to understand, use, define, modify or compose views). We stand out from this by not requiring these skills but by requiring that convenient transformations are provided instead.

C. Transformations between other pairs of dual architectures

Our approach is not limited to function oriented versus data oriented views. First, one can also provide a security view, a transaction view, or any view which reifies a concern that is subject to change. Second, views can be used to other aims than modularity. It can be used to navigate between conflicting design choices.

1) Add or remove structure: For instance, instead of changing the main axis of structure, one can need to add/remove structure. Adding a function that factorizes some code allows to hide a behavior, to name a concept, to remove code duplicates, to move piece of code for a concern to a given module/class. On the opposite, inlining/unfolding a function enables to remove an indirection or a dependency to a module, to ease an analysis. The same is true for class hierarchies

hal-00662777, version 1 - 25 Jan 2012

(class hierarchies make clean architectures but behavior code is spread over several files), or for aspects (understanding aspect interactions can be tricky). Is is also sometimes useful to add/remove polymorphism or machinery such as iterators to improve understanding and analysis.

2) Change internal behavior: More generally, software engineering offers fundamental design choices that could be (at least partially) supported by views. For instance, λ-lifting [30] (resp. λ-dropping [31]) adds (resp. removes) extra function parameters corresponding to free variables. The λ-lifted view promotes function reuse, and the λ-dropped view promotes efficiency. A same relationship exists between continuation passing style and direct style [32].

Another design tradeoff exists between computation time and storage in memory. This is exemplified by the choice to use memoization. Second example: when implementing a collection, one has the choice to compute the number of elements in the data-structure on demand or to store it in the data-structure and maintain it. In the latter case, yet another tradeoff occurs between updating the stored size at each update of the elements, or updating it only when the size is accessed.

Finally, a last tradeoff is related to when a computation occurs. For example, two processes can communicate synchronously or asynchronously, with or without buffers, etc.

These views are quite general and maybe impossible to support automatically. But, when possible, views can reduce the impact of making these design choices early, when future changes in requirements are not known yet.

VIII. Conclusion

The contributions of this article are the following:

- We show how invertible program transformations make continual modular maintenance along crosscutting concerns feasible.
- We point some technical and scientific challenges to make the approach workable, based on our experience in building tools to support such transformations.

Applying invertible structure transformations with (yet to provide) appropriate, fully automatic tools can enable to:

- Reduce structure degeneration with continual change.
- Reduce the impact of early design choices and reduce the cost of maintenance or incremental development for concerns which are transverse to the main axis of decomposition.
- Reduce the need for specific programming skills (such as aspects) for separation of concerns.

Acknowledgment

The authors would like to thank Jean-Louis Giavitto (IRCAM), Jean-Claude Royer (EMN) and the Haskell Refactorer team for their comments.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, December 1972.

[2] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Intl. Conf. on Soft. Eng. (ICSE '99)*. ACM, 1999, pp. 107–119.

[3] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "Multijava: modular open classes and symmetric multiple dispatch for java," in *OOPSLA '00*. ACM, 2000, pp. 130–145.

[4] M. Zenger and M. Odersky, "Independently extensible solutions to the expression problem," in *FOOL*, Jan. 2005.

[5] P. Wadler, "The expression problem," Nov. 1998, note to Java Genericity mailing list.

[6] J. Cohen and R. Douence, "Views, Program Transformations, and the Evolutivity Problem in a Functional Language," Jan. 2011, Research Report hal-00481941, http://hal.archives-ouvertes.fr/hal-00481941/en.

[7] W. B. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, July 1991.

[8] "Eclipse Foundation," www.eclipse.org.

[9] "Intellij IDEA," www.jetbrains.com/idea.

[10] A. Ajouli and J. Cohen, "Refactoring Composite to Visitor and Inverse Transformation in Java," Tech. Rep. hal-00652872, Dec. 2011, http://hal.archives-ouvertes.fr/hal-00652872/en.

[11] J. M. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code," in *ASE*. IEEE, 2003, pp. 215–224.

[12] H. Li and S. Thompson, "Tool Support for Refactoring Functional Programs," in *Partial Evaluation and Program Manipulation*, Jan. 2008.

[13] J. Cohen, "Haskell View Switcher," http://www.emn.fr/z-info/ascola/doku.php?id=internet:view_switcher, 2011.

[14] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[15] A. Garrido and J. Meseguer, "Formal specification and verification of Java refactorings," in *SCAM*. IEEE, 2006, pp. 165 – 174.

[16] N. Sultana and S. Thompson, "Mechanical Verification of Refactorings," in *Partial Evaluation and Program Manipulation*. ACM, Jan. 2008.

[17] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: resourceful lenses for string data," in *Principles of Programming Languages (POPL '08)*. ACM, 2008, pp. 407–419.

[18] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Intl. Conf. on Soft. Maint. (ICSM)*. IEEE, Sept. 2010.

[19] G. Kniesel and H. Koch, "Static composition of refactorings," *Science of Computer Programming*, vol. 52, no. Issues 1-3, pp. 9–51, Aug. 2004.

[20] A. Hannousse, G. Ardourel, and R. Douence, "Views for Aspectualizing Component Models," in *AOSD Workshop ACP4IS*, Mar. 2010.

[21] W. F. Opdyke, "Refactoring: A program restructuring aid in designing object-oriented application frameworks," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, 1992.

[22] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 228–269, July 1993.

[23] M. Ó Cinnéide, "Automated refactoring to introduce design patterns," in *Intl. Conf. on Soft. Eng. (ICSE '00)*. ACM, 2000, pp. 722–724.

[24] M. Hills, P. Klint, T. Van Der Storm, and J. Vinju, "A case of visitor versus interpreter pattern," in *Intl. Conf. on Objects, models, components, patterns (TOOLS'11)*. Springer-Verlag, 2011, pp. 228–243.

[25] S. Meyers and S. P. Reiss, "An empirical study of multiple-view software development," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 47–57, Nov. 1992.

[26] P. Wadler, "Views: a way for pattern matching to cohabit with data abstraction," in *POPL '87*. ACM, 1987, pp. 307–313.

[27] K. Mens, T. Mens, and M. Wermelinger, "Maintaining software through intentional source-code views," in *SEKE '02*. ACM, 2002, pp. 289–296.

[28] M. Shonle, W. G. Griswold, and S. Lerner, "Beyond refactoring: a framework for modular maintenance of crosscutting design idioms," in *ESEC-FSE*. ACM, 2007, pp. 175–184.

[29] A. P. Black and M. P. Jones, "The case for multiple views," in *ICSE 2004 Workshop on Directions in Soft. Eng. Environments*, 2004.

[30] T. Johnsson, "Lambda lifting: Transforming programs to recursive equations," in *FPCA*. Springer-Verlag, 1985, pp. 190–203.

[31] O. Danvy and U. P. Schultz, "Lambda-dropping: transforming recursive equations into programs with block structure," in *PEPM*. ACM, 1997, pp. 90–106.

[32] O. Danvy, "Back to direct style," in *European symp. on programming (ESOP'92)*. Elsevier Science Publishers B. V., 1994, pp. 183–195.