Recommending Refactorings to Reverse Software Architecture Erosion

Ricardo Terra[†‡], Marco Tulio Valente[†], Krzysztof Czarnecki[‡] and Roberto S. Bigonha[†]
[‡]*University of Waterloo, Canada*
[†]*Universidade Federal de Minas Gerais, Brazil*
Email: {terra,mtov}@dcc.ufmg.br, kczarnec@gsd.uwaterloo.ca, bigonha@dcc.ufmg.br

***Abstract*—Architectural erosion is a recurrent problem faced by software architects. Despite this fact, the process is usually tackled in ad hoc way, without adequate tool support at the architecture level. To address this issue, we describe the preliminary design of a recommendation system whose main purpose is to provide refactoring guidelines for developers and maintainers during the task of reversing an architectural erosion process. The paper formally describes first recommendations proposed in our current research and results of their application in a web-based application.**

I. Introduction

Software architecture erosion designates the progressive gap normally observed between the planned and the actual architecture of a software system as implemented by its source code [1]–[4]. Although the causes for this architectural gap are diverse, ranging from conflicting requirements to deadline pressures, the consequences always include degradation in the internal quality of the system, with a negative impact on properties like maintainability, evolvability, extensibility, and reusability [5]. When the process is accumulated over years, architectural erosion can transform software architectures into unmanageable monoliths [6], [7].

In order to maintain the long-term survival of any system, software architects have the responsibility for establishing policies to monitor the erosion process. Particularly, such policies must provide answers to the following concerns:

1) How to locate the points of erosion in a software system? In other words, how to locate in the source code the implementation decisions that represent violations to the planned architecture?
2) After locating the violations, how to reverse the erosion process? Stated differently, how to refactor the source code to replace architectural violations by implementation decisions that are consistent with the planned architecture?

The first concern has been widely investigated in the literature on software architecture evolution and several solutions have been proposed to uncover architectural violations [8]–[10]. As particular examples, we can mention techniques such as reflexion models [11], intensional views [12], and domain-specific languages, such as SCL [13], DCL [14], LogEn [15], and .QL [16]. There are also industrial-strength architecture analysis tools that, to some extent, are based on such techniques and languages, including Lattix LDM [17], IESE SAVE [18], and Semmle ODASA [19].

There has been less research effort dedicated to the second concern, however. Usually, the common recommendation for removing architectural violations reduces to the use of traditional refactorings [20], as supported by today's IDEs. However, this recommendation is too generic in most scenarios. Suppose, for example, that a developer has created an object of type `Product` in a module where this object creation is not allowed. To fix this violation, the standard recommendation consists in applying a *Replace Constructor by a Factory Method* refactoring. However, most developers do not have a complete understanding of the system and therefore they may not know the name of the Factory method to call in this situation. As a second example, suppose a system where database operations are confined to Data Access Objects (DAOs). Suppose also that a database query has been performed outside a DAO. To fix this violation, developers must apply the *Extract Method* and then *Move Method* refactorings. However, they may require considerable time to determine the particular DAO the query must be moved to.

To address the lack of tool support for removing architectural violations, we describe in this paper the preliminary design of a recommendation system whose main purpose is to provide refactoring guidelines for developers and maintainers during the task of reversing an architectural erosion process. For example, our system not only indicates the application of a *Replace Constructor by a Factory Method*, but it complements this information with the name of the Factory method that must be called in this particular context. Similarly, when a *Move Method* is recommended, the system suggests the target class of the move refactoring. We acknowledge that providing a fully automatic approach for removing architectural violations is challenging and far ahead the state of the art in refactoring tools. On the other hand, due to its relevance, we take the position that architectural erosion should not continue to be tackled in ad hoc way, without adequate tool support at the architecture level.

Specifically, our approach provides recommendations to remove architectural violations detected by the DCL language [14], [21]. DCL (*Dependency Constraint Language*) is a domain-specific language with a simple, easy understandable syntax for defining structural constraints between modules. Once defined, such restrictions are statically checked by a conformance tool, called `dclcheck`. Figure 1 illustrates how the proposed approach leverages the architecture conformance definition provided by the DCL language. The proposed recommendation engine,

called `dclfix`, requires the following inputs: a set of DCL constrains (specified by a software architect or designer), a set of architectural violations (raised by the `dclcheck` tool), and the source code of the system. Based on these inputs, the engine provides a set of recommendations to guide the process of removing the detected violations.

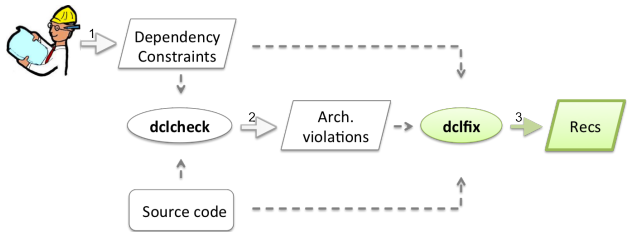

Figure 1. Proposed recommendation engine

Structure: The remainder of this paper is structured as follows. Section II provides an overview on the DCL language. Section III presents a specification for our current set of recommendations. Section IV presents preliminary results on applying these recommendations in a web-based system. Finally, Sections V and VI briefly discuss related and further work, respectively.

II. The DCL Language

Because existing object-oriented languages allow client modules to reference any public type exported by server modules, the rationale behind DCL is to provide architects with means to control such dependencies [14], [21]. Particularly, the language provides constraints to capture two types of architectural violations: *divergences* (when a dependency that exists in the source code violates the planned architecture) and *absences* (when the source code does not establish a dependency that is prescribed by the planned architecture) [8], [11]. To capture divergences, DCL allows architects to specify that dependencies *only can*, *can only* or *cannot* be established by specified modules (the differences will be explained shortly). In addition, to capture absences, architects can specify that particular dependencies *must* be present in the source code.

To illustrate the use of DCL, assume the following constraints:

```
1: only Factory can-create Product
2: Util can-only-depend $java, Util
3: View cannot-access Model
4: Product must-implement Serializable
```

These constrains state that only classes in the `Factory` module can create objects from classes in the `Product` module (line 1), classes in the `Util` module can only establish dependencies with the Java API and with classes in the own `Util` module (line 2), and classes in the `View` module cannot access classes in the `Model` module (line 3). The last constraint defines that all classes in the `Product` module must implement the mentioned interface (line 4).

When defining constraints, DCL allows developers to specify dependencies caused by accessing methods and fields (`access`), declaring variables (`declare`), creating objects (`create`), extending classes (`extend`), implementing interfaces (`implement`), throwing exceptions (`throw`), or using annotations (`annotate`). It is also possible to define constrains including any form of dependency (`depend`).

Formal Semantics: Since the proposed recommendations aim to remove violations in architectural constraints defined in DCL, it is important to provide a formal semantics for the language. For this purpose, suppose that $\mathtt{M_A}$ and $\mathtt{M_B}$ are modules, which are sets of classes. Moreover, suppose that $\overline{\mathtt{M_A}}$ denotes the complement of module $\mathtt{M_A}$, i.e., all classes of the system under analysis except those in $\mathtt{M_A}$. Finally, suppose that `dep` denotes a dependency type provided by DCL, i.e., `dep` can be `access`, `declare`, `create`, etc.

First, the semantics of *only can* and *can only* rules are defined in terms of *cannot* rules, as described next:

$$\mathtt{only\ M_A\ can\text{-}dep\ M_B} \Longrightarrow \overline{\mathtt{M_A}}\ \mathtt{cannot\text{-}dep\ M_B}$$

$$\mathtt{M_A\ can\text{-}only\text{-}dep\ M_B} \Longrightarrow \mathtt{M_A\ cannot\text{-}dep}\ \overline{\mathtt{M_B}}$$

Furthermore, a violation in a rule of the form $\mathtt{M_A\ cannot\text{-}dep\ M_B}$ happens whenever

$$\exists \mathtt{A}\, \exists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \ \wedge\ \mathtt{B} \in \mathtt{M_B} \ \wedge\ \mathtt{dep(A,B)}\,]$$

where `A` and `B` denote classes and `dep` is a predicate that checks whether there is a dependency of type `dep` from `A` to `B`.

Finally, a violation in a rule in the form $\mathtt{M_A\ must\text{-}dep\ M_B}$ happens whenever

$$\exists \mathtt{A}\, \neg\exists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \ \wedge\ \mathtt{B} \in \mathtt{M_B} \ \wedge\ \mathtt{dep(A,B)}\,]$$

III. Refactoring Recommendations

The proposed refactoring recommendations aim to assist developers and maintainers—especially those who are unfamiliar with the architecture—to fix violations detected by DCL. We decided to rely on the following syntax to specify the recommendations:

Architectural_Rule
Code_With_Violation $\Longrightarrow$ *Recommendation*, `if` *Preconditions*

This syntax should be interpreted as follows: whenever the *Architectural_Rule* is violated by the particular *Code_With_Violation* and the *Preconditions* hold, the *Recommendation* can be triggered. More specifically, *Architectural_Rule* is a constraint defined in DCL and *Code_With_Violation* is the particular statement or expression in the source code where this constraint has been violated. A *Recommendation* consists of a sequence of refactoring operations, using the functions described in Table I; and *Preconditions* for the proposed recommendations are defined using the functions listed in Table II.

Based on the proposed syntax and functions, Table III shows some of the refactoring recommendations we have

Table I
REFACTORING FUNCTIONS

Refactoring	Description
`extract(stm)`	Extracts method with statements `stm`
`move(f, C)`	Move method `f` to the class `C`
`move(C, M)`	Move class `C` to the module `M`
`remove(S)`	Removes the block of code `S`
`replace(stm1, stm2)`	Replaces block of code stm_1 by stm_2
`propagate(exp, v, S)`	Propagates `exp` to the uses of the variable `v` in the block of code `S`
`promote(f, v, exp)`	Promotes variable `v` to a formal parameter of `f`; the expression `exp` is used as the argument in the calls to `f`

Table II
FUNCTIONS USED IN PRECONDITIONS

Function	Description
`can(T1, dep, T2)`	Checks whether type T_1 can establish a dependency of the `dep` kind with T_2
`call_sites(f)`	Returns the call sites of `f`
`delegate(f)`	Searches for a delegate method for `f`
`factory(C, exp)`	Searches for a factory for class `C`, accepting `exp` as input
`gen_decl(T, f)`	Declares a variable of class `T` to access `f`
`gen_factory(T, exp)`	Generates a factory for class `T`, accepting `exp` as input
`super(t)`	Returns the supertypes of type `t` (in order from the most specific to the most generic)
`type(v)`	Returns the type of the variable `v`
`typecheck(stm)`	Checks whether code `stm` type checks
`user_code()`	Prompts the user for a block of code

already formalized. This table shows recommendations for violations in *cannot* rules including dependencies due to `create`, `declare`, `access`, `throw`, and `derive` relations. We have formalized more than twenty recommendations for violations in *cannot* rules. Table III shows only a subset of them due to space restrictions. Alternatively, code may be surrounded by quotation brackets. It is important to mention that the proposed recommendations can also handle violations in *only can* and *can only* rules, because these rules are defined in terms of *cannot* constraints, as described in Section II. Finally, Table III does not include recommendations for *must* rules; we are currently investigating recommendations for these rules.

We briefly describe the recommendations listed in Table III:

- Recommendations for `cannot-create` rules are always associated to a `new` operator, as in the case of the first two recommendations. In both cases, the recommendation includes the replacement of the `new` operator by a call to the `get` method of a Factory class. The recommendation may suggest using a method from an existing Factory (rec. 1a) or the creation of a new Factory class, using the function `gen_factory` (rec. 1b).
- Recommendations for `cannot-declare` rules may include the replacement of the unauthorized type `B` by one of its supertypes B' (rec. 2a). This recommendation is particularly useful to handle violations due to the use of a concrete implementation of a service, instead of its general interface. Repairing violations of `cannot-declare` rules may also involve the removal of the unauthorized declaration followed by the propagation of the initialization expression `exp` to all uses of the declared identifier (rec. 2b).
- Recommendations for `cannot-access` rules may include the replacement of an unauthorized call to a method `f` by a call to a delegate method `g` (rec. 3a). As an alternative, the system may suggest the extraction of a new method `g` with the call to `f` and the movement of `g` to another class `C` (rec. 3b). Section III-A provides more information about the algorithm proposed to find an appropriate class for the extracted method. Finally, the recommendation engine may suggest promoting the variable—whose initialization expression contains the unauthorized access—to a formal parameter of the enclosing method `g`. In this case, the initialization expression `exp_b` must be used as the argument in `g` calls (rec. 3c).
- Recommendations for `cannot-throw` rules may include the removal of the `throws` clause, in the cases it is not needed (rec. 4a). On the other hand, when the `throws` is needed, its removal can be followed by the insertion of a `try-catch` block around the body of the method to handle the exception internally (rec. 4b). In this particular case, the developers must provide the code that handles the exception, as required by function `user_code`.
- Recommendations for `cannot-derive` rules may include the movement of the entire class to a more appropriate module (rec. 5a).

In Table III, the recommendations are listed according to their priority. When two or more recommendations match a given violation in the source code, the system only triggers the recommendation with the highest priority.

A. Class Similarity

Recommendation 3b suggests the movement of a method to another class, as computed by the function `suitable_class`. To implement this function, the similarity between a method and a class is calculated using the Jaccard Similarity Coefficient, which is a statistical measure for the similarity between two sets. To calculate the coefficient, we assumed that a method or a class is represented by the set of structural dependencies it establishes with other program elements. This assumption is based in the fact that our recommendations have been proposed to handle violations in DCL constraints, which basically denote divergences (or absences) in the expected set of dependencies established by a given program element.

Based on such assumptions, the similarity between a method `f` and a class `C` is defined by:

$$similarity(f, C) = \frac{|Deps(f) \cap Deps(C)|}{|Deps(f) \cup Deps(C)|}$$

Table III
REFACTORING RECOMMENDATIONS

A cannot-create B, where $A \in M_A \wedge B \in M_B$		
`new B(exp)`	$\Longrightarrow$ `replace( [new B(exp)], [FB.getB(exp)] ),` if `FB = factory(B, [exp] )` $\wedge$ `can(A, access, FB)`	(1a)
`new B(exp)`	$\Longrightarrow$ `replace( [new B(exp)], [FB.getB(exp)] ),` if `FB = gen_factory(B, [exp] )` $\wedge$ `can(A, access, FB)`	(1b)
A cannot-declare B		
`B b; S`	$\Longrightarrow$ `replace( [B], [B'] ),` if $B' \in$ `super(B)` $\wedge$ `typecheck( [B' b; S] )` $\wedge$ $B' \notin M_B$	(2a)
`B b = exp; S`	$\Longrightarrow$ `propagate( [exp], b, [S] ),` if `can(A, access, B)`	(2b)
A cannot-access B		
`b.f`	$\Longrightarrow$ `replace( [b.f], [D; c.g] ),` if `g = delegate(f)` $\wedge$ `D = gen_decl(type(c), g)` $\wedge$ `type(c)` $\notin M_B$	(3a)
`b.f`	$\Longrightarrow$ `g = extract( [b.f] ), move(g, C),` if `C = suitable_class(g)` $\wedge$ `can(A, access, C)`	(3b)
`g { T v = exp_b }`	$\Longrightarrow$ `promote(g, v, [exp_b] ),` if $\forall$`C` $\in$ `call_sites(g), can(C, access, B)`	(3c)
A cannot-throw B		
`g (p) throws B { S }`	$\Longrightarrow$ `remove( [throws B] ),` if `typecheck( [g (p) { S }] )`	(4a)
`g (p) throws B { S }`	$\Longrightarrow$ `remove( [throws B] ), replace( [S], [try {S} catch (B b){S'}] ),` if `can(A, declare, B)` $\wedge$ `S' = user_code()`	(4b)
A cannot-derive B		
`A extends | implements B`	$\Longrightarrow$ `move(A, M),` if `M = suitable_module(A)` $\wedge$ `can(A, extends | implements, B)`	(5a)

where *Deps* denotes the set of dependencies established by a method or class. This definition ensures that similarity value ranges over the interval $[0, 1]$. Using this definition, function `suitable_class(f)` returns the class C with the highest value for `similarity(f, C)`.

A similar approach is used by function `suitable_module(C)` to return a more appropriate module to host the implementation of a class `C`.

IV. EXAMPLE

To illustrate the use of our approach, we have developed a simple web-based e-commerce system, called `MyWebMarket`, including functions to manage customers and products, handle purchase orders, generate reports, etc. This system has been carefully designed and implemented to resemble on a smaller scale the architecture of a real-world human resource management system, with more than 200 KLOC, which we have used to evaluate the DCL language [14]. The decision to mirror the architecture of a previously evaluated system has allowed us to reuse its DCL constraints. It also simplified the insertion of architectural violations similar to those that have been detected in the real system. The `MyWebMarket` system provides us with a controlled environment to illustrate architectural erosion and our approach to counter it.

Architecture: Figure 2 illustrates `MyWebMarket` architecture. The architecture follows the well-known Model-View-Controller (MVC) pattern and it relies on several frameworks widely used when architecting web-based systems (including Hibernate, Struts, JSP, DWR, Quartz, etc.).

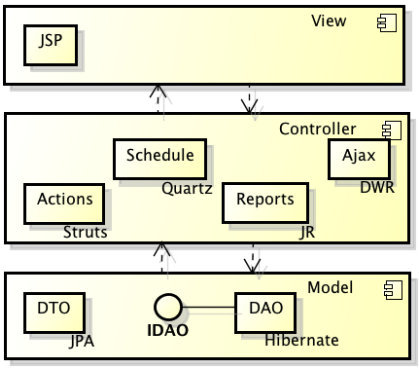

Figure 2. `MyWebMarket` architecture

Architectural Constraints: The following DCL constraints have been defined for `MyWebMarket`:

```
#1: only DAOFactory can-create HibernateDAO
#2: Controller cannot-handle HibernateDAO
#3: Model can-only-throw DAOException
#4: only SystemScheduling can-depend QuartzAPI
```

These constraints cover the main categories of architectural violations detected in our past experience [14]. More specifically, they provide illustrative examples for the following categories of violations: improper use of design patterns (constraint #1), improper use of persistence patterns (constraint #2), bypassing the MVC layers (constraint #3), and unauthorized use of frameworks (constraint #4).

Architectural Violations and Recommendations: Table IV presents the number of violations we have inserted in the code for each defined DCL constraint. It also shows the number of triggered recommendations to fix such

violations and the number of violations that remained in the code after following these recommendations.

Table IV
VIOLATIONS AND RECOMMENDATIONS FOR `MyWebMarket`

Constraint	# Violations before	# Recs.	# Violations after
#1	21	21	0
#2	43	15	12
#3	44	44	0
#4	9	9	0

We briefly comment the results presented in Table IV:

- Constraint #1 prescribes the use of a factory. To violate this constraint, we have deliberately replaced all uses of the factory by a direct instantiation, resulting in 21 violations. For each of such violations, our approach has triggered recommendation 1a, with the suggestion to use the existing factory. After following this recommendation, all violations have been removed.

- Constraint #2 prescribes the use of well-defined interfaces to persistence purposes. To violate this constraint, we have deliberately bypassed the use of such interfaces by coupling directly with concrete implementations, resulting in 43 violations (21 declarations and 22 accesses). In 15 `declare` violations, our approach has triggered recommendation 2a, with the suggestion to generalize the type. By following this recommendation, 31 violations have been removed, even `access` violations, since the accesses go through the interfaces. On the other hand, the remaining violations could not be removed because the accessed methods were not defined in the interfaces (i.e., the interface has not evolved to include signature of new public methods provided by the concrete implementations).

- Constraint #3 prescribes the encapsulation of Hibernate exceptions in a specific type to decouple the Controller layer from the underlying persistence framework. To violate this constraint, we have removed the encapsulation of Hibernate exceptions, resulting in 44 violations. For each violation, our approach has triggered recommendation 4b, with the suggestion to remove the `throws` declaration and to surround the body of the method with a `try-catch` block. After following this recommendation, all violations have been removed.

- Constraint #4 prescribes the use of a particular module to handle job scheduling. To violate this constraint, we have intentionally implemented a simple schedule operation in a Struts action whose only responsibility is to handle HTTP requests and responses, resulting in 9 violations. For such violations, our approach has triggered recommendation 3b, suggesting to extract the statements and move them to a suitable class. The most adequate class returned by $\texttt{similarity}(\texttt{f}, \texttt{C})$ function was one inside the `SystemScheduling` module, whose similarity was `0.41`. By following this recommendation, all violations have been removed.

Threats to Validity: We have illustrated our approach using a single example, which mirrors the architecture and respective violations detected in a real-world system. Therefore, this preliminary illustration presents at least three threats. First, we cannot claim that our approach will provide equivalent results in systems following different architectures. Second, since we have considered a minimal set of restrictions, we cannot certify that our approach will provide useful recommendations for the whole spectrum of architectural constraints. Third, due to the controlled environment used in our example, we also cannot claim that our approach is effective in systems already facing a major architectural erosion process. However, we are currently working in a new evaluation scenario, including a large-scale and long-lived software system.

V. RELATED WORK

Recommendation Systems for Software Engineering (RSSEs) are an emerging research area [22]. For example, current RSSEs can recommend relevant source code fragments to help developers to use frameworks and APIs (Strathcona [23]), software artifacts that must be changed together (eRose [24]), and replacement methods for adapting code to a new library version (SemDiff [25]). However, we are not aware of recommendation systems whose precise goal is to help developers in tackling the architectural erosion process. For example, Tsantalis and Chatzigeorgiou have proposed a methodology to suggest Move Method refactoring opportunities [26]. Their general goal is to tackle coupling and cohesion anomalies manifested in the form of the Feature Envy bad smell. On the other hand, the refactoring problem we have investigated in this paper has a broader scope, since we target the elimination of architecture anomalies.

The relevance and challenges involved in the reconstruction of software architectures are well-documented in the literature. For example, Sarkar *et al.* report their experience in a two-year project involving the remodularization of a legacy banking application, which has more than 25 MLOC [6]. Although they have built some program analysis tools (such as tools to extract function-call information), they state that the refactoring step of the remodularization process has been completely manual. Rama and Patel [27] have analyzed several large software modularization projects in order to define recurring modularization operations, including module decomposition and union. However, they do not provide tool support for the proposed modularization operators.

VI. Further Work

The preliminary evaluation presented in Section IV has provided us with encouraging feedback about our refactoring recommendations. Nevertheless, in order to provide more robust arguments we are starting to apply the proposed approach to a real-world system. The analysis of a large-scale system will help us to improve our approach, possibly suggesting new recommendations not defined in this current work. We are also investigating recommendations to address violations in `must` constraints and we are extending `dclcheck` to incorporate our approach.

Acknowledgment

This research has been supported by grants from CAPES, CNPq, and FAPEMIG.

References

[1] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[2] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, pp. 105–119, 2002.

[3] M. Lindvall and D. Muthig, "Bridging the software architecture gap," *Computer*, vol. 41, no. 6, 2008.

[4] X. Dong and M. W. Godfrey, "Identifying architectural change patterns in object-oriented systems," in *16th IEEE International Conference on Program Comprehension (ICPC)*, 2008, pp. 33–42.

[5] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, p. 12.

[6] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a large-scale business application: A case study," *IEEE Software*, vol. 26, pp. 28–35, 2009.

[7] J. Borchers, "Invited talk: Reengineering from a practitioner's view – a personal lesson's learned assessment," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 1–2.

[8] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. Mendona., "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[9] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[10] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[11] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *3rd Symposium on Foundations of Software Engineering (FSE)*, 1995, pp. 18–28.

[12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.

[13] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[14] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[15] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 391–400.

[16] O. de Moor, "Keynote address: .QL for source code analysis," in *7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2007, pp. 3–14.

[17] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 167–176.

[18] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 279–294.

[19] Semmle Inc., "Semmle's on-demand analytics of software assets (ODASA)," http://semmle.com.

[20] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[21] R. Terra and M. T. Valente, "Towards a dependency constraint language to manage software architectures," in *Second European Conference on Software Architecture (ECSA)*, ser. Lecture Notes in Computer Science, vol. 5292. Springer, 2008, pp. 256–263.

[22] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010.

[23] R. Holmes, R. Walker, and G. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, 2006.

[24] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[25] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 481–490.

[26] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, pp. 347–367, 2009.

[27] G. M. Rama and N. Patel, "Software modularization operators," in *26th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.