One Graph to Rule Them All
Software Measurment and Management

Robert Dąbrowski, Krzysztof Stencel, and Grzegorz Timoszuk

Institute of Informatics
Warsaw University
Banacha 2, 02-097 Warsaw, Poland
{r.dabrowski,k.stencel,g.timoszuk}@mimuw.edu.pl

Abstract. For a software-intensive system, software architecture is typically defined as the fundamental organization of the system embodied in its components, their relationships to one another and to the system's environment, and the principles governing the system's design and evolution. In this paper we propose a unified approach to the problem of managing knowledge about the architecture of a software system. We postulate that only a holistic model that supports continuous integration and verification for all system artifacts is the one worth taking, and we formally define it. Then we demonstrate that our approach facilitates convenient project measurement. First we show how existing software metrics can be translated into the model in a way that is independent of the programming language. Next we introduce new metrics that cross the programming language boundaries and are easily implemented using our approach. Eventually we show that other concerns for architectural knowledge can be also dealt with using our approach. We conclude by demonstrating how the new model can be implemented using existing tools. In particular, graph databases are a convenient implementation of an architectural repository, and graph query languages and graph algorithms are an effective way to define metrics and specialized graph views.

Keywords: software measurement, architectural knowledge, software architecture

1 Introduction

Modern software projects are typically developed by multiple teams that are scattered around the world. Moreover the programmers frequently use a significant number of programming languages, sometimes even more than one dialect of a single language. Despite applying different software development methodologies, the project still face similar problems. First of all, when a project approaches certain complexity level, its documentation, models, unit tests etc. tend to become unsynchronized with its source code. This happens regardless to agile or formal development methodology. At the same time, the quality of source

code drops. Usually, the reason is that the architectural knowledge is stored separately from source code and is not revised. Changes in the source code are not reflected in the architectural documents (and vice versa). There are no links from the architectural decisions and restrictions to the source code. When issues arise (requests for change, bugs, code refactorings, library upgrades), the lack of traceability escalates the problems and further increases the number of issues. The change impact is hard to estimate and regression problems appear. Multiple models are created (e.g. to present architecture, to measure the quality, cost or reliability) and they habitually are not kept in sync with current state of the source code. Most of the problems are due to the significant cost and complexity of preserving the architectural model up-to-date.

The correct direction is the graph approach [8] and in this paper we take a step forward. First of all there is a need for a holistic solution that integrates all artifacts developed during a software project. By an artifact we understand everything that contains a part of the project knowledge. This includes code artifacts like classes or methods, as well as their dependencies from external libraries, as well as UML diagrams or use cases, as well as software configuration and description of development processes that define how project is organized and developed. It is worth noting that sometimes one artifact can carry knowledge from multiple areas. For example a change request has both architectural knowledge (depicts changes to be made) and management knowledge (impacts cost or project timeline). In the approach we propose all artifacts created during project development (source code, documentation, issues, metrics) are stored in one graph data structure. Each artifact is a vertex and can be described by multiple tags (or attributes). The vertices are connected by labeled edges depicting relations between artifacts. Two artifacts can be connected by multiple edges as they can be in more than one relation. The contributions of this paper is as follows: we define a graph structure to store the architectural knowledge of a project that allows to trace dependencies and manage changes and show that it allows to: 1) define known software metrics independently of programming languages; 2) define new cross-language metrics. The paper is organized as follows. In Section 2 we analyze the topic background and the related work that motivated our approach. In Section 3 we provide a definition of the graph-based model for architectural knowledge management. In Section 4 we present how current metrics can be translated into the new graph model and how the model improves them. Section 6 concludes our reasoning.

2 Motivation and related work

In coming years software engineers must research a new vision of software development process, as current visions do not keep up with the scale and pace of current software projects. Such a research will encompass developing both theoretical foundations and supporting tools. We postulate that in this new approach artifacts created in a software project are organized according to a consistent graph model [8]. In this paper we show how to translate existing software metrics

into the new graph model and how to track changes using graph approach. Our research follows existing work on software metrics and development models.

The need for quantitative assessment of software quality and software process predictability inspired measuring of source code and measuring of development process. At the beginning, metrics quantified the source code. Chidamber and Kemerer defined a set of metrics [6] that qualified if classes are well designed. It significantly influenced further research, especially LCOM (lack of cohesion of methods) has been studied extensively. LCOM tries to check if a class is following a single responsibility predicate. Otherwise it becomes hard to maintain and gets error prone. Henderson-Sellers, Constantine and Graham [11] defined newer versions of LCOM, addressing the existence of call chains. Eventually, Hitz and Montazeri [12] defined the most advanced version of LCOM (and the most expensive to compute).

Other cohesion measures, similar to LCOM, have been applied to other software artifacts. Emerson defined a cohesion metric for modules [10] that checks whether a module has a single responsibility. Additionally, Page-Jones defined several levels of cohesion [17] and described its desirable and undesirable types. Nowadays LCOM is applied to modern programming paradigms and a good example is measuring cohesion in Aspect Oriented Programming [9] presented by Zhao and Xu [21].

Another set of source code metrics is MOOD (metrics for object-oriented design). It have been defined by Abreu and Carapuça in [1] and further analyzed by Abreu, Goulão and Esteves in [2]. MOOD introduces metrics like: method or attribute hiding factor; method or attribute inheritance factor; coupling factor; clustering factor; reuse factor. The metrics have been revised by Baroni, Braz and Abreu in [4] where they formalized them using OCL (object constrain language) and proposed a framework for testing and comparing other coupling metrics.

The most popular metric defined in MOOD is the coupling factor. Its importance is increasing: modern systems usually use numerous libraries and domain sub-systems and unmanaged coupling makes classes difficult to maintain. Briand, Daly and Wüst presented a unified framework for measuring coupling in [5], they named and classified the sources of coupling and their impact on software, and presented many coupling metrics.

Another approach to measuring systems was proposed by Robert Marin in [14]. He introduced the term *software package* as a group of related classes. Some of the metrics defined at the level of software packages are: afferent couplings, efferent couplings, abstractness, instability. In the last twenty years numerous other code metrics have been defined, however only a few of those metrics are still in use and are general enough to work well with C++ as well as with modern languages like Java or Python.

Concurrently with software metrics, engineers also established requirements for proper metrics. Roche [18] and Abrieu with Carapuca [1] summarized principles of software measurement. The most important properties that a measure should posses are: having a formal definition; being system size independent;

being obtainable early in project life cycle; being down-scalable; being easily computable; and being independent from particular programming language.

Although the mathematical nature of metrics calls for their clear definitions, there still exist many contradicting definitions of the same metric depending on the implementation language. It has been suggested by Mens and Lanza [15] that metrics should be expressed and defined using a language-independent metamodel. This approach would allow for an unambiguous definition of generic object-oriented metrics and higher-order metrics. The authors have also indicated some prototype tools that implemented such idea.

3 Graph model

We recall definition of a model for software projects based on directed labeled multigraph [8]. According to the model the *project graph* is an ordered triple $(\mathcal{V}, \mathcal{L}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices that reflects all project artifacts. $\mathcal{L}$ is the set of labels which qualify vertices and edges. $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}$ is the set of directed labeled edges. There can be more than one edge from one vertex to another vertex, as artifacts can be in more than one relationship.

Vertices of the project graph are created when artifacts are produced during software development. A vertex can represent a part of the source code like: module, class, method, test (a unit test, an integration test, or a stress test). Other examples of vertices are metrics, documents (requirements, use cases, change requests), coding guidelines, source codes in higher level languages (i.e. yacc grammars), configuration files, build recipes, tickets in issue tracking systems and so on. Therefore, vertices may be of different granularities (densities). There is a special containment edge to connect vertices representing artifacts for different granularities. This can be presented on printed graphs as an arrow or using vertex nesting.

Each artifact is described by the set of labels with properties. A method can be described by general labels showing that: this is a part of project source code (*code*); it is written in Java programming language (*java*); its revision is 456 (*r:456*). There can be also some language specific labels, e.g. *abstract*, *public*. Edges have labels, too. One of the edge labels is the already mentioned *containment*. For example a package (bigger chunk) *contains* a class (smaller chunk), while a class *is a part of* a package. Some other important labels are: *calls*, *depends*, *generates*, *verifies*, *tests*, *defines*, *implements*.

Such model for a large project can be too large to be human-tractable as a whole. However, we are typically interested only in its parts that satisfy given restrictions. This is the reason for defining subgraphs as model views. Examples of such views are: the set of methods that call a given method; all public methods of a class (either including or excluding inherited ones). Queries that create such subgraphs are computationally inexpensive, as usually we need to traverse only a small fraction of the graph. More examples are given in Section 4.

4 Metrics

In our opinion, the holistic graph approach empowers development teams to understand and measure their projects better. Nowadays developers use numerous metrics to estimate quality of their products and to spot possible problems. A noteworthy number of tools has been created in order to automate the computation of software metrics. Unfortunately most of these tools are limited to only one programming language. Furthermore, they do not take into consideration the information stored outside of the source code, e.g. in XML configuration files. This makes their results incomplete, since in modern frameworks (e.g. Spring, Hibernate) a significant part of information is stored in configuration files. In addition, metrics serving different purposes (e.g. cost estimation/tracking and measuring quality) often use different models and need manual synchronization. Thus, their maintenance is too expensive.

4.1 Graph metrics are good metrics

The graph approach is in line with best practices for metrics [1, 18]. Graph metrics depend only on a graph's structure, thus they are language independent. Furthermore, they are well-defined, easily computable and available even at the beginning of a project. The translation of some OO metrics to graph models has been already analyzed [15], however only metrics based on the source code has been considered. In our research we take into account significantly more artefacts and their relationships. Since the formalism proposed in [15] is precise and easy to understand, we use it in our presentation.

Below we show how to express selected metrics in graph terms. Although this way the metrics become programming language independent, they still properly reflect the properties of the source code.

4.2 Chidamber and Kemerer metrics

We start with the first metric defined in [6]. The weighted method per class (WMC) is defined as:

$$WMC = \sum_{i=1}^{n} c_i$$

Where c_i is the complexity of the i-th method in the class. If each method has the same complexity, WMC is just the number of methods in the class.

In order to translate this metric into the graph model we start with the definition of the counting function NC. Let $n \in \mathcal{V}$ and $\eta_1 \in type(n)$. Then:

$$\begin{aligned} NC(n, \eta_1, \eta_2, \Phi, \eta_3) = \#\{m \in \mathcal{V} \mid type(m) \ni \eta_2 \wedge \Phi(m) \wedge \\ \exists e \in \mathcal{E} \colon source(e) = n \wedge target(e) = m \wedge type(e) = \eta_3\} \end{aligned}$$

NC counts the number of nodes m such that (1) the type of m belongs to the set η_2, (2) m satisfies the formula Φ, (3) there is an edge e of the type η_2 from n to m. The number of methods in a class c is easily defined as:

$$WMC(c) = NC(c, class, method, \Phi_{true}, contains)$$

NOC (the Number Of Children) is usually understood as the number of direct subclasses. Using the function NC we define NOC in the following way:

$$NOC(c) = NC(c, class, class, \Phi_{true}, inherits)$$

CBO (coupling between objects) has different versions by several authors. We will use the metric called Fan-out, that is often chosen as a CBO implementation. The Fan-out of a class c is the number of other classes that are referenced in c. A reference to another class a is a call to a method or an access to data member of a. In Fan-out multiple references to a class are counted as one. The graph definition of FAN-OUT(n) follows:

$$\begin{aligned}\text{FAN-OUT}(n) = \#\{m \in \mathcal{V} \mid (&contains(n,m) \wedge type(m) \ni class) \vee \\ (\exists_{o \in \mathcal{V}}\ &calls(n,o) \wedge contains(m,o) \wedge type(m) \ni class \wedge type(o) \ni method)\}\end{aligned}$$

The notation $contains(n, m)$ is a handy abbreviation of the formula:

$$\exists_{e \in \mathcal{E}}\ source(e) = n \wedge target(e) = m \wedge type(e) \ni contains$$

LCOM (the lack of cohesion of methods) was originally defined as follows: let c be the class and $\{M_i\}_{i=1}^n$ be the set of its methods and I_j be the set of instance class variables used by method M_i. Let $P = \#\{(I_i, I_j) \mid I_i \cap I_J = \emptyset\}$ and $Q = \#\{(I_i, I_j) \mid I_i \cap I_J \neq \emptyset\}$. Then:

$$LCOM(c) = \max(P - Q, 0)$$

In order to translate LCOM into graph terms we only need to express I_j in graph terms. Assume $j \in \mathcal{V}$ and $type(j) \ni method$. Then we define I_j as follows:

$$I_j = \#\{v \in \mathcal{V} \mid contains(j, v) \wedge type(v) \ni variable\}$$

Another ways of measuring cohesion were presented in [11]. To analyze them we need the number $m(c)$ of procedures in a class c, the number $a(c)$ of variables in a class c and the number of methods that access a variable i of a class c:

$$\begin{aligned} m(c) &= NC(c, class, method, \Phi_{true}, contains) \\ a(c) &= NC(c, class, variable, \Phi_{true}, contains) \\ mA(i,c) &= \#\{m \in \mathcal{V} \mid (contains(c,m) \wedge type(m) \ni method \wedge uses(m,i))\} \end{aligned}$$

Now we can define two improved versions of LCOM:

$$\begin{aligned} LCOM2(c) &= 1 - \frac{\sum_{i \in c} mA(i,c)}{m(c)a(c)} \\ LCOM3(c) &= \frac{m(c) - \sum_{i \in c} mA(i,c)/a(c)}{m(c) - 1} \end{aligned}$$

We can also introduce a purely graph-based metric of the cohesion of classes. Consider a graph of components of a class with the relationships *calling*, *called*, *using* and *used*. The value of the metric is the number of strongly connected components of this graph. A class is cohesive of this metric is 1. The computation of this metric is cheap as it can be done in linear time with respect to the size of the graph [7].

4.3 MOOD and MOOD2 metrics

Now we focus on quantitative class measurements. *MHF* (the method hiding factor) measures the degree of invisibility of methods in classes. The invisibility of a method is the fraction of all classes from which the method is not visible. The general intuition behind this measure is that most methods should be encapsulated within a class and not available for use by other objects. High method hiding factor boosts reusability of code and reduces its complexity. In line with the down scalability, this metric can be defined on multiple levels: the whole project, a module and a package.

Here we define it for package level. We assume that all but private methods are visible. For two vertices w and v, a vertex condition Φ_V and an edge condition Φ_E we define the predicate $path(w, v, \Phi_E, \Phi_V)$ to be true, iff there exists a directed path from w to v such that all its edges satisfies Φ_E and all its intermediate vertices fulfil Φ_V.

We can use this predicate to define $ALL_T(v, t)$ to be the number of nodes of the type t that are contained transitively in a vertex v and satisfy the condition T:

$$\begin{aligned} \Phi_{MC}(v) &= type(v) \ni package \vee type(v) \ni class \\ ALLSET_T(v, t) &= \{m \in \mathcal{V} \mid type(m) \ni t \wedge path(v, m, contains, \Phi_{MC} \wedge T)\} \\ ALL_T(v, t) &= \#ALLSET_T(v, t) \end{aligned}$$

Now we can define one MHF metric that is down scalable, i.e. it works for models, packages, packages in packages, etc.

$$MHF(p) = \frac{ALL_{isPrivate}(p, method)}{ALL_{true}(p, method)}$$

Nowadays the most popular metric from the *MOOD* set is *CF* (the coupling factor). The coupling is a call to other class's methods or an access to its variables. The coupling factor is defined as the ratio of actual coupling to maximum possible coupling. Higher coupling means higher complexity and lower maintainability. Additionally it reduces encapsulation and potential reuse. It also increases the number of potential defects. High coupling should be avoided. We propose the following graph-based definition of the coupling factor:

$$CF(p) = \frac{\#\{c, d \in (ALL_{true}(p, class) \mid accesses(c, d)\}}{0.5(ALL_{true}(p, class) - 1)ALL_{true}(p, class)}$$

The predicate $accesses(c, d)$ is true, iff c calls a method in d or accesses a variable in d.

4.4 Robert Martin's metrics

Robert Martin defined metrics that operate on packages, i.e. logical containers for source code artefacts. *NCF* (the number of classes and interfaces) in a package is one of the simplest metrics. Its definition is straightforward using ALL_T from the previous section:

$$NCF(p) = ALL_{true}(p, class) + ALL_{true}(p, interface)$$

AC (the afferent coupling) is the number of classes and interfaces from outside a package that depend on its classes. It can be defined in the graph-based style as follows. We subsequently define $CPC(p)$ (classes in the package p), $AAC(p)$ (classes and interfaces that are not in the package p) and finally $AC(p)$.

$$\begin{aligned} CPC(p) &= ALLSET_{true}(p, class) \\ AAC(p) &= (ALLSET_{true}(project, class) \cup ALLSET_{true}(project, interface)) \\ &\quad \setminus ALLSET_{true}(p, class) \\ AC(p) &= \#\{c \in AAC(p), d \in CPC(p) \mid accesses(c, d)\} \end{aligned}$$

Similarly *EC* (efferent couplings) is the number of classes and interfaces from outside of the package that are used by classes inside the package. *EC* can be defined in graph terms as follows:

$$EC(p) = \#\{c \in AAC(p), d \in CPC(p) \mid accesses(d, c)\}$$

Using *AC* and *EC* it is possible to define *instability* measure:

$$I(p) = \frac{EC(p)}{EC(p) + AC(p)}$$

$I(p)$ estimates the resilience to change of the package p.

The *package abstractness* is the fraction of abstract classes and interfaces in a package. In graph terms we can define it as follows:

$$A(p) = \frac{ALL_{true}(p, interface) + ALL_{isAbstract}(p, class)}{ALL_{true}(p, interface) + ALL_{true}(p, class)}.$$

DMS is the distance from the main sequence. The main sequence is the idealized line $A(p) + I(p) = 1$. DMS is the normalized distance to this line:

$$DMS(p) = \frac{\mid A(p) - I(p) - 1 \mid}{\sqrt{2}}$$

4.5 New metrics

One place to store and integrate whole architectural knowledge facilitates tracing not only changes in the source code but also changes of the documentation and meta-models. This opportunity gives raise to new graph-defined metrics concerned with software processes.

Automatic validation ratio As mentioned above even partial automatic model translation is extremely hard to achieve in practice. On the other hand, validation if all project artefacts are in sync is much easier when all of them are stored in one place. If there is one such place, the development of validators will also be easier.

Consider the typical problem of synchronization between a UML model and a Java code. In real life projects this task is too complex. Even in not so big projects like JLoXiM [19] it has turned out to be impossible. Building a synchronization tool that keeps in sync these two models is a difficult problem since UML models do not contain all information about classes. If a UML model contained all the information, each class definition would have too many dependencies and the architect would not be able to ever finish modelling.

We propose to introduce a reliability measure AVR (the automatic validation ratio) that is based on the percentage of automatic validations. A path from a requirement to the implementation is the subject of this measurement. Such a path usually contains few nodes that specify more implementation details and architectural decisions. AVR is the percentage of automatic validation vertices on a path. In an ideal case it would be 100% but it is hard to achieve, if the language of business analytical documents is not formal. This measure is scalable as it can be computed for a single path, for all paths having last vertex in a given package, a module or in the whole project. Additionally, it is language independent as it only operates on graph vertices. Moreover, it is available from the very beginning of the project.

Definition ratio Intuitively everybody understands that there should be clear path from requirements to the source code and backwards. On the other hand, this rule is often neglected. Thus, it is hard to track changes and their impact, especially in software documents and metamodels. With the graph approach we can easily model even abstract artefacts and define dependencies between them. Therefore, to check if the process is complete we check whether there is a path from requirement documents to interfaces and public methods.

The reverse operation of tracking if all interfaces and public methods have paths to requirements and models (e.g. UML diagrams) can be easily performed on the project graph. We propose to use DF (definition ratio) as the fraction of interfaces and public methods that have paths to documentation. A high definition rate means that our code is not exposing unnecessary methods. This reduces the coupling and thus increases the maintainability of the code.

One graph to rule them all AVR and DF advocated in this section are facilitated by the holistic graph of all software artefacts. Without such a repository, it seems impossible to use such high-level project metrics.

5 Implementation

A pilot version of our graph approach to measuring software projects has been already implemented. Standard relational databases proved to be not suitable

for storing graphs, so it has been build on Neo4J [20] graph database. The engine fit our needs as: its vertex contains a map of properties; every edge has a type (besides of a map of parameters) which can be used in graph traversal; definition like ALL_T are defined as *TraversalDescriptions* hence can be reused on multiple vertices. Additional reasons for choosing Neo4J databases were: good integration with Lucene full text indexer, which can improve significantly typical searches; and Gremlin graph querying language. Currently we are working on production version that will offer possibility to import multiple projects and allows us to run more experiments, like comparing metrics results evaluated by tools working on Java source code (like JDepend) with ones achieved in our graph-based environment.

6 Conclusion

We follow the research on architecture of software [13] and software process. We support the approach that incorporates in one model both software and software process artefacts, as the only one worth taking [16]. An implementation of such a model is possible using graph databases [3] as the storage layer for artefact representation. Graph querying languages and graph algorithms allow to easily define measures, define graph views and find information in graph.

This paper can be summarized as follows: software artefacts and software development process artefacts created during a software project are organized as vertices of a graph connected by edges that represent multiple kinds of dependencies among those artefacts. Quality assurance of software and predictability of software development process are supported by metrics that are easily expressed and calculated in graph terms.

We are aware that the concept is not an entirely novel one, rather it should be perceived as an attempt to support existing trends with a sound and common foundation. We hope that many software practitioners perceive such graph-oriented approach as a next step in evolution of the level of integration of software's architectural knowledge. In our opinion actual software projects continue to suffer from the lack of visible, detailed and complete setting to govern their architecture and evolution, despite using many advanced tools that are currently available. Our model fills this gap and as it integrates in one place areas that are presently managed independently.

If the graph-based approach gains attraction and recognition, it will establish common grounds for the integration of currently used concepts, methodologies and supporting tools and open an interesting opportunity for a new future consistent software development methodologies. Obviously this is the beginning of a road to such a methodology, and the theoretical foundations, the range of supporting tools, and the extension of its systematic evaluation call for a significant amount of further research from the software engineering community.

References

1. F. Abreu and R. Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th International Conference on Software Quality*, 1994.
2. F. Abreu, M. Goulão, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA*, pages 44–57, 1995.
3. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40:1:1–1:39, February 2008.
4. A. Baroni, S. Braz, and F. Abreu. Using OCL to formalize object-oriented design metrics definitions. *Proceedings of Quantitative Approaches in Object-Oriented Software Engineering*, 2002.
5. L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25:91–121, January 1999.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.
7. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
8. R. Dabrowski, K. Stencel, and G. Timoszuk. Software is a directed multigraph. In I. Crnkovic, V. Gruhn, and M. Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 360–369. Springer, 2011.
9. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44:29–32, October 2001.
10. T. J. Emerson. A discriminant metric for module cohesion. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 294–303, Piscataway, NJ, USA, 1984. IEEE Press.
11. B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.
12. M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, volume 50, pages 75–76, 1995.
13. P. Kruchten, P. Lago, H. van Vliet, and T. Wolf. Building up and exploiting architectural knowledge. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 291–292, Washington, DC, USA, 2005. IEEE Computer Society.
14. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
15. T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. In *Electronic Notes in Theoretical Computer Science*, volume 72, pages 57–68, 2002. GraBaTs 2002, Graph-Based Tools (First International Conference on Graph Transformation).
16. L. Osterweil. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
17. M. Page-Jones. *The practical guide to structured systems design: 2nd edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

18. J. M. Roche. Software metrics and measurement principles. *SIGSOFT Softw. Eng. Notes*, 19:77–85, January 1994.
19. P. Tabor and K. Stencel. Stream Execution of Object Queries. *Grid and Distributed Computing, Control and Automation*, 121:167–176, 2010.
20. C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.
21. J. Zhao and B. Xu. Measuring aspect cohesion. In *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2004.