

PREPRINT VERSION

Using Modularity Metrics to assist Move Method Refactoring of Large Systems

C. Napoli, G. Pappalardo, and E. Tramontana

PUBLISHED ON: 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), pp. 529-534, 2013

BIBITEX:

```
@inproceedings{Napoli20132013,  
author={Napoli, C. and Pappalardo, G. and Tramontana E.},  
booktitle={2013 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)},  
title={Using Modularity Metrics to assist Move Method Refactoring of Large Systems},  
year={2013},  
pages={529-534},  
publisher={IEEE},  
doi={10.1109/CISIS.2013.96},  
}
```

Published version copyright © 2013 IEEE

UPLOADED UNDER SELF-ARCHIVING POLICIES
NO COPYRIGHT INFRINGEMENT INTENDED

Using Modularity Metrics to assist Move Method Refactoring of Large Systems

Christian Napoli, Giuseppe Pappalardo, Emiliano Tramontana

Dipartimento di Matematica e Informatica

University of Catania, Italy

Email: {napoli, pappalardo, tramontana}@dmi.unict.it

For large software systems, refactoring activities can be a challenging task, since for keeping component complexity under control the overall architecture as well as many details of each component have to be considered. Product metrics are therefore often used to quantify several parameters related to the modularity of a software system.

This paper devises an approach for automatically suggesting refactoring opportunities on large software systems. We show that by assessing metrics for all components, move methods refactoring can be suggested in such a way to improve modularity of several components at once, without hindering any other. However, computing metrics for large software systems, comprising thousands of classes or more, can be a time consuming task when performed on a single CPU. For this, we propose a solution that computes metrics by resorting to GPU, hence greatly shortening computation time.

Thanks to our approach precise knowledge on several properties of the system can be continuously gathered while the system evolves, hence assisting developers to quickly assess several solutions for reducing modularity issues.

Index Terms—software engineering; metric; refactoring; GPU;

I. INTRODUCTION

For assessing and enhancing the modularity of existing object-oriented systems, several techniques have been proposed [1], [2], [3]. Refactoring techniques, e.g. extract snippets of code from a method, move a method to a different class, etc., are a fundamental support for improving the cohesion of a class, reducing coupling between classes, shortening long methods/classes, etc. [2], [4]. Accordingly, *code smells*, i.e. characteristics indicating that “*the code has to be changed*” [2], have been proposed to guide refactoring activities. Moreover, several well-known *metrics* have been proposed to assess the characteristics of software systems and to guide refactoring [5], [6], [7], [8]. Indeed, using several metrics at once, for a large software system, with each metric assessing a different facet, can give the developer a better understanding of modularity and quality characteristics.

In our view, refactoring should help to meet more than one modularity need and assessing multiple metrics at once would be essential when trying to balance several “*forces*” acting into a software system.

For *crosscutting concerns*, i.e. concerns that have been implemented by spreading the correspondent code across

several modules, refactoring can make good use of *aspect-oriented programming* (AOP) [9]. Refactoring to aspects has been proposed in order to extract method calls, conditional statements, etc. [10], [11] or in order to have separation between domain and pattern-related code [12] or non-functional requirements [13], [14]. Accordingly, some metrics have been proposed to evaluate how concerns spread over modules (i.e. classes or aspects) [16], [17].

This paper aims at tackling two issues related to the suggestion of move method refactoring opportunities. Firstly, automatically finding refactoring suggestions that improve several components at once, which becomes possible by computing several metrics. E.g. cohesion for a pair of components is improved at once by carefully selecting a single move method refactoring, without hindering other characteristics of the same pair and without negatively affecting other components. How a change affects components is unclear when having to cope with a large number of them. Automatic suggestion of refactoring opportunities is of great importance for large systems, since the unassisted developer, having to reason on thousands of classes, could miss the proper refactoring. Moreover, the number of ways in which *perfective* changes can be introduced is dramatically increased in such large systems. Manual exploration of such changes would be cumbersome or time consuming.

Secondly, computing metrics should take a tiny amount of time, when a software system consists of a large number of methods, attributes, and classes, i.e. in the order of several thousands, to be considered a useful indication to the developer while she explores refactoring opportunities. For this, we have devised a parallel algorithm that runs on a GPU to compute time consuming product metrics and we have greatly reduced, even by a factor of 50, the typical CPU computing time needed. A GPU provides hundreds of computing cores, whereas a CPU provides a few (typically 8). To effectively employ hundreds of cores, our solution let threads run without synchronisation as much as possible.

The remaining part of the paper is organised as follows. Section II introduces relevant metrics assessing the modularity of software systems. Section III proposes how to combine metrics for suggesting the most appropriate refactoring and how computationally costly this can be. Section IV describes our solution for collecting data on systems, as well as for computing metrics on GPUs. Section V reports the results

of our experiments on some real-world systems. Finally, concluding remarks are drawn in Section VI.

II. MODULARITY METRICS

Among the most useful modularity metrics, in our approach we compute the following ones.

A. Structural metrics

Henry and Kafura proposed *Fan-in* and *Fan-out* for procedural programming [5], however these are also used for object-oriented systems. *Fan-in* counts for a method m the number of methods calling m . A method having a high value of fan-in encloses a fragment of code that has been reused many times within the system itself. For this, a change on it could trigger many changes on parts of the system relying on it, i.e. suffer of the *ripple effect*. A method with high fan-in should be carefully tested to ensure that calling methods rely on its correct behaviour. High fan-in values can indicate crosscutting concerns, and accordingly method calls could be extracted and implemented as aspects [11].

Fan-out counts for a method m the number of methods called by m . A method with a high value of fan-out can be considered complex and having too many responsibilities. Hence, it is difficult to reuse, because it depends on a large number of methods. A method exhibiting a large fan-out, having a large amount of dependencies, should be split into several methods, each constituting a lower-level abstraction for the caller [1], [2], [18].

For a pair of entities, *Jaccard similarity coefficient* measures the ratio between the intersection cardinality of the sets of properties for each entity, and the union cardinality of the two sets [8]. For a method, its properties can be defined as called methods and accessed attributes. Hence, the similarity for a pair of methods gives a measure of how close they should be, i.e. whether it is appropriate to have such methods on the same class or not [19], [8]. Methods pairs with high similarity are better handled on the same class. This is because, in general, a high degree of intra-class interactions suggests high cohesion for the class, whereas for a pair of methods belonging to different classes, the high degree of coupling hinders modularity.

B. Object-oriented metrics

Chidamber and Kemerer have introduced a widely used suite of metrics that provides several parameters for assessing the modularity of object-oriented systems [6]. The well-known suite comprises *WMC*, *NOC*, *DIT*, *CBO*, *RFC* and *LCOM*. In this paper we focus on *CBO* and *LCOM*.

CBO measures *coupling between objects* by counting for a class C the number of other classes used by C , hence method calls or instantiations of classes, etc. performed by C . Similarly to *fan-out*, high values of *CBO* indicate complexity of classes, i.e. too many dependencies, and as such it would be better to refactor.

LCOM measures *lack of cohesion on methods* of a class C and has been originally defined as the number of method

pairs sharing no attributes minus the number of method pairs that share at least one attribute. Later, it has been proposed to measure *LCOM* as 1 minus the average, among methods of C , of the ratio between the number of attributes of C used by each method and the total number of attributes of C [20]. For a class with a high value of *LCOM*, its methods have been incidentally put together, hence some methods should be moved to another class, or some attributes should be moved into this class.

III. APPROACH FOR REFACTORING

A. How refactoring is suggested

By evaluating the above metrics simultaneously, we advocate that the modularity of an object-oriented system can be improved by means of the following indications.

When the similarity between a pair of methods (m_1, m_2) is high¹ and the methods happen to be on different classes, we check how *LCOM* values for classes varies when moving one method from its origin class to the class of the other method on the pair, or to other classes holding one of the called methods. Then, we suggest to move a method, i.e. m_1 or m_2 , from its origin class to another class when we find that *LCOM* of both the origin and destination classes will be improved.

When *LCOM* for a class is high, hence methods happen to incidentally be on such a class, then an improvement is given by taking out one (or more) method from the class and find another class for it. Tentative destination classes for a method will be the ones holding methods called (or attribute accessed) by the method to be moved. Methods with the highest values of fan-out will be selected as methods to be moved out from the origin class. The class suggested as a destination for a method will be the one whose *LCOM* will be lowered, while also lowering *LCOM* of the origin class. Note that, a desired side-effect is that *CBO* of the origin class could be lowered by such a suggested refactoring.

When *CBO* for a class is high, we check possible methods relocations, starting with the ones having the highest fan-in and fan-out, to another class. Candidate destination classes are all the classes used by the method to be moved. *CBO* will be computed for candidate destination classes, for each possible relocation. The proposed refactoring selects as a destination for a method, the class whose *CBO* will be not higher than its original value, while of course the *CBO* of the origin class will be lowered. The granularity of the desired improvement can be decided by setting a threshold for the desired *CBO* (e.g. the threshold could be set as the average value of *CBO* for all classes).

For large software systems, computing metrics for all the tentative methods moves is time consuming unless appropriate high performance resources are employed.

B. How computing metrics scales

This section estimates the number of metric values computed when using the above approach to suggest refactoring

¹Let us say that a threshold is chosen for the similarity value as e.g. the average among all similarity values for the whole system.

opportunities. Firstly, fan-in and fan-out will be computed for every method. Suppose m is the number of methods, then the number of values for each of the two metrics is the number of available methods m .

Note that, as far as complexity is concerned, for computing fan-in for a method m_p , all other $m-1$ methods will have to be scanned, and each invoked method will have to be compared with method m_p . Let us suppose that maximum k_m methods are called by each method, then the worst case is that $k_m \cdot (m-1)$ comparisons have to be performed.

Secondly, similarity will be computed on all possible combinations of methods pairs. From combinatorics, given m methods, then the number of all pairs, without considering the ordering, is given by

$$\#\text{Similarity values} = \binom{m}{2} = \frac{m \cdot (m-1)}{2}$$

From an analysis likewise the one above would come out that for a method pair the number of comparisons to be performed are $(k_m + k_a)^2$ in the worst case, where k_a is the maximum number of attributes accessed by each method. Computing similarity *sequentially* for thousands of methods can be time consuming, since the number of pairs quickly grows with the square of the number of methods.

Thirdly, LCOM and CBO will be computed for each class, as well as on each candidate class that could receive or give up a method. For c classes, and m methods, the worst case is to assess LCOM and CBO after having moved into each class every method, as well as having moved out a method from each class. The number of LCOM values is

$$\#\text{LCOM values} = c + c \cdot m = c \cdot (m+1)$$

For thousands of methods and classes the said number of values grows as $c \cdot m$, hence computing it *sequentially* can be time consuming. The number of CBO values is the same as the above number of LCOM values, in the worst case.

Therefore, for the whole set of metrics presented above, we have that the number of values to compute is

$$\#\text{values} = 2 \cdot m + \frac{m \cdot (m-1)}{2} + 2 \cdot c \cdot (m+1)$$

The second amount of the sum grows quicker than the others (while the third grows quicker than the first) for increasing values of m . Note also that, generally, m grows quicker than c , for practical software systems.

Let us consider a software system having 1,200 methods and 231 classes, this is the case for JUnit, which is considered a small software system. Then, the number of values to compute are 1.2 *millions*. Section V shows the number of classes and methods, as well as measured computing times, for several real-world software systems.

IV. ANALYSING SOFTWARE SYSTEMS

A. Overview

Figure 1 provides an overview of the architecture for a tool computing the above metrics. Firstly, the system under analysis will be explored and for this several filters are needed to recognise the characteristics of the system. Once exploration

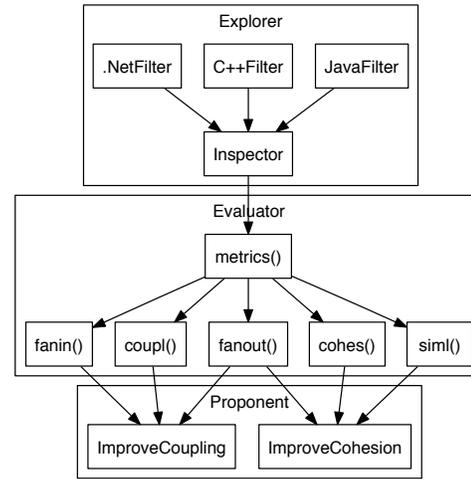


Fig. 1. Main components of the tool for the proposed approach

has been performed, metrics will be computed, according to extracted characteristics and the definitions of metrics. Finally, metrics are used together to suggest some move method refactoring. Each refactoring could improve a subset of metrics only, hence a desired characteristic, such as e.g. cohesion or coupling. The developer could select the characteristic to improve or ask for refactoring suggestions that improve all of them. In the latter case, a smaller subset of suggestions would be given, i.e. the intersection of refactoring suggestions.

The following subsections provide details of both the exploration of the system and metric computation on a GPU, whereas the logic for selecting a refactoring has been described in Section III.

B. Exploring the system under analysis

Metrics described in Section II are mainly based on the identification of attributes, methods and classes of a software system, as well as the number of method calls and attribute accesses from each method of every class. In order to compute the above metrics, firstly such data will have to be gathered. Several approaches can be followed for pursuing this *exploration* task, e.g. computational reflection, as in [3], or supporting libraries. For our experiments, we have used the Bytecode Engineering Library (BCEL) for gathering data on the bytecode of a Java software system, hence greatly simplifying the exploration phase [21]². BCEL can be considered as representing the JavaFilter block shown in Figure 1. Other filters can be built as desired for different languages or executable code.

We have then built an Inspector that gathers all the classes names of a software system, and for each class analyses the method bodies. Firstly, classes names, attributes and methods names have been found and stored into a list. For attributes and methods the name of the class they belong to has been stored. Moreover, methods names have been stored together with their input parameters, in order to properly consider overloaded methods.

²<http://commons.apache.org/bcel/>

Every method body has been analysed and method calls and attribute accesses have been stored into another list. Such a list can be navigated when accessing a method (on the previous list) in order to have all the called methods and accessed attributes. This is the main part of *Inspector*, finding dependencies between classes and between methods.

C. Computing metrics on a GPU

In CUDA programming model, an application consists of a *host* program that executes on the CPU and other parallel *kernel* programs executing on the GPU [22]. A *kernel* program is executed by a set of parallel threads. The *host* program can dynamically allocate device *global* memory to the GPU and copy data to (and from) such a memory from (and to) the memory on the CPU. Moreover, the *host* program can dynamically set the number of threads that run on a *kernel* program. Threads are organised in blocks, and each block has its own *shared* memory, which can be accessed only by each thread on the same block.

For maximum performances, threads on a GPU should ideally be given a task that can run unconstrained, i.e. without having to synchronise with others [23]. Moreover, it is paramount that interactions between CPU and GPU are minimised, this avoids communication bottlenecks and delays due to data transfers.

Following such general guidelines, we have developed a program, realising block Evaluator in Figure 1, that takes as input the list of classes, their methods and attributes, of the software system to be analysed. For handling the minimum possible amount of data, classes, methods and attributes have been represented by a numerical id. We have used arrays for holding several ids, since they can be easily and efficiently passed to the GPU, i.e. the *host* program allocates memory and transfers data to such a memory, which CUDA *kernel* program can use. For the classes, array `cls` holds the ids of each class, as well as the corresponding ids of methods and attributes. For methods, an array `dpnd` holds for each method its *dependencies* in terms of the ids of methods invoked and attributes accessed.

By calling the standard `cudaMemcpy()` function, arrays `cls` and `dpnd` are passed to the GPU memory, hence they become available to our provided *kernel* global function `metrics()`. This calls other *kernel* device functions, each computing one of the different metrics used. Among the said arrays, only appropriate ones are given to the function specialised for computing one of the metrics. The values of arrays are read by each thread, however threads need not write any value on the arrays, hence no synchronisation has been used for accesses.

For the function computing the similarity metric, `siml()`, each available thread is given a range of method ids, representing a subset of all the available methods to be analysed. For the given range of ids, a thread executing inside our function `siml()` computes all the similarity values between one method and all the other methods, by reading values from array `dpnd`, providing data representing dependencies. The selection of the range of methods to be given to a thread is

easily determined by the maximum number of methods available and `ThreadId`, available in CUDA programs, indicating the current working thread.

Each thread stores results into its own local array, i.e. separately from other threads, hence minimising the need of synchronisation. Given the large amount method pairs (see the above analysis III-B, and the the analysed systems V), only meaningful values are stored, i.e. only similarity values that are greater than zero (otherwise we risk filling up all available memory). Once a thread has finished executing, it will have computed and stored a given amount of results, which likely differs in number from that of other threads. This is because each thread will find a different number of zeros as a result. The meaningful values will have to be stored on a globally accessed array, so that other functions on the device can use them. For this, each thread reserves an amount of locations to store its computed values. Reservation has been performed by updating a global variable, shared by threads, hence by using the `atomicAdd()` function. This is the only moment for threads to synchronise with each other.

For metric LCOM, our function `cohes()` is given a range of ids for classes, hence each thread computes values of LCOM for a subset of classes by reading values from arrays `cls` and `dpnd`. Like the previous function, `siml()`, each thread stores the non zero value of LCOM for a class locally, before processing the following class. Once all classes have been processed, relevant results are transferred to the global array by resorting to a synchronous update of a shared variable. Similarly, functions have been implemented for each other metric, i.e. fan-in, fan-out, and CBO.

For assessing the benefits of methods moves refactoring, candidate classes to be changed, both as an origin or a destination class for a moved method, will have to be examined and their relevant changing values, i.e. LCOM and CBO computed again. This is performed by changing the representation of the methods belonging to a class, and then by repeating execution within the said functions.

V. EVALUATION

Table I provides values of several structure characteristics and the computed the metrics for several software systems, ranging from small to medium size systems. The size of analysed systems is given as the number of lines of code (LOC) as well as the number of lines no comment no blank (NCNB). The number of total classes is shown along with the number of attributes and methods for each class. Column `#values` shows the amount of values for the several metrics that have to be computed, according to the analysis given in Section III. It can be seen that such values grow to the order of thousands of millions.

The three columns *execution time* of Table I show measured wall times when computing metrics for improving cohesion possibly for all classes, i.e. values of similarity, LCOM, and again LCOM for classes involved into move method opportunities, as in block `ImproveCohesion` within `Proponent` in Figure 1. The wall times refer to a single CPU and on a Tesla GPU, with 448 cores, and their ratio. As we can

TABLE I
ANALYSED SOFTWARE SYSTEMS

Systems	LOC	NCNB	Classes	Attributes	Methods	#values	execution time			#move methods
							CPU	Tesla	ratio	
1 JUnit	30K	22K	231	265	1,200	1.2M	0.10	0.07	1.53	2
2 JHotDraw	72K	28K	600	1,151	4,814	17.4M	1.61	0.30	5.43	26
3 JavaStyle	69K	26K	600	1,423	6,816	31.4M	5.55	0.24	22.74	54
4 Hammurapi	80K	30K	986	2,595	7,705	44.9M	9.73	0.24	40.54	167
5 Dependometer	75K	32K	907	2,932	7,858	45.1M	7.43	0.41	18.03	632
6 MapperXML	42K	16K	1,146	2,726	8,074	51.1M	10.26	0.31	33.31	687
7 JEdit	183K	77K	1,267	3,804	9,629	70.8M	13.31	0.56	23.93	309
8 Commons-math	276K	115K	1,930	4,196	13,676	146.3M	30.54	1.10	27.86	367
9 Weka	529K	206K	2,138	9,194	22,028	336.8M	80.79	1.63	46.62	435
10 JRefractory	302K	120K	2,775	6,053	23,639	410.6M	66.10	1.65	40.01	864
11 Derby	1174K	403K	3,191	13,900	44,394	1,268.8M	256.51	4.68	54.81	887
12 Libomv	195K	73K	7,134	14,211	43,593	1,572.2M	218.45	4.80	45.51	438
13 ProjectLibre	465K	174K	6,399	28,444	69,751	3,325.4M	569.75	12.94	44.04	2,358

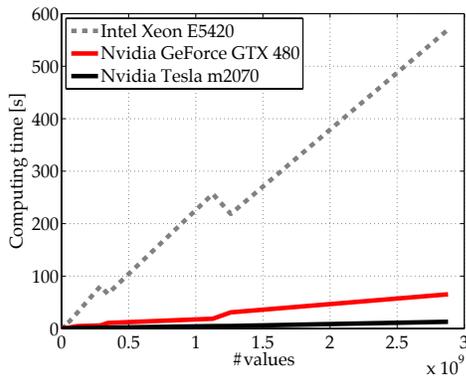


Fig. 2. Measured computing times for analysed systems

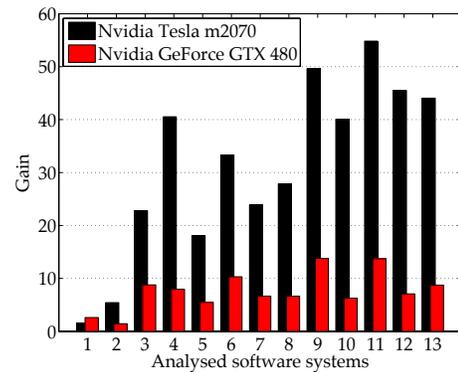


Fig. 3. Gain ratio between GPUs and CPU computing times

see also from Figure 2, computing times go from minutes to seconds for larger systems passing from a CPU to a GPU (from 11 min to 13 secs for the largest system). The plot comprises computing times needed with both a Tesla and a GeForce GPUs. Note that when comparing the gain CPU over Tesla and the gain CPU over GeForce, the latter is smaller, however still very significant (and with a fraction of the price for the hardware needed). The gain in performance shows important improvements ranging from a 1.5 gain for a small system to a 54.8 gain for one of the largest systems analysed (see Figure 3). The gain that we have obtained is in good agreement with previous assessments of other programs, when we compare an appropriate solution using GPU resources with an optimised solution on a CPU [24].

Finally, the last column in Table I shows the number of move methods refactoring that have been automatically suggested by our tool for each software system under analysis when considering move methods refactoring that optimise LCOM values only.

From the number of suggestions, reaching more than 2 thousand for the largest system, it is possible to conclude that the high-quality systems analysed can be actually further improved thanks to the selection of move method opportunities proposed by our approach. By enhancing modularity, we make systems more prone to incorporate other functional and non-functional requirements. E.g. less interactions between

components could facilitate consistent runtime updates [25].

VI. CONCLUSIONS

This paper has proposed several criteria for suggesting move methods refactoring opportunities that aim at improving several values of modularity metrics for a software system. It has been shown that for obtaining a proper view on the effect of changes for a large software system the number of values to be calculated grows very quickly. This can become overwhelming for a single CPU, hence a GPU can be purposely employed. It is also important to notice that refactoring opportunities can be more difficult to assess for the unassisted developer when having to reason on a large system. Hence, a tool that readily checks thousands of millions of values is of great help. Experiments have shown that existing large systems can be further

ACKNOWLEDGMENT

This work has been supported by project Infinity Web Edition funded within POR FESR Sicilia 2007-2013 framework, and project PRISMA PON04a2 A/F funded by the Italian Ministry of University.

This paper has been published in the final and reviewed version on **7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), pp. 529-534, 2013** [26].

REFERENCES

- [1] E. Gamma, R. Helm, R. Johnson, and R. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [3] G. Pappalardo and E. Tramontana, “Automatically discovering design patterns and assessing concern separations for applications,” in *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2006, pp. 1591–1596.
- [4] J. Kerievsky, *Refactoring to patterns*. Addison-Wesley, 2005.
- [5] S. Henry and K. Kafura, “Software structure metrics based on information flow,” *IEEE Transactions on Software Engineering*, vol. 5, no. 7, pp. 510–518, 1981.
- [6] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994.
- [7] S. Sarkar, A. C. Kak, and G. M. Rama, “Metrics for measuring the quality of modularization of large-scale object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 700–720, 2008.
- [8] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loningtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241, 1997.
- [10] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, “Tool-supported refactoring of existing object-oriented code into aspects,” *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 698–717, 2006.
- [11] M. Marin, A. van Deursen, and L. Moonen, “Identifying crosscutting concerns using fan-in analysis,” *ACM Trans. on Software Engin. and Methodology*, vol. 17, no. 1, pp. 1–37, Dec. 2007.
- [12] R. Giunta, G. Pappalardo, and E. Tramontana, “AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns,” in *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2012, pp. 1243–1250.
- [13] R. Giunta, F. Messina, G. Pappalardo, and E. Tramontana, “Augmenting a web server with QoS by means of an aspect-oriented architecture,” in *Proceedings of Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2012, pp. 179–184.
- [14] —, “Kaqudai: a dependable web infrastructure made out of existing components,” in *Proceedings of Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2013.
- [15] S. Apel and D. Batory, “How aspectj is used: an analysis of eleven aspectj programs,” *Journal of Object Technology (JOT)*, vol. 9, no. 1, pp. 117–142, 2010.
- [16] A. Di Stefano, M. Fargetta, G. Pappalardo, and E. Tramontana, “Metrics for Evaluating Concern Separation and Composition,” in *Proceedings of Symposium on Applied Computing (SAC)*. ACM, March 2005.
- [17] U. Kulesza, C. Sant’Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena, “Quantifying the effects of aspect-oriented programming: A maintenance study,” in *Proceedings of International Conference on Software Maintenance (ICSM)*. IEEE, 2006.
- [18] F. Messina, G. Pappalardo, and E. Tramontana, “Design and evaluation of a high-level grid communication infrastructure,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 9, pp. 1299–1316, 2007.
- [19] F. Simon, F. Steinbrückner, and C. Lewerentz, “Metrics based refactoring,” in *Proceedings of Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2001.
- [20] L. C. Briand, J. W. Daly, and J. Wüst, “A unified framework for cohesion measurement in object-oriented systems,” *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.
- [21] M. Dahm, *Byte Code Engineering with the Java Class API*. Freie Univ., Fachbereich Mathematik und Informatik, 1998.
- [22] “NVIDIA CUDA compute unified device architecture programming guide,” <http://www.nvidia.com>.
- [23] H. Nguyen, *GPU Gems 3*. Addison-Wesley, 2008.
- [24] V. W. Lee *et al.*, “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*. ACM, 2010.
- [25] F. Bannò, D. Marletta, G. Pappalardo, and E. Tramontana, “Handling consistent dynamic updates on distributed systems,” in *Proceedings of Symposium on Computers and Communications (ISCC)*. IEEE, 2010, pp. 471–476.
- [26] C. Napoli, G. Pappalardo, and E. Tramontana, “Using modularity metrics to assist move method refactoring of large systems,” in *2013 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*. IEEE, 2013, pp. 529–534.