Software (re)modularization: Fight against the structure erosion and migration preparation

N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet
RMoD INRIA
stephane.ducasse@inria.fr

R. Ducournau, R. Giroudeau, M. Huchard, J.C. König, D. Seriai
LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
huchard@lirmm.fr

I. Context

Software systems, and in particular, Object-Oriented systems are models of the real world that manipulate representations of its entities through models of its processes. The real world is not static: new laws are created, concurrents offer new functionalities, users have renewed expectation toward what a computer should offer them, memory constraints are added, etc. As a result, software systems must be continuously updated or face the risk of becoming gradually out-dated and irrelevant [34]. In the meantime, details and multiple abstraction levels result in a high level of complexity, and completely analyzing real software systems is impractical. For example, the Windows operating system consists of more than 60 millions lines of code (500,000 pages printed double-face, about 16 times the Encyclopedia Universalis). Maintaining such large applications is a trade-off between having to change a model that nobody can understand in details and limiting the impact of possible changes. Beyond maintenance, a good structure gives to the software systems good qualities for migration towards modern paradigms as web services or components, and the problem of architecture extraction is very close to the classical remodularization problem.

II. A significant problem

Most of the effort while developing and maintaining a software system is spent in supporting its evolution [50]. It is well-known that up to 80% of the total cost of software development project is spent in maintenance and evolution of existing applications [15], [19]. Large IT applications including applications running central and critical business (*e.g.,* command tracking, banking, railway) have to run and evolve over decades.

Such situations are crystallized in the following two laws of Lehman and Belady. These laws, known as the laws of software evolution, stress the fact that software *must* continuously evolve to stay useful and that this evolution is accompanied by an increase of complexity.

> **Continuous Changes.** *"an E-type program—i.e., a software system that solves a problem or implements a computer application in the real world—that is used must be continually adapted else it becomes progressively less satisfactory"* [34]
>
> **Increasing Complexity.** *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*[34]

From a business perspective, maintenance is mandatory and vital. It is worth to note that the Year 2000 Bug was also revealing some business occasions. For example in France SOPRA which was mainly an SSII, has since the year 2000 developed its TMA (Tierce Maintenance Applicative) branch to get into the maintenance business. SOPRA TMA now represents 2500 developers working on maintenance on a total of 11000 developers.

If many organizations use third party software (e.g. COTS) which they don't have to maintain, this is less true for their core business applications where their market differential must be implemented in the applications that help running day to day activities. As highlighted by the law of Lehman and Belady, these in-house applications must evolve and in this process will extend far beyond their initial structure, in many independent (and sometimes even conflicting) directions. After some time, it becomes mandatory to restructure the application to federate these evolutions inside a more general and renewed architecture to foster possible future evolutions.

Supporting evolution and prepare migration of applications will be *always* mandatory. Different programming paradigms have been invented to cope with changes: late-binding, the cornerstone of object-oriented programming, is a typical illustration. But we think that no paradigm will eradicate the need for evolution and changes and that the only possible approach is to guide evolution or to repair the damages caused by an inevitable erosion.

III. The failure of current solutions

Whereas software (re-)modularization is a relatively old research field in the context of C or Cobol, it is still really important and requires *innovative approaches* to deal with the complexity of modern systems especially those

developed in OOP languages. Our analysis — also confirmed in the recent literature (e.g., [1], [7], [47]) — is that this failure is a direct consequence of: (1) the *complexity of the manipulated concepts and the variety of modular abstractions* (subsystems, packages, classes, class hierarchies, late-binding, aspects, various import relationships....) as well as (2) a monolithic approach: the use of *one* type of algorithms (clustering) and *one* kind of system representation (software components interactions found in the source code).

A. Modular Abstractions

Modular constructs have been the focus of a large body of research. Here we give a non-exhaustive list. A lot of work is currently underway in the context of aspect-oriented programming. Module and package systems have been the focus on a large amount of work. The recent work on Units [22], Jiazzi [42], Mixjuice [28], MJ [14], JAM [52], mixin layers [48] shows that this topic is a crucial research area. Bergel *et al.* conducted a survey on modular language constructs that reflects such a diversity [6].

Modular constructs have also been considered at a lower level than classes, *e.g.,* with mixins [9], traits [17]. These constructs denote, here, sets of properties that somewhat represent the *differentia* in the Aristotelian *genus-differentia* definition. Overall, remodularization must address both modular construct levels, by clustering related properties for defining classes and related classes for defining packages. Recursively nested class models have also been proposed [20], [44]—however they cannot be considered as long as the point with non-recursive models is not fixed.

Component-based software approach proposed to build software systems by assembling prefabricated reusable components [54]. Assembly consists in connecting matching interfaces of the components: in a connection, a required interface (describing the services a component needs) is connected to a provided interface (describing the services another component offers). Extracting a component architecture from an object-oriented software has many in common with remodularization because classes need to be grouped based on their dependencies to form the components [11].

Current Issues – *The class notion is the only universally accepted modular abstraction. Higher and lower level abstractions are often still in flux. Different languages use different concepts such as modules, packages, namespaces. Hence, current issues involve both identifying adequate abstractions and adapting remodularization algorithms to the various alternative abstractions. Moreover, assessing the modularity of software requires specific tools (e.g., metrics, visualization) that must be adapted to each modular abstraction.*

B. Remodularization Approaches

Class hierarchy analysis: Class hierarchy analysis has been largely investigated, for restructuring purposes or finding separate concerns (aspects, traits). As it has been shown in [26], most of the refactorization approaches use explicitly or implicitly substructures of those obtained by Formal Concept Analysis (FCA).

The application of clustering algorithms for software remodularization has been intensively studied [2]. Thousands of experiments were conducted to compare different clustering algorithms, different representation schemes and different coupling metrics between files. Although the experiments used procedural systems, many conclusions may be applied to OO systems as well. The extraction of class views based on Formal Concept Analysis has been proposed in [3]. They evaluated how FCA supports the identification of traits in existing hierarchies [8], [35]. Godin [24] developed FCA algorithms to infer a non-redundant form for implementation and interface hierarchies and carried out experiments on several Smalltalk applications. For dealing with UML models, Relational Concept Analysis, an extension of FCA, takes the relations into account [27]. Other approaches analyze class hierarchies using access or usage information. [49], [51] analyze the *usage* of the hierarchy by a set of client programs. Mining aspects has been considered in the context of FCA [10].

Current Issues – *Factorization is by nature a combinatorial process. Recent studies [21] show that Relational Concept Analysis, applied to rich UML descriptions including references between concepts, produces a huge number of artefacts which is quite impossible to analyze by hand. Execution time can become a problem for large size software, but the actual difficulty is the result size.*

Other modular construct discovery: Clustering approaches are, by far, the preferred algorithmic approach to the problem. They have been proposed to identify modules in applications that are not specifically object-oriented (e.g. [29], [30], [38], [43]).

Finding components in object-oriented software is proposed in [12]. Simulated annealing is used to gather classes into components by optimizing metrics measuring cohesion and coupling. The problem has similarity with package mining.

It is a well-known practice to layer applications with bottom layers being more stable that top layers [41]. Until now few works have been done in practice to identify layers: Mudpie [55] is a first cut at identifying cycles between packages as well as package groups potentially representing layers. DSM (dependency structure matrix) [53], [46] are adapted for such a task but there is a lack of detail information. From the side of remodularization algorithms, a lot of them were defined for procedural languages [31]. However object-oriented programming languages bring some specific problems linked with late-binding and the fact that a package does not have to be systematically cohesive since it can be an extension of another one [18], [57].

Current Issues – *These approaches are often not customized for object-oriented applications. Existing solutions propose modules at a very low level of abstraction that do not reduce enough the size of the system comprehension problem. Solutions that may offer larger (potentially more abstract) modules, result in modules that have no meaning for the software engineers.*

C. Techniques for module assessment

Software Metrics: Re-modularization of software systems is geared toward producing highly cohesive and loosely coupled modules. Many different cohesion/coupling metrics were proposed (including a study by [2]). In the more specific case of object-oriented programming, assessing cohesion and coupling has been the focus of several metrics. However their success is rather mitigated as the number of critics raised. For example, LCOM [13] has been highly criticized [4]. Other approaches have been proposed such as RFC and CBO [13] to assess coupling between classes. However, many other metrics have not been the subject of careful analysis such as Data Abstraction Coupling (DAC) and Message Passing Coupling (MPC) [5], or some metrics are not clearly specified (MCX, CCO, CCP, CRE) [36]. New cohesion measures were proposed [40], [45] taking class usage into account. The Cohesion/Coupling dogma, however, started to receive critics in recent times [1], [47]. People argue that software engineers do not base clustering on this criterion but rather use more semantical approaches.

Software Visualization: There is a significant effort to create efficient software visualizations to support the understanding and analyses of applications [32], [37], [39], [56]. Lanza and Ducasse worked on system level understanding combining metrics and visualization [33] and class understanding support [16].

Current Issues – *Existing cohesion and coupling metric resulting values are difficult to map back to the actual situation, they lead to packages that seem artificial and are not understood by experts of the systems. There is a lack for package cohesion and coupling software metrics in presence of late-binding promoted by object-oriented programming. There is a need for program visualization to support the understanding of packages and procedural code. In addition, there is a need for new metrics that would yield more "natural" packages.*

IV. Scientific challenges and tracks of research

The main scientific challenges are as follows.

Abstraction Diversity: One of the major problems to solve when tackling remodularization of object-oriented systems is the choice of good abstractions and the appropriate relations between them.

Complementary Remodularization Algorithms: There is a need for a global remodularization infrastructure in terms of analyses (algorithms, information presentation, metrics) that can take into account the diversity of the abstractions in presence (different module semantics, different abstractions and relationships including different levels, functions, classes, packages, etc.).

Complexity and approximation: In the point of view of graph theory, the central problem seems to be close to the classic k-cuts problem [23], [25]. Nevertheless, we must add several new criteria, like the *quality* of the proposed solution. The first step of the research will be the characterization of an optimization criterion. We also think useful to study and analyse the sensibility of the problem with respect to the operations: add/delete of vertices/edges. It is a major challenge to be able to propose robust approximations.

Scalability: Computational complexity of algorithms has a limit, already known for Formal Concept Analysis (FCA/RCA) and foreseeable for exact methods. Checking the scalability of these algorithms is thus an additional challenge. Another issue is the combinatorial explosion which may occur in the remodularization results. Because of the size of current applications, presenting these results to the engineers and guiding them to take a decision is an additional challenge.

Reengineer inputs and quality of the solution: Engineers should drive the remodularization. Fully automated approaches are applicable only to a very limited context. In reality, external constraints have to be specified and taken into account by the remodularization algorithms (such as the inclusion of a class in a specific package). Software engineers should guide the process possibly confronted to different solutions and their relative impacts. Often favoring minimum impact on existing code has to be considered. Finally the quality of the resulting modularizations has to be taken into account.

As a conclusion: The problems of software evolution are many and varied, in this proposal we plan to consider one of these problems: software remodularization. We think urgent to drive such a complete study of the problem, both "vertically" by studying all the aspects of the modularization problem (modeling of the software, modularization quality metrics, modularization algorithms, presentation of the results), and, "horizontally", by considering different modularization approaches. The solution will not apply one single method, but a combination of various skills in different research domains. Such a research would also be guided by platforms for testing ideas on real-world applications.

References

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, Mar. 2001.

[2] N. Anquetil and T. Lethbridge. Comparative study of clustering algorithms and abstract representations for software remodularization. *IEE Proceedings - Software*, 150(3):185–201, 2003.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of 18th Conference on Automated Software Engineering (ASE'03)*, pages 267–270. IEEE Computer Society, Oct. 2003. Short paper.

[4] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[5] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, 11(8):47–52, Jan. 1999.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, Nov. 2005.

[7] P. Bhatia and Y. Singh. Quantification criteria for optimization of modules in oo design. In *Software Engineering Research and Practice*, pages 972–979, 2006.

[8] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, Oct. 2003.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[10] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, 2006.

[11] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.

[12] S. Chardigny, A. Seriai, M. C. Oussalah, and D. Tamzalit. Extraction d'Architecture à Base de Composants d'un Système Orienté Objet. In *INFORSID*, pages 487–502, 2007.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[14] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.

[15] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[16] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[18] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[19] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[20] E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

[21] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization (short paper). In *ASE 2008: 23th IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[22] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[23] A. Freize and M. Jerrum. Improved approximation algorithm for $\max -k$-cut and $\max$ bisection. *Algorithmica*, 18:67–81, 1997.

[24] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[25] O. Goldschmidt and D. Hochbaum. Polynomial algorithm for the k-cut problem. In I. C. Society, editor, *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 444–451, 1988.

[26] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify building class hierarchies algorithms. *ITA*, 34(6):521–548, 2000.

[27] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007.

[28] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[29] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, 1988.

[30] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[31] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[32] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[33] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[34] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[35] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.

[36] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[37] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, May 2001.

[38] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[39] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[40] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[41] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[42] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.

[43] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[44] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[45] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[46] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[47] R. Sindhgatta and K. Pooloth. Identifying software decompositions by applying transaction clustering on source code. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth.*, 11(2):215–255, 2002.

[49] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[50] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[51] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[52] R. Strniša, P. Sewell, and M. Parkinson. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM.

[53] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[54] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] D. Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[56] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.

[57] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.