IVAR: A Conceptual Framework for Dependency Management

Gustavo Ansaldi Oliva and Marco Aurélio Gerosa

Instituto de Matemática e Estatística – Universidade de São Paulo (USP)
CEP: 05508-090 – São Paulo – SP – Brasil

{goliva,gerosa}@ime.usp.br

Abstract. *The design of software systems tend to degrade throughout time. Changes in code end up introducing new and unplanned dependencies among modules, causing the elegance of the design to be slowly lost in the form of structural patches and violations of previously established architectural rules. In this study, we propose a conceptual framework for dependency management. Our aim is to conceive a framework that adequately represents and organizes the core concepts of dependency management. The framework revolves around four main concepts, namely: identification, visualization, analysis, and restructuring of dependencies. In a long-term perspective, we envision that a systematization of the field will provide a solid ground for the development of more effective dependency management methods and tools.*

1. Introduction

The volatility of requirements and the wish for new functionalities impose constant pressure for changes in software systems. The need for end-user satisfaction and the great amount of investment leave no room for the software to be largely reworked each time a requirement changes. Indeed, software that is not tolerant to modifications is doomed to abandonment or replacement. In this context, a great research effort has been devoted to the problem of *software design degradation* over time (a.k.a. design erosion) [Jacobson 1992, Lehman *et al.* 1997, van Gurp and Bosch 2002, Martin and Martin 2006]. Degradation occurs as the elegancy of the system modules becomes lost in the form of structural patches and violations of architectural rules [Martin and Martin 2006]. In this scenario, a dense network of interdependencies among modules emerges, resulting in code that is difficult to change, not reusable, and that does not communicate its intention [Foote and Yoder 1999]. As a result, software evolves increasingly slower and its maintenance cost turns to account for a large portion of the total production costs.

One of the reasons why designs degrade is because requirements change in ways not anticipated by the initial design [Martin and Martin 2006]. Consequently, design principles are violated and changes to code end up introducing new and unplanned dependencies among the system's modules. In this context, the notion of *dependency management* emerged as a promising approach [Arnold 1996, Lakos 1996, Martin and Martin 2006, Binkley 2007] to deal with this issue. Throughout the years, a series of advances have been achieved in the field, especially when it comes to the management of structural dependencies. Furthermore, given the advent of mining software repositories, a great progress has also been made in detecting and analyzing logical

dependencies. Nevertheless, as a consequence, dependency management has given rise to an ever growing body of knowledge that is scattered throughout the literature and existing tools. This effective leads to a series of issues that hamper further advances in research and practice: it becomes difficult to know which dependency management techniques exist and which one is most suitable for each kind of problem, how do techniques related to each other, how different researches can work together to accomplish greater progress, which theories need further validation in practice and which practices would benefit from having a deeper theoretical basis, etc. In a long term perspective, a greater understanding of the dependency management concept should provide a more solid basis for the development of more effective tools and methods.

In this paper, we propose an initial version of a conceptual framework to represent and organize the main concepts of dependency management. Our framework is intended for object-oriented systems and revolves around four main core concepts, namely: identification, visualization, analysis, and restructuring of dependencies. The name IVAR is an acronym for such concepts. We believe our framework to be a first step towards the proper systematization of the dependency management field.

This paper is structured as follows. In Section II we present the IVAR conceptual framework and its main elements. In Section III we discuss related work and in Section IV we state our conclusions and plans for future work.

2. The Dependency Management Conceptual Framework

Dependency management encompasses a large body of knowledge that is currently scattered throughout the literature and existing tools. Therefore, prior to digging into the specific aspects and details of dependency management, we believe that the boundaries and scaffolding of the field should be first appropriately set. In particular, it is important to define which techniques are the most important, which relationships are likely to be most meaningful, and, as a consequence, what information should be collected and analyzed. In this sense, we propose a dependency management conceptual framework. A conceptual framework explains, either graphically or in narrative form, the main things to be studied (the key factors, constructs, or variables) and the presumed relationship among them [Miles and Huberman 1994]. The IVAR framework represents and organizes *dependency management* as a set of four main interconnected activities: identification, visualization, analysis, and restructuring of dependencies. Figure 1 shows an overview of the framework (BPMN2 process notation).

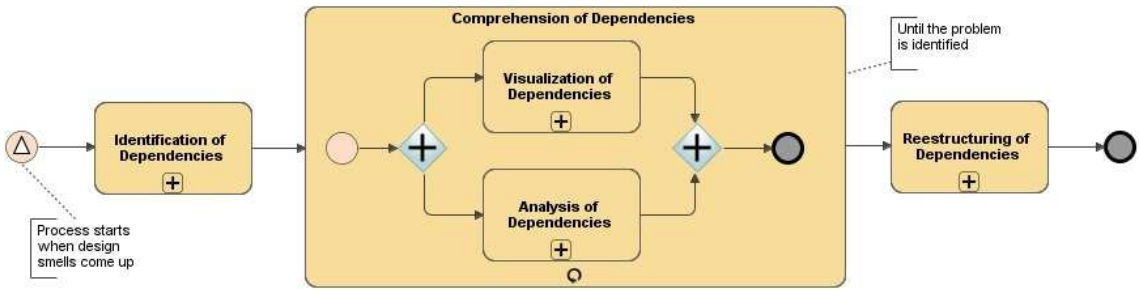

Figure 1. Dependency management framework overview

An important aspect concerns understanding when dependency management should be actually performed. According to Martin [Martin and Martin 2006],

dependency management should be conducted only when design smells come up and are noticed by developers, as too much engineering may lead to needless complexity. Traditional design smells include *rigidity* (the design is difficult to change), *fragility* (the design breaks easily), *immobility* (the design is difficult to reuse), and *viscosity* (it is difficult to "do the right thing").

As dependencies are the subject of management, then identifying dependencies should be the first primary activity. The identification of dependencies involves capturing structural and logical dependencies from the code and the associated version control system respectively. The next step involves investigating dependencies and discovering the cause behind the perceived smell. This is achieved by means of techniques for dependency visualization and analysis. Finally, once the architecture of dependencies is known and problems are identified, the restructuring activity starts. In this step, software restructuring and reengineering techniques are applied.

As our goal is to define the boundaries and scaffolding of dependency management, we focus on presenting existing tools and significant research efforts. In the following sections, we describe each of the main activities of our proposed framework.

2.1 Dependency Identification

Strategies and techniques for identifying dependencies play a crucial role in dependency management, since they concretely define the set of dependencies that will be available for the visualization and analysis tools. In our proposed framework, we focus on two dependency types: structural and logical dependencies.

Structural dependencies comprise the set of all detectable relationships among compilation units (concrete classes, abstract classes, and interfaces) through static analysis. Structural dependencies occur whenever a compilation unit depends on another during compilation time or linkage time [Lakos 1996]. The identification of structural dependencies involves primarily recognizing client compilation units and their respective suppliers through static analysis. Capturing other metadata such as relationship type and dependency stereotype is usually useful, as it provides information that may assist in the understanding of the relationship nature and coupling degree between the involved elements. Static analysis may be performed on both source code and compiled code, being the latter the most common approach. A discussion about the advantages and disadvantages of scanning each form of code can be found on [Barowski and Cross 2002]. Although the identification step is usually embedded in dependency visualization and analysis tools, there are specific ones solely dedicated to this task. Examples of tools that detect structural dependencies in Java code include Dependency Finder, Java Dependency eXtractor (JDX), and the more general-purpose bytecode manipulation tool ASM.

Logical dependencies (a.k.a. change dependencies, evolutionary dependencies, and co-changes) are implicit dependencies between software artifacts that have evolved together [Zimmermann *et al.* 2005]. These logically linked artifacts are not necessarily structurally related, since they are connected from an evolutionary point of view, i.e. they have often changed together in the past and thus they are likely to change together in the future. While structural dependencies are defined for a specific time instant,

logical dependencies are defined based on a time interval. Furthermore, unlike structural dependencies, logical dependencies may occur between any kind of artifact that composes the system, including configuration files and documentation. Cataldo and Nambiar [Cataldo and Nambiar 2010] investigated 189 projects and showed that logical coupling was the most significant factor impacting software quality among a series of factors, including structural coupling, process maturity, developers' experience, and number of sites. The identification of logical dependencies is usually performed by parsing and analyzing the logs of a version control system [Zimmermann *et al.* 2005, Oliva *et al.* 2012]. Examples of tools that detect logical dependencies include XFlow [Santana *et al.* 2011] and Churrasco [DAmbros and Lanza 2010].

2.2. Dependencies Visualization

Throughout the years, many different dependency visualization techniques were conceived, developed, and implemented. This plurality of options has enabled the visualization of dependencies according to different perspectives and levels of abstraction, thus aiding in software comprehension and management of complexity. We now briefly present some valuable dependency visualization techniques.

Fine-grained analysis. The UML is the standard visual language for describing object-oriented software and several dependency visualization tools employ it. The radial tree visualization is available in IBM Structural Analysis for Java (IBM SA4J) tool and it displays packages, classes and interfaces in a radial layout with notation inspired on the UML. This interactive visualization provides detailed information about the clients and suppliers of a particular compilation unit. In a case study [Padda *et al.* 2009], such technique achieved the highest average comprehension score when compared to the Skeleton/Pyramid (SA4J), Tree (Creole), and NestedView (Creole) visualization techniques. The use of radial tree is suitable when detailed information about a compilation unit is required. In some scenarios however, the dependency graphs can be too large, complex, and difficult to understand. In these cases, graphs with auto-partition feature may replace the radial tree technique. This feature analyzes the dependency graph and partitions it according to the dependency structure. The resulting partitions are indicated as groupings on the dependency diagram, which can be of two kinds: tangles and clusters. Structure 101 (S101) and Stan4J commercial tools implement this visualization technique. An additional strategy and tool for visualizing detailed information is called Evolution Radar [D'Ambros *et al.* 2009]. Although originally proposed for visualizing logical coupling, it can be adapted to show either logical or structural coupling. Evolution Radar makes it possible to investigate the degree to which a particular chosen module (or compilation unit) is coupled to the rest of the system. Finally, we highlight a change impact analysis visualization technique called *what if*, which is implemented on the SA4J tool. This feature assists in rapidly verifying the possible effects that stem from a change made into a single compilation unit. When a specific element is selected, the tool transitively paints and connects all the dependent elements, denoting the transitive closure of relationships

Coarse-grained analysis. There are times when it is more suitable to visualize software structure at a higher level of abstraction, especially when the intent is to analyze architectural properties. In this case, the dependency structure matrix (DSM) is a traditional suitable option as it provides valuable information at a coarse-grained level

[Sangal *et al.* 2005]. In this visualization scheme, lines and columns correspond to modules, and each cell number corresponds to the coupling degree between the involved modules. Lattix LDM tool implements this visualization technique and additionally provides partitioning algorithms (to find layers and highlight cycles).

2.3. Dependencies Analysis

Besides software visualization, there is a series of techniques devoted to the analysis of software design and architecture. In the following, we present important techniques. For a more general and extended review on dependency analysis, we refer the reader to work of Arias *et al.* [Callo Arias *et al.* 2011].

Architectural Conformance Checking. The role played by software architecture in the development process is widely known as crucial. Architectural conformance consists in verifying whether a low-level representation of a software system (e.g. the code) complies with the planned architecture (also known as documented architecture) [Sangal *et al.* 2005, Passos *et al.* 2010]. No conformation means that decisions implemented in code somehow violate the planned architecture, leading to unexpected collateral effects and design degradation. Furthermore, it can be than no conformation means that the planned architecture should be reviewed. Several known techniques and tools aid in architecture conformance. Lattix LDM tool enforces architectural conformance through the specification of architectural rules (in *x can-use y* and *x cannot-use y* forms) and partitioning algorithms. Structure 101 tool aids on this task by comparing a user defined architectural diagram to the actual code, resembling the reflexion models technique [Murphy *et al.* 2001]. JDepend tool accomplishes this task by providing an appropriate Java API that can be used in unit tests.

Structural Anti-Patterns Identification. The anti-pattern term was coined in 1995 by Andrew Koenig to describe ineffective or counter-productive patterns [Koenig 1995]. Structural anti-patterns frequently point to design portions that should be refactored or treated with special caution. We borrow the terminology employed in IBM SA4J to present a few structural anti-patterns. A *local butterfly* is a compilation unit that has many immediate incoming dependencies. A *global butterfly* is a compilation unit that has many indirect incoming dependencies and can thus affect many components of the system due to a change. Butterflies are not harmful as long as they comprise stable compilation units (basic interfaces, abstract base classes, etc.). In turn, a *local breakable* is a compilation unit that has many immediate outgoing dependencies (high coupling). A *global breakable* is a compilation unit that has many outgoing dependencies and is thus often affected when any other compilation unit in the system is changed. A *local hub* (a.k.a *change propagator*) is a compilation unit that is both a butterfly and a breakable at the same time. A *global hub* is a compilation unit that is often affected when any other compilation unit is changed and also frequently affects a lot of other compilation units when it is changed. Finally, a *tangle* is a strongly connected component of a dependency graph (i.e. every compilation unit is involved in a cycle). Cyclic dependencies involving modules are especially harmful and thwart code reuse. Structure 101 and Stan4J provide some support for breaking large tangles.

Dependency-Centric Metrics. A variety of object-oriented metrics have long been used to support software maintenance activities [Lanza and Marinescu 2006]. In this version

of the IVAR framework, we pick a specific set of dependency-centric metrics that we believe can aid in the identification of design problems. Structural coupling can be measured in terms of the traditional Coupling Between Objects (CBO) and Fan-Out metrics. A more coarse-grained notion of coupling can be achieved using the *system overall stability* metric, which was proposed in the IBM SA4J tool. This metric relies on the transitive closure of the system dependency graph and reveals whether the coupling between elements is being controlled (in average). We also highlight the *distance from the main sequence* metric [Martin and Martin 2006], which evaluates the balance between the stability and the abstraction of a given package. As for logical dependencies, their coupling degree is often given in terms of support and confidence measures (which originate from the data-mining field) [Zimmermann *et al.* 2005, Oliva and Gerosa 2011, Oliva *et al.* 2012].

2.4. Dependencies Restructuring

While dependency identification, visualization, and analysis are included in the reverse engineering domain, the dependencies restructuring activity turns the focus to the software reengineering domain. The problematic dependencies spotted during visualization or analysis activities are restructured through the joint application of fundamental object-oriented design techniques.

In this version of the conceptual framework, we highlight two sets of design principles, namely GRASP [Larman 2004] and SOLID [Martin and Martin 2006]. Furthermore, we also consider design patterns [Gamma *et al.* 1994] and refactoring techniques [Fowler 1999, Lanza and Marinescu 2006, Demeyer *et al.* 2009]. In particular, we highlight the didactic survey on software refactoring by Mens and Tourwé [Mens and Tourwé 2004]. We believe this set of conceptual tools establishes a solid proven theoretical basis for the restructuring and evolution of software design.

3. Related Work

To the best of our knowledge, this is the first research attempt that aimed to put dependencies identification, visualization, analysis, and restructuring together in a conceptual framework. We highlight the work of Pirklbauer *et al.* [Pirklbauer *et al.* 2010], who propose a method and tool to manage dependencies, giving focus to the change impact analysis aspect. An interesting point is that the tool they propose tries to group structural and logical dependencies into a single database, which is later used by the visualization component they developed. In fact, a deeper understanding of the interplay between structural and logical dependencies remains as a research challenge [Oliva and Gerosa 2011]. We also highlight the work of Sangal *et al.* [Sangal *et al.* 2005], in which they present an approach for managing software architecture using DSMs. A case study is shown and the authors discuss that their approach helps in visualizing the architecture, identifying structural problems, and checking the results of refactoring procedures.

4. Conclusion

In this paper, we discussed the design degradation problem and proposed an initial version of a conceptual framework named IVAR, which comprises four main interconnected activities: identification, visualization, analysis, and restructuring of

dependencies. The goal was to represent and organize the main concepts of dependency management, so that it becomes clearer what the main techniques are, what they can be used for, how do different techniques related to each other, and how different researches can work together to accomplish greater progress.

As future work, we plan to extend the framework by applying a more systematic and unbiased research method. In this current version of the framework, we focused on existing tools and significant research results. Also, the selection of framework elements was done mainly based on the researchers experience in the topic. We believe that a systematic review on each one of the IVAR main activities would be an appropriate research method to further extend the current version of the framework. In a long term perspective, we think that this will provide a solid basis for the elaboration of more effective dependency management methods (to be embedded into development processes) and tools. We also think it would be particularly helpful for the restructuring activity, in which little computational support exists to help enforce design principles and the appropriate use of design patterns.

Acknowledgements

This work is also partially supported by HP (Baile project) and CHOReOS EC FP7 project. Marco Gerosa receives individual grant from CNPq.

References

Arnold, R. S. (1996). Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA.

Barowski, L. A. and Cross, J. H. (2002). Extraction and use of class dependency information for java. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), WCRE '02, pages 309–, Washington, DC, USA. IEEE Computer Society.

Binkley, D. (2007). Source code analysis: A road map. In 2007 Future of Software Engineering, FOSE '07, pages 104–119, Washington, DC, USA. IEEE Computer Society.

Callo Arias, T. B., Spek, P., and Avgeriou, P. (2011). A practice-driven systematic review of dependency analysis solutions. Empirical Softw. Engg., 16:544–586.

Cataldo, M. and Nambiar, S. (2010). The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects. Journal of Software Maintenance and Evolution: Research and Practice.

DAmbros, M. and Lanza, M. (2010). Distributed and collaborative software evolution analysis with churrasco. Sci. Comput. Program., 75:276–287.

D'Ambros, M., Lanza, M., and Lungu, M. (2009). Visualizing co-change information with the evolution radar. IEEE Trans. Software Eng, 35(5):720–735.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2009). Object-Oriented Reenginering Patterns. Square Bracket Associates, first edition.

Foote, B. and Yoder, J. W. (1999). Pattern Languages of Program Design, volume 4. Addison-Wesley Professional.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, first edition.

Jacobson, I. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Professional, first edition.

Koenig, A. (1995). Patterns and antipatterns. Journal of Object-Oriented Programming (JOOP), 8(1):46–48.

Lakos, J. (1996). Large-Scale C++ Software Design. Addison-Wesley Professional, first edition.

Lanza, M. and Marinescu, R. (2006). Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, first edition.

Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, third edition.

Lehman, M., Perry, D., Ramil, J., Turski, W., and Wernick, P. (1997). Metrics and laws of software evolution–the nineties view. In Proceedings IEEE International Software Metrics Symposium (METRICS'97), pages 20–32, Los Alamitos CA. IEEE Computer Society Press.

Martin, R. C. and Martin, M. (2006). Agile Principles, Patterns, and Practices in C#. Prentice Hall, first edition.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. IEEE Trans. Softw. Eng., 30(2):126–139.

Miles, M. B. and Huberman, A. M. (1994). Qualitative Data Analysis: An Expanded Sourcebook. Sage Publications, second edition.

Murphy, G. C., Notkin, D., and Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. IEEE Trans. Softw. Eng., 27(4):364–380.

Oliva, G. A. and Gerosa, M. A. (2011). On the interplay between structural and logical dependencies in open-source software. In Proceedings of the 2011 25th Brazilian Symposium on Software Engineering, SBES '11, pages 144–153, Washington, DC, USA. IEEE Computer Society.

Oliva, G. A., Santana, F. W., Gerosa, M. A., and de Souza, C. R. (2012). Preprocessing change-sets to improve logical dependencies identification. In Proceedings of the 6th International Workshop on Software Quality and Maintainability, SQM '12. Proceedings available on http://sqm2012.sig.eu/2012_proceedings.pdf.

Padda, H., Seffah, A., and Mudur, S. (2009). Investigating the comprehension support for effective visualization tools - a case study. In Proceedings of the 2009 Second International Conferences on Advances in Computer-Human Interactions, ACHI '09, pages 283–288, Washington, DC, USA. IEEE Computer Society.

Passos, L., Terra, R., Valente, M. T., Diniz, R., and das Chagas Mendonca, N. (2010). Static architecture-conformance checking: An illustrative overview. IEEE Softw., 27(5):82–89.

Pirklbauer, G., Fasching, C., and Kurschl, W. (2010). Improving change impact analysis with a tight integrated process and tool. In Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations, ITNG '10, pages 956–961, Washington, DC, USA. IEEE Computer Society.

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

Santana, F., Oliva, G., de Souza, C. R. B., and Gerosa, M. A. (2011). Xflow: An extensible tool for empirical analysis of software systems evolution. In Proceedings of the VIII Experimental Software Engineering Latin American Workshop, ESELAW '11.

van Gurp, J. and Bosch, J. (2002). Design erosion: problems and causes. J. Syst. Softw., 61(2):105–119.

Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. IEEE Trans. Softw. Eng., 31:429–445.