# Recovering Lost Software Design with the Help of Aspect-based Abstractions

Kiev Gama
Centro de Informática
Universidade Federal de Pernambuco (UFPE)
Recife, PE
email: kiev@cin.ufpe.br

Didier Donsez
Laboratoire d'Informatique de Grenoble
University of Grenoble
Grenoble, France
email: didier.donsez@imag.fr

*Abstract*— **In this paper, we propose an unconventional usage of aspect-oriented programming, presenting and discussing a novel approach for recovering layered software design. It consists of a reengineering pattern based on aspect abstractions that work as a strategy for recovering software design. By using our approach it is possible to employ general purpose aspects that represent software layers. This is useful for capturing such design in systems where a layered architecture exists but was not documented or where it has been inconsistently translated from design to code. The pattern is a generalization of our initial validation performed in a case study on the Open Service Gateway Initiative (OSGi) service platform. We could verify that its software layers are well defined in the specification and design, however when analyzing the actual Application Programming Interface (API), such layers are completely scattered over interfaces that inconsistently accumulate roles from different layers. By extracting the layered design into separate aspects, we were able to better understand the code, as well as explicitly identifying the affected layers when applying dependability crosscutting concerns to a concrete aspect solution on top of three different implementations of the OSGi platform.**

***Keywords-Software architecture; Software layers; Software reengineering; Aspect-oriented programming.***

## I. INTRODUCTION

*Reverse engineering*, *Reengineering* and *Restructuring* are close terms, with subtle differences. Definitions from [3] indicate reengineering as the examination and alteration of a system to reconstitute it to a new form, while restructuring consists of transforming the system code keeping it at the same relative abstraction level, and preserving its functionality. Reverse engineering would consist of analyzing a system in order to identify its components and to create abstract representations of it.

Recovering lost information (e.g., design) and facilitating reuse are important reasons for reengineering [3]. Other reasons [4] leading to reengineering a software system are: insufficient documentation, improper layering, lack of modularity, duplicated code or functionality are among the coarse-grained problems. As a part of the reengineering process, one may employ techniques, such as refactoring [7] as a form of code restructuring. Refactoring consists of "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure". Aspect-oriented programming (AOP) [15] is a paradigm that is very useful when restructuring and reengineering systems. It allows changing the system without actually changing the system's source code. It is possible to keep cross-cutting concerns separate from the target system code at development time. Such concerns can be later integrated by "weaving" them to the target application either at compile time or during runtime.

Typical usages of AOP are straightforward solutions that either refactor crosscutting concerns out of the system code or introduce crosscutting concerns in the form of aspects woven in the system. Sometimes it may not be clear which system layers are being crosscut by which aspects, especially in systems with weak design or where the implemented design differs from documentation. In this paper, we propose the usage of aspects for providing another level of indirection that helps understanding systems that are reengineered with AOP. We provide an AOP refactoring pattern that helps capturing system design by aspectizing software layers, which can be reused by aspects that are applying concrete aspects as concerns that crosscut such layers (and consequently the system). Such abstractions we propose are useful for better understanding software architectures in systems with weak design (e.g., monolithic systems) or where design has been badly translated from the specification during its implementation. Therefore, the contributions of this paper are: 1) an approach for using aspects as an abstraction for capturing lost architectural design; 2) the refactoring of specific aspects that will target such abstractions instead of coding the aspects directly against the target system code; and 3) a reengineering pattern that guides through the process of extracting such design.

The pattern described here was validated in a case study of the OSGi [17] service platform. We present an architectural perspective that is useful in the context of reverse engineering for recovering lost design information, as well as in the context of reengineering when applying changes to the system and reusing the definitions of such abstractions that recover lost design. The remainder of this paper is organized as follows: Section 2 provides an overview of the problem, Section 3 describes the reengineering pattern, Section 4 details the case study in the OSGi platform, Section 5 discusses related work, followed by Section 6 that concludes this article.

## II. OVERVIEW

Software layers [1] are an architectural pattern extensively used for grouping different levels of abstraction in a system. By employing such pattern for layered architectures, it is a good practice to design a flat interface that offers all services from a given layer. In a purist layer design, a layer of a system should only communicate with its adjacent layers, via such flat

 ISBN: 978-1-61208-438-1

interfaces. Such type of design gives a commonly used architectural view of systems. We find cases where the system is well designed in terms of layering, but the corresponding implemented code does not represent explicitly such architecture. In other (worst) cases, the system lacks good abstraction during design, resulting in a monolithic architecture which is hard to understand.

From now on in this article, the term reengineering will be employed as a general task – which may involve reverse engineering and restructuring – for improving system code and design. By reengineering the code, it is possible to arrive at a system whose architecture is more transparent, and easier to understand. In [4], extracting the design is considered as a first step for performing new implementations. Either if reimplementing the system or just applying the required changes, this step is very important.

### A. Aspect-oriented Programming

The principle of Aspect-oriented Programming [15] is a paradigm that improves the modularity of applications by employing the principle of Separation of Concerns (SoC) advocated by Dijkstra [6]. In SoC, one should focus on one aspect of a problem at a time, as a way to have a better reasoning on a specific aspect of a system. An aspect should be studied in isolation from the other aspects but without ignoring them.

Putting these concepts into practice, AOP allows the separation of concerns (e.g., logging, transactions, distribution) that crosscut different parts of an application. These crosscutting concerns are kept separate from the main application code instead of being scattered over different parts of the system, as illustrated in Figure 1. A source file (e.g., module, class) may also have code that accumulates different responsibilities not necessarily related, giving an impression of tangled code.

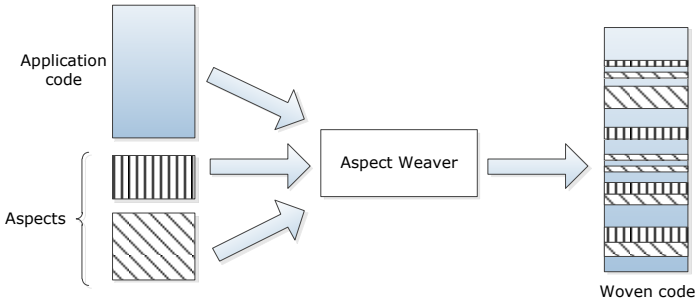


Figure 1. Illustration of how aspects are maintained outside the target application code, and then are intermixed with it.

AOP employs its own terminology, from which we briefly clarify some of the commonly used expressions that are going to be frequently cited throughout this article. Jo*in points* are constructs that capture specific parts of program flow (e.g., method call, constructor call). *Pointcuts* are elements that pick one or more specific join points in the program flow. The code that is injected into pointcuts during the weaving process is called advice in AOP terminology.

### B. Lost Design

AOP is useful in the context of reengineering either to apply changes to code by introducing new crosscutting concerns, or by refactoring out from code existing crosscutting concerns into aspects. When in such AOP usage, we propose to give more semantics to pointcuts in a way that it is possible to represent part of the system design, by grouping the pointcuts in meaningful abstractions (e.g., layers) that could be reused. Our proposition does not involve changes in the aspect language level, but rather relies on existing constructs for building such abstract representations. Figure 2 illustrates an example where aspects are applied directly to the system code, and later layers are introduced as reusable aspects that contain more semantics.

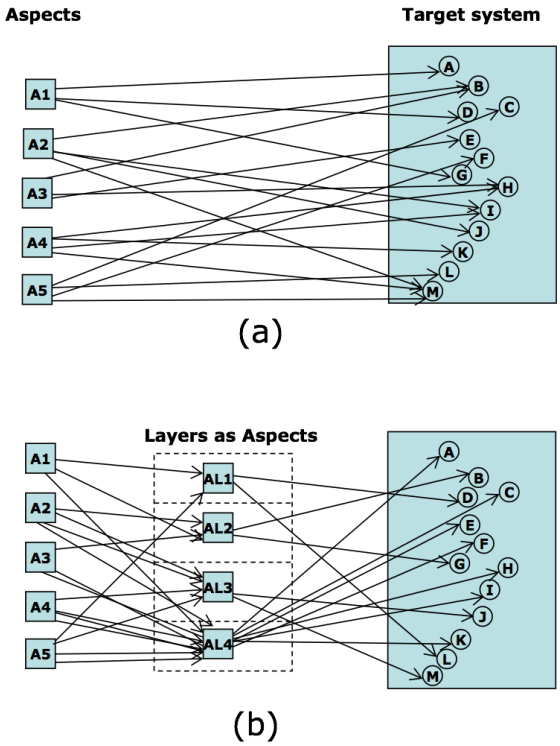


Figure 2. Aspects defining pointcuts (circles) on the reengineered system that are logically grouped in intermediary abstractions (layers as apects) that can be used to "visualize" the system's layers.

In a typical utilization of aspects, we define pointcuts using join points that directly reference the code of the target system, without any intermediary abstractions. This may end up with redundant pointcut definitions, especially in larger systems or in systems where aspects represent a significant part of the code. This redundancy is illustrated in Figure 2 by the pointcuts B, H, I and M, which are used by more than one aspect. If each definition involves several join points (e.g., method calls, method executions, instantiations), it may be difficult to give some reusable semantics to it. In addition, if the same set of join points is to be used in another aspect, we end up with redundant code. Indeed, we can give aliases to pointcuts for better expressiveness and reuse within the same aspect as we illustrate further.

### C. Approach

At large, what we propose is to logically group pointcuts in general purpose aspect definitions that do not provide advices

```
public aspect A2 {
 void   around():   execution(void   Foo+.set*(..))
        || execution(void Bar.setFoo(Foo)){
    //advice code
 }}
public aspect A4 {
 pointcut   X():   execution(void   Foo+.set*(..))
          || execution(void Bar.setFoo(Foo));
 void around(): X() {
    //advice code
  }}
```

Figure 3. The same pointcut definition in the form of an anonymous pointcut in aspect A2 and as a named pointcut in aspect A4.

but only pointcut definitions. That gives more semantics to the aspects, allowing us to logically represent software layers that were not correctly (or not at all) represented in the original system. In the case from our example, the monolithic design of the target system is now represented with aspects that mimic layers (e.g., data access layer, GUI layer). They provide a new abstraction that captures such design concept. We also avoid redundant definitions of pointcuts. For instance, instead of aspects A2 and A3 having to write pointcut B twice, such pointcut is going to be logically grouped together with B in an aspect layer (AL2). The code from A2 and A3 can then reuse the pointcuts from AL2. After this change we now know explicitly that aspects A2 and A3 crosscut the layer represented by AL2. Another conclusion that can be drawn is that layer AL4 is crosscut by all aspects.

To clarify this proposition, we provide some code illustrating our approach. By taking the example of Figure 2 (a), the origin of the links toward the pointcuts (A through M, in the figure) denotes where the corresponding pointcut definitions are located. In such approach, it is normal to have the same pointcut definitions that may be present in different aspects, which represents redundant code as exemplified in Figure 3. The anonymous pointcut definition in A2 is the same used in A4 but cannot be reused, working as a sort of ad hoc pointcut. In contrast, the pointcut X of aspect A4 can be used by different advices just by referring to its name. Based on that reuse possibility, we suggest reusable pointcut definitions logically grouped, providing the semantics of a software layer.

In Figure 2 (b), our approach proposes the introduction of an intermediary abstraction that uses aspects for gathering cohesive pointcuts that would refer to join point in the same software layer. We can use these groupings to represent software layers and also to reuse the pointcut definitions with more semantics. Whenever reusing a pointcut, one would know which layer it refers to. In the example, each aspect layer (AL) illustrated will just group pointcut definitions (A to M) that belong to the same software layer, thus providing a representation of that layer as an aspect. The actual crosscutting concerns should be coded in aspects that refer to the pointcut definitions of these layer aspects, instead of repeating them in their code.

```
public aspect AL3 {
  pointcut J(): /* ... ... */
 pointcut    M():    execution(void    Foo+.set*(..))
        || execution(void Bar.setFoo(Foo));
}
public aspect A2 {
 void around():  AL3.M() {
        //code
  }
}
public aspect A4 {
 void around():  AL3.M() || AL4.K(){
        //code
  }
}
```

Figure 4. Layer aspect AL3 defines the redundant pointcut of previous example.

The code in Figure 4 that illustrates the layers is presented in Figure 2 (b) where we provide the example of the aspect layer AL3, which represents an architectural layer (e.g., data access layer) that was "captured" using two pointcuts. The other two aspects of the example, A2 and A4, reuse the definition of the pointcut M. It is clear that both aspects A2 and A4 crosscut the layer represented by AL3. In the case of aspect A4, one can easily identify just by reading the code that it also crosscuts the layer represented by AL4. The illustrated advice of AL4 will be used whenever the program flow reaches the join points defined by pointcuts AL3.M or AL4.K.

## III. PROPOSED PATTERN

A reengineering pattern is more related with the discovery and transformation of a system, than with the design structure [4]. It is important to note, however, that our proposed reengineering pattern describes a discovery process that involves the identification of a design element (an architectural pattern).

In the next subsections, we employ a similar organization (intent, problem, solution, tradeoffs) to the patterns defined in the Object Oriented reengineering patterns book [4] for describing our pattern named as *"Aspectize the Software Layers"*.

### A. Intent

Utilizing reusable aspects for extracting the layered design of the system and clarifying where (and which) are such software layers.

### B. Problem

Common usages of AOP are basically employed in two ways. The first one consists of refactoring crosscutting concerns out of the system code. The second case consists of introducing previously non-existent crosscutting concerns into the system, in the form of aspects. Both cases typically employ straightforward

 ISBN: 978-1-61208-438-1 

solutions that do not use intermediary abstractions. It is not clear which system layers are being affected (i.e., crosscut), especially in systems with weak design (e.g., monolithic systems) or where design has been badly translated from the specification during its implementation. In larger solutions, pointcuts tend to be repeated where reuse could be possible. An extra level of indirection could introduce more semantics and pointcut reuse.

### C. Solution

Introduce general purpose aspects (i.e., without advices) logically grouping correlated pointcuts, allowing to provide representations of the software layers used in the systems. The pointcut can be reused with better semantics than previously.

Before actually executing the necessary steps, it is important to understand the system being refactored. Applying some of the reverse engineering patterns defined in [4] can help:

- *Speculate about design*. It will allow making hypotheses about existing design so we are able to understand which ones are the existing layers.
- *Refactor to understand.* This is important to understand the code; even if these performed refactorings are not taken into account later (it might be the case when changing existing code is not desired).
- *Look for the contracts*. The proposed intent of this pattern is to infer the usage of class interfaces by observing how client code uses it. In the context of our pattern, this may be the case when contracts are not specific.

After identifying which are the layers and which to be abstracted, it is necessary to create their corresponding aspects. Each aspect will define the pointcuts that represent the services provided by a layer. The granularity level depends on the usage or what is necessary to be represented. For example, a data access layer abstraction could include pointcuts defining the general CRUD (create, read, update, delete) operations as the layer's services.

The layer aspects themselves do not provide any code for advices; therefore alone they are useless. The layer aspects should be reused by advices from other aspects that apply crosscutting concerns (e.g., logging, transactions, distribution) to the target system. In the case where such crosscutting concerns already exist in the form of aspects, it is necessary to apply the look for the contracts pattern in order to understand how these aspects use the target system. Wrapping the aspectized layers as a library that can be imported by the concrete aspects consists of a good reuse practice that should be employed whenever possible.

### D. Trade-Offs

Following the format proposed in [4], the following trade-offs can be considered.

Pros:

- Higher level abstractions
- Clarification of the existing architecture through the extracted design
- Reusable pointcut definitions

Cons:

- Depending on the coverage of the aspects (e.g., crosscuts only parts of the system) the resultant design that was extracted may not completely describe the system architecture
- Poor knowledge of the system may also result in an incomplete representation

## IV. Case Study

Our initial validation of the proposed pattern was performed on the OSGi Service Platform [17], a dynamic environment where components may be installed, started, stopped, updated or unloaded at runtime. The API is standardized and the common point for different implementations. When aspects target the API they become applicable to any of the implementations. In the case of OSGi, we could verify this in our experiment involving the open source implementations of that API: Apache Felix, Equinox and Knopflerfish. We initially applied dependability aspects that were scattered over layers. Our approach used Aspect-J and the Eclipse IDE for defining and weaving the aspects in those implementations. An important fact to be pointed out is that the OSGi implementations in question did not use any aspect-oriented language prior to our intervention.

When we needed to identify which layers were being affected by which aspects, we could not easily tell because the way the specification presents the layers is much cleaner and less entangled that the reality in the API. The next subsections show the steps taken for applying our reverse engineering pattern.

### A. Disentangling OSGi layers

As part of our analysis (speculate about design and look for the contracts) we have noted that useful concepts described in the OSGi specification are not well represented in its API, making it difficult to distinguish the layers in the specification from their counterparts in the API. The OSGi specification proposes a layered architecture, as depicted in the gradient boxes in Figure 5. The service, lifecycle, module and security layers are provided by the OSGi implementations, while the bundles layer represents the third party components that are deployed and executed on the OSGi platform. However, the software layers specified by OSGi are scattered over different interfaces, which accumulate roles from different layers.

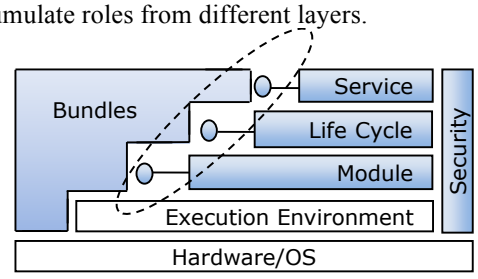


Figure 5. The proposed aspects simulate a single point of access (dashed ellipse) for each layer in OSGi's pseudo-layered architecture.

We found no single entity to describe individual layers in OSGi neither single access points for accessing the services provided by each layer. However, with such layer concept lost when a specification is translated into an API, we lose modularity as well. In the OSGi platform, the bundles layer

 ISBN: 978-1-61208-438-1

freely accesses the other three layers (Figure 5). But, in practice, such access in OSGi is not done through a single interface per layer. Actually, there is no such flat interface for explicitly representing layers in OSGi's API. The functionality of each layer is scattered over different interfaces which may accumulate roles from different layers. In our case study targeting OSGi we have employed our pattern for abstracting the Service, Lifecycle and Module layers, and then refactoring the dependability patterns to use it. We have not handled the security layer because it is an optional layer in OSGi implementations. The "aspectization" of the lifecycle layer (service and module layers were left out due to space limitations) is illustrated in Figure 6 and is a result of the next step when applying our pattern. The refactor to understand pattern was also helpful, and in our case it happened previously, at the time we applied the dependability aspects.

```
public aspect LifeCycle {

 pointcut install():
     execution(Bundle
BundleContext+.installBundle(String,..));

  pointcut stop():
     execution(void Bundle+.stop(..));

  pointcut start():
     execution(void Bundle+.start(..));

  pointcut uninstall():
     execution(void Bundle+.uninstall());

  pointcut update():
     execution(void Bundle+.update(..));

  pointcut resolve():
     execution(boolean
     PackageAdmin+.resolveBundles(Bundle[]));

  pointcut refresh():
     execution(void
    PackageAdmin+.refreshPackages(Bundle[]));

  pointcut activate():
     call(void
      BundleActivator+.start(BundleContext));

  pointcut deactivate():
     call(void
       BundleActivator+.stop(BundleContext));
}
```

Figure 6. Aspect representing OSGi's lifecycle layer

In Figure 6, the methods and transitions that concern bundle lifecycle are scattered across four interfaces (Bundle, BundleContext, BundleActivator, PackageAdmin) that already have roles other than lifecycle management. The different state transitions of a bundle's lifecycle are scattered over different interfaces. The install state transition is actually fired in the BundleContext interface. The resolve transition is defined in the PackageAdmin service interface, while the update and uninstall can be found in the Bundle interface. The refresh transition is part of the package admin, which is not part of the core API but rather declared in the PackageAdmin. The start and stop are both located in the Bundle and BundleActivator interfaces. In case of a Bundle having a BundleActivator, those calls are delegated to it. In the LifeCycle aspect we have rather called it as activation and deactivation, respectively.

A simple illustration of the lifecycle layer aspect being reused is shown in the advice from Figure 7, which provides a practical usage of an aspect targeting that layer by reusing the Lifecycle aspect (i.e., an aspect that abstract a software layer). The semantics of the code becomes clearer with a higher level concept. Although the original definition of the start pointcut involves only one join point in the Bundle interface, other cases that involve long pointcut definitions would gain more in terms of reuse and semantics gain.

```
public aspect ComponentIsolation {
void around(Bundle b): LifeCycle.start()
     && !cflowbelow(LifeCycle.start())&&
         this(b){
 if (!PlatformProxy.isSandbox() &&
     PolicyChecker.checkIsolation(b)){
     PlatformProxy.start(b.getBundleId());
  } else {
    proceed();
  }
 }
}
```

Figure 7. Example of layer aspect being reused

### B. Discussion

Reverse engineering is a fundamental part of the reengineering process, since understanding the system is an important step before changing or reconstructing it. The usage of our pattern allowed us to recover lost information (the translated design) in OSGi, and also facilitated the reuse of that abstraction, thus achieving essential goals of reengineering. Although this article focuses only on one architectural pattern, software layers, we could illustrate how to use aspects to capture an architectural abstraction without needing to restructure the code, which in addition can be reused to apply other crosscutting concerns. The lack of an automated approach was a major drawback that required the manual analysis of the target system code. This would represent an obstacle for applying such approach in systems with significant size. Therefore, the development of auxiliary tools for applying that reengineering pattern would significantly improve the efficiency of using such approach.

## V. Related Work

Other approaches, such as [5][12][14][16] and [20], employed pattern-based reverse engineering, which consists of detecting design patterns in software. An important motivation for providing such mechanisms is that patterns provide a common idiom for developers. Therefore by understanding what patterns where employed, the effort necessary to understanding the whole software will be reduced [20]. These approaches help identifying the architectural elements based on the recognition of patterns. Although a method for software architecture reconstruction is discussed in [12], the process is based on design patterns recognition. In summary, most of the above strategies try to automate the lookup of the more traditional

 ISBN: 978-1-61208-438-1

design patterns [9], with tools inferring patterns based on graph analysis and visual tools showing such relationships and pattern match.

The work in [5] slightly differs from such approaches because it allows looking for anti-patterns and "bad smells" that may negatively affect the architecture recovery. In contrast to our work, although the previously mentioned approaches provide a sort of (semi-) automated discovery of patterns, they are rather focused on a fine grain perspective of patterns (i.e., design patterns), while we intend to employ a strategy that gives us a coarse grain perspective of an architectural pattern (currently limited do software layers).

The relationship between patterns and AOP that we found in literature mainly deals with the implementation of design patterns with the help of AOP [13], and studies that analyze impacts and drawbacks of such implementations [2][11][23]. Under the perspective of software architectures, for instance, some research efforts focused on establishing a way to represent aspects in the software architecture early in the design phase, using aspect-oriented architectural models [8] – sometimes identifying them even earlier, during the requirements elicitation phase [22] – and more specific forms of expressing them such as the definition of representations of aspects using the Unified Modeling Language (UML), as found in [18] and [25]. Another example in the architectural level is that of specific Architecture Description Languages [10][19][21] that are able to express aspects and other crosscutting concerns. However, the above cases of aspect usage focus on how to represent in the architecture the aspects that crosscut the system elements (e.g., components, modules, subsystems), while our approach uses an aspect abstraction to mimic an architectural pattern.

Specifically talking about the layers architectural pattern, the only study we have found explicitly dealing with software layers and AOP was performed in [24]. However, that report deals with software layers and aspects using a perspective that differs from our work. Their approach consists in the assessment of the impact of using AOP on layered software architectures.

## VI. Conclusions and Future Work

The reengineering of systems may be motivated by different reasons, such as lack of modularity, improper layering, duplicate code or functionality. Refactoring with the help of aspect-oriented programming provides a way of performing reengineering by employing the separation of concerns principle. It allows cross-cutting concerns to be separated from the application, allowing better maintenance and readability.

This article proposed the usage of aspects in a novel way. It was used to provide an abstraction that provides a correct perspective of a layer architecture that was not well represented in the system code. It refactors specific aspects in order to use such abstractions instead of targeting the system code, and we also propose a reengineering pattern describing such process.

In the case study, the usage of aspects allowed us to abstract logical layers that were scattered over the OSGi API, providing a vision that disentangles the OSGi layers from the interfaces and classes that accumulate responsibilities from different layers. The resulting aspectized layers were reused for applying concrete aspects that concerned dependability. Since analyzing a single case study may be a limitation of the generalized perspective provided in this article, for future work we plan to apply this pattern in other systems for evaluating its occurrence, as well as getting a deeper understanding of its advantages and drawbacks. Another interesting path to take is to evaluate how employ aspects to recover and represent other architectural patterns.

## Acknowledgement

This work was supported by INES - Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (http://www.ines.org.br/) - with financial support from CNPq (CNPq grant #573964/2008-4). Kiev Gama was also supported by the CNPq grant #485420/2013. The work presented here was initially carried out as part of the ASPIRE project, co-funded by the European Commission under the FP7 programme, contract #215417.

## References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns", Wiley, 1996, ISBN: 978-0-471-95869-7

[2] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena, "Composing design patterns: a scalability study of aspect-oriented programming". In Proceedings of the 5th international conference on Aspect-oriented software development. ACM, 2006, pp. 109-121, ISBN:1-59593-300-X, doi:10.1145/1119655.1119672

[3] E. Chikofsky and J. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy". IEEE Software 7, 1, January 1990, pp. 13-17, ISSN:0740-7459, doi:10.1109/52.43044

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Object Oriented Reengineering Patterns". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002, ISBN: 978-3952334126

[5] M. Detten and S. Becker, "Combining clustering and pattern detection for the reengineering of component-based software system" Joint ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium -- ISARCS on Quality of software architectures -- QoSA and architecting critical systems. ACM, New York, NY, USA, 2011, pp. 23-32, ISBN: 978-1-4503-0724-6, doi:10.1145/2000259.2000265

[6] E. Dijkstra, "On the role of scientific thought", EWD 447, 1974, appears in E.W.Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer Verlag, 1982

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999, ISBN: 978-0201485677

[8] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modeling" Software, IEE Proceedings-Vol. 151, No. 4, 2004, pp. 173-185, doi: 10.1049/ip-sen:20040920

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Menlo Park, CA, 1995, ISBN: 978-0201633610

[10] A. Garcia, C. Chavez, T. Batista, C. Sant'Anna, U. Kulesza, A. Rashid, and C. Lucena, "On the modular representation of architectural aspects" Software Architecture, Springer Berlin

 ISBN: 978-1-61208-438-1 

Heidelberg, 2006, pp. 82-97, ISBN 978-3-540-69272-0 doi: 10.1007/11966104_7

[11] A. Garcia et al. "Modularizing design patterns with aspects: a quantitative study" Transactions on Aspect-Oriented Software Development I, 2006, pp. 36-74, ISBN: 1-59593-042-6 doi:10.1145/1052898.1052899

[12] G.Y Guo, J. M. Atlee, and R. Kazman, "A software architecture reconstruction method". In Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), Kluwer, The Netherlands, 2006, pp. 15-34, ISBN:0-7923-8453-9

[13] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ". Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '02), ACM SIGPLAN Notices, 2002, pp. 161-173, ISBN:1-58113-471-1 doi:10.1145/582419.582436

[14] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-Based Reverse-Engineering of Design Components" 21st International Conference on Software Engineering, IEEE Computer Society Press, May 1999, pp 226–235, ISBN: 1-58113-074-0

[15] G. Kiczales, G., et al. "Aspect-Oriented Programming" European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Finland, 1997, ISBN: 978-3-540-69127-3, doi: 10.1007/BFb0053381

[16] J. Niere, W. Shafer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery" International Conference on Software Engineering, IEEE Computer Society Press, May 2002, pp. 338–348, ISBN: 1-58113-472-X

[17] OSGi Alliance. OSGi Service Platform. http://www.osgi.org [retrieved: 09, 2015]

[18] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli, "A UML notation for aspect-oriented software design" Proceedings of the AOM with UML workshop at AOSD, Vol. 2002.

[19] N Pessemier, L. Seinturier L., T. Coupaye, and L. Duchien, "A model for developing component-based and aspect-oriented systems" Software Composition. Springer Berlin Heidelberg, 2006, pp. 259-274, ISBN: 978-3-540-37659-0, doi: 10.1007/11821946_17

[20] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineering of design patterns" Software Systems Modeling, 2005, pp. 55–70, ISSN: 1619-1374 , doi: 10.1007/s10270-004-0059-9

[21] M. Pinto and L. Fuentes, "AO-ADL: An ADL for describing aspect-oriented architectures". Early Aspects: Current Challenges and Future Directions, 2007, pp. 94-114, doi:10.1007/978-3-540-76811-1_6

[22] A. Rashid, P. Sawyer, A. Moreira, and J. Araújo, "Early aspects: A model for aspect-oriented requirements engineering" IEEE Joint International Conference on Requirements Engineering, 2002, pp. 199-202, ISSN: 1090-705X, doi: 10.1109/ICRE.2002.1048526

[23] C. Sant'Anna, A. Garcia, U. Kulesza, C. Lucena, and A. V. Staa, "Design patterns as aspects: A quantitative assessment" Journal of the Brazilian Computer Society, 10(2), 2004, pp. 42-55, ISSN: 1678-4804, doi: 10.1007/BF03192358

[24] J. Saraiva, F. Castor, S. and Soares, "Assessing the Impact of AOSD on Layered Software Architectures" ECSA 2010, LNCS 6285, 2010, pp. 344–351, doi: 10.1007/978-3-642-15114-9_27

[25] J. Suzuki and Y. Yamamoto, "Extending UML with aspects: Aspect support in the design phase" Lecture Notes in Computer Science, Springer-Verlag London, UK, 1999, pp. 299-300, ISBN:3-540-66954-X

 