Using Normalized Systems Patterns as Knowledge Management

Peter De Bruyn, Philip Huysmans, Gilles Oorts,
Dieter Van Nuffel, Herwig Mannaert and Jan Verelst
Normalized Systems Institute (NSI)
University of Antwerp
Antwerp, Belgium
{*peter.debruyn,philip.huysmans,gilles.oorts,dieter.vannuffel,*
herwig.mannaert,jan.verelst}*@ua.ac.be*

Arco Oost
Normalized Systems eXpanders factory (NSX)
Antwerp, Belgium
{*arco.oost*}*@nsx.normalizedsystems.org*

***Abstract*—The knowledge residing inside a firm is frequently considered to be one of its most important internal assets to obtain a sustainable competitive advantage. Also in software engineering, a substantial amount of technical know-how is required in order to successfully deploy the organizational adoption of the technology. In this paper, we focus on the wide-spread approach of using design patterns for knowledge management purposes. It is discussed how they facilitate the transfer and (re)use of state-of-the-art knowledge in three dimensions: (1) more efficient documentation, (2) the development of new applications, and (3) the incorporation of new knowledge in existing applications. More specifically, we show how the use of Normalized Systems elements can be considered as an advanced form of design patterns for the development of highly evolvable software architectures, further enhancing the inherent design patterns advantages. Normalized Systems captures software engineering knowledge in a limited set of theorems and patterns, and enables the application of this knowledge through systematic pattern expansion. Because of the highly structured way of working, the reuse of knowledge can be significantly improved.**

***Keywords-Normalized Systems*; *Design Patters*; *Knowledge Management*.**

I. INTRODUCTION

As an important movement within the strategic management literature, the resource-based view of the firm (RBV) states that internal resources (e.g., money, patents, buildings, geographical location, etcetera) are the key elements for organizations in order to obtain a sustainable competitive advantage [1]. More specifically, the *knowledge* residing inside a firm is frequently considered to be its most important internal asset [2]. Further, focusing on the case of software adoption and development within organizations, the prevalence of the available knowledge becomes even more clear and the need for knowledge management practices in this respect have been acknowledged frequently [3]. Indeed, information technology in general can be considered as a knowledge-intensive or complex technology innovation, requiring a substantial amount of know-how and technical knowledge by the adopting firm [4]. As such, the degree of expertise or advanced knowledge of best-practices regarding a certain software technology becomes a decisive factor in the chances for an organization to successfully deploy and manage it. Consequently, a firm should either already (i.e., prior to the adoption) possess the advanced knowledge required to operate the software technology or engage in *organizational learning* during exploitation.

Organizational learning is generally regarded as the result of individual learning experiences of members of an organization, which become incorporated into the behavior, routines and practices of the organization the individuals belong to [4]. According to Levitt and March [5], such an organizational learning can occur in two general ways: (1) "learning by doing", which involves a learning process by self-experienced trial-and-error and (2) learning from the direct experiences of other people. While the first type of learning is typically a very profound and thorough way of knowledge gathering, it can be time-consuming, expensive and error-prone in the earliest stages. At this point, know-how, experiences and best-practices formulated by other users (i.e., the second type of organizational learning) come into play. Inside organizations, such knowledge transfers in software development can occur in many different ways, including for example explicit knowledge bases or experience repositories [6], "yellow pages" enabling search actions for accessible knowledgeable people [7] and mentoring programs [8]. At the inter-organizational or industrial level, the gathered knowledge can benefit from experience based on many different development projects. Design patterns are a wide-spread approach to achieve this goal [9]. In this paper, we explore three types of benefits when using knowledge captured by design patterns:

- Improved documentation;
- Using the captured knowledge to build new applications;
- and Incorporating new knowledge into existing applications.

We introduce the Normalized Systems (NS) theory, which proposes more concrete and structured patterns. We argue that the use of knowledge captured in such patterns can further enhance the discussed benefits of applying design

 ISBN: 978-1-61208-230-1

patterns.

II. Knowledge Management in Software Engineering

The specific use of design patterns in object-orientation during the 90's, exemplified by the seminal work of Gamma et al. [10], was incited by the fact that modern computer literature regularly failed to make tacit (but success determining) knowledge regarding low-level principles of good software design explicit [11]. Patterns provide high-level solution templates for often-occurring problems. The patterns proposed by Gamma et al. [10] were conceived as the bundling of a set of generally accepted high-quality and best-practice solutions to frequently occurring problems in object-orientation programming environments. For instance, in order to create an one-to-many dependency between objects so that when the state of one object changes, all its dependents are notified and automatically updated, the observer pattern (i.e., an overall structure of classes giving a description or template of how to solve the concerned problem) was proposed [10]. As a consequence, the use of these patterns can be considered as specifically aimed at facilitating (inter-)organizational learning by learning from direct experiences of other people — in this case experienced software engineers —, and being one specific way of knowledge base distribution.

According to Schmidt [12], design patterns have been so successful because they explicitly capture knowledge that experienced developers already understand implicitly. The captured knowledge is called implicit because it is often not captured adequately with design methods and notations. Instead, it has been accumulated through timely processes of trial and error. Capturing this expertise allows other developers to avoid spending time rediscovering these solutions. Moreover, the captured knowledge has been claimed to provide benefits in several areas [13]. In this paper, we focus on the usage of patterns to (a) document software code, (b) build new applications, and (c) incorporate new knowledge in existing software applications.

A. Documentation

Patterns provide developers with a vocabulary which can be used to document a design in a more concise way [10], [13], [14]. For example, pattern-based communication can be used to preserve design decisions without elaborate descriptions. By delineating and naming groups of classes which belong to the same pattern, the descriptive complexity of the design documentation (e.g., a UML class diagram) can be reduced [14]. Consequently, the vocabulary offered by patterns allows a shift in the abstraction level of the discussions. This usage of design patterns is mostly applied at the conceptual level, and neglects the source code documentation. However, the abstract nature of patterns, i.e., as a solution template, means that it is possible to implement a certain design pattern using different alternatives. Therefore, it has been argued that the addition of source-code level documentation of the pattern usage is required to perform coding and maintenance tasks faster and with fewer errors [15].

B. Using knowledge to build new applications

Several authors propose the usage of design patterns to create new software applications (e.g., [16]). We discussed above how patterns provide high-level solution templates, and, as such, do not dictate the actual source code. Consequently, knowledge concerning the implementation platform remains important. A correct and efficient implementation of a design pattern requires a careful selection of language features [12]. Clearly, design patterns alone are not sufficient to build software. As a result, the implementation of a design pattern during a software development process remains essentially a complex and activity [12]. Developing software for a concrete application then requires the concrete experience of a domain and the specifics of the programming language, as well as the ability to abstract away from details and adhere to the structure prescribed by the design pattern. Nevertheless, certain companies and researchers attempt to integrate the knowledge available in design patterns in other approaches, in order to create automated code generation. For example, so-called software factories attempt to create software similar to automated manufacturing plants [17]. This should drastically improve software development productivity. However, such approaches have not yet reached wide-spread adoption.

C. Incorporating new knowledge in existing applications

Because of the increasing change in the organizational environment in which software applications are used, adaptability is considered to be an important characteristic. However, adapting software remains a complex task. Various studies have shown that the main part of the software development cost is spent after the initial deployment [18]. Several design patterns focus on incorporating adaptability into their solution template. Empirical observations have been reported which confirm the increased adaptability when using design patterns [19]. Adaptations could be made easier in comparison with an alternative which was programmed using no design patterns, and achieved adaptability was retained more successfully because of the prescribed structure. Nevertheless, some researchers also report negative effects on adaptability, caused by the added complexity of the design patterns. By prescribing additional classes in comparison to simpler solution, more errors have been introduced in some cases [19].

III. Normalized Systems

The Normalized Systems (NS) theory starts from the postulate that software architectures should exhibit *evolvability*

 ISBN: 978-1-61208-230-1

due to ever changing business requirements, while many indications are present that most current software implementations do not conform with this evolvability requisite. Evolvability in this theory is operationalized as being the absence of so-called *combinatorial effects*: changes to the system of which the impact is related to the size of the system, not only to the kind of the change which is performed. As the assumption is made that software systems are subject to unlimited evolution (i.e., both additional and changing requirements), such combinatorial effects are obviously highly undesirable. In case changes are dependent on the size of the system and the system itself keeps on growing, changes proportional to the systems size become ever more difficult to cope with (i.e., requiring more efforts) and hence hampering evolvability. Normalized Systems theory further captures its software engineering knowledge by offering a set of four theorems and five elements, and enables the application of this knowledge through pattern expansion of the elements. The theorems consist of a set of formally proven principles which offer a set of necessary conditions which should be strictly adhered to, in order to obtain an evolvable software architecture (i.e., in absence of combinatorial effects). The elements offer a set of predefined higher-level structures, primitives or "building blocks" offering an unambiguous blueprint for the implementation of the core functionalities of realistic information systems, adhering to the four stated principles.

A. Theorems

Normalized Systems theory proposes four theorems, which have been proven to be necessary conditions to obtain software architectures in absence of combinatorial effects:

- *Separation of Concerns*, requiring that every change driver (concern) is separated from other concerns in its own construct;
- *Action Version Transparency*, requiring that data entities can be updated without impacting the entities using it as an input or producing it as an output;
- *Data Version Transparency*, requiring that an action entity can be upgraded without impacting its calling components;
- *Separation of States*, requiring that each step in a workflow is separated from the others in time by keeping state after every step.

In terms of knowledge management, as mentioned explicitly in [20], it must clearly be noted that the design theorems proposed are not new themselves; in fact, they relate to well-known (but often tacit or implicit) heuristic design knowledge of experienced software developers. For instance, well-known concepts such as an integration bus, a separated external workflow or the use of multiple tiers can all be seen as manifestations of the Separation of Concerns theorem [20]. As such, the added value of the theorems should then rather be situated in the fact that they (1) make certain aspects of that heuristic design knowledge explicit, (2) offer this knowledge in an unambiguous way (i.e., violations against the theorems can be proven), (3) are unified based on one single postulate (i.e., the need for evolvable software architectures having no combinatorial effects) and (4) have all been proven in a formal way.

B. Normalized Systems Elements as Patterns

The above stated theorems illustrate that typical software primitives do not offer explicit mechanisms to incorporate the principles. Also, the systematic application of the principles leads to a very fine-grained modular structure, which could form an additional design complexity on its own when performed "from scratch". Therefore, NS theory proposes a set of five elements as encapsulated higher-level patterns complying with the four theorems:

- *data elements*, being the structured encapsulation of a data construct into a data element (having get- and set-methods, exhibiting version transparency, etcetera);
- *action elements*, being the structured encapsulation of an action construct into an action element;
- *workflow elements*, being the structured encapsulation of software constructs into a workflow element describing the sequence in which a set of action elements should be performed in order to fulfill a flow;
- *connector elements*, being the structured encapsulation of software constructs into a connector element allowing external systems to interact with the NS system without calling components in a stateless way;
- *trigger elements*, being the structured encapsulation of software constructs into a trigger element controlling the states of the system and checking whether any action element should be triggered accordingly.

Each of the elements is a pattern as they represent a recurring set of constructs: besides the intended, encapsulated core construct, also a set of relevant cross-cutting concerns (such as remote access, logging, access control, etcetera) is incorporated in each of these elements. For each of the patterns, it is further described in [20] how they facilitate a set of anticipated changes in a stable way. In essence, these elements offer a set of building blocks, offering the core functionalities for contemporary information systems.

Regarding these patterns, it can be noted that their separate definition and identification is based on the implications of the set of theorems. For instance, the theorems Separation of Concerns and Separation of States indicate the need to formulate a workflow element next to an action element, in order to allow for the stateful invocation of action elements in a (workflow) construct other than action elements containing functional tasks. Next, each of the five patterns themselves contain knowledge concerning all the implications of the theorems referred to in Section III-A. Finally, each of these patterns has been described in a very detailed way. Consider for instance a data element in a JEE

 ISBN: 978-1-61208-230-1

implementation [21]. In [20] it is discussed how a data element `Obj` is associated with a bean class `ObjBean`, interfaces `ObjLocal` and `ObjRemote`, home interfaces `ObjHomeLocal` and `ObjHomeRemote`, transport classes `ObjDetails` and `ObjInfo`, deployment descriptors and EJB-QL for finder methods. Additionally, methods to manipulate a data element's bean class (create, delete, etcetera) and to retrieve the two serializable transport classes are incorporated. Finally, to provide remote access, an agent class `ObjAgent` with several lifecycle manipulation and details retrieval methods is included. It can be argued that these elements incorporate the main concerns which are relevant for their function.

Moreover, the complete set of elements covers the core functionality of an information system. Consequently, as such detailed description is provided for each of the five elements, an NS application can be considered as an aggregation of a set of instantiations of the elements. Consider for example the implementation of an observer design pattern [10]. In order to implement this pattern in NS, three data elements (i.e., `Subscriber`, `Subscription` and `Notification`) are required. A `Notifier` connector element will observe the subject, and create instances of the `Notification` data element. These `Notification` data elements will be sent to every `Subscriber` that has a `Subscription` through a `Publisher` connector element. The sending is triggered by a `PublishEngine` trigger element which will periodically activate a `PublishFlow` workflow element. Consider that each (NS) element consists of around ten classes [22]. The seven identified elements therefore result in around seventy classes used to implement the design pattern, whereas the original implementation of the design pattern consists of two classes and two interfaces. Consequently, it is clear that, in order to prevent combinatorial effects, a very fine-grained modular structure needs to be adhered to.

C. Pattern Expansion

As stated before, in practice, the very fine-grained modular structure implied by the NS principles seems very unlikely to arrive at without the use of higher-level primitives or patterns. Consequently, as NS proposes a set of five elements which serve for this purpose, the actual software architecture of NS conform software applications can actually be generated relatively straightforward. For example, in case of the data element pattern structure, the pattern expansion mechanism would need a set of parameters including the basic name of the data element (e.g., `Invoice`), context information (e.g., component and package name) and data field information (e.g., data type). Next, based on these parameters, the pattern expansion mechanism will generate the predefined structured (i.e., the set of classes and data fields) as illustrated above: the bean class `InvoiceBean`, interfaces `InvoiceLocal` and `InvoiceRemote`, etcetera.

However, in terms of knowledge management, it should be noted that the patterns and the expansion mechanism should not be considered as separate knowledge reuse mechanisms: rather, the pattern expansion facilitates the re-use of knowledge embedded in the patterns, as each expansion of the patterns results in a new application of the knowledge encapsulated in the pattern. Through this, pattern expansion facilitates both types of learning discussed earlier (i.e., "learning by doing" and learning from experience of other people) by utilizing the knowledge contained in the patterns.

Also, the information codified in a pattern may not be sufficient to adequately transfer the intended knowledge. This was already the case when using the design patterns proposed by Gamma et al. [10]. For example, it has been claimed that the *Dependency Inversion Principle* helps to gain a better understanding of the *Abstract Factory* pattern [23]. Similarly, the structure of the NS patterns can only be understood when the NS theorems are taken into account.

IV. Normalized Systems Patterns as Knowledge Management

In the previous sections, we explained how traditional design patterns entail knowledge management related benefits regarding documentation, the development of new applications and the adaptation of existing applications. We also argued how NS patterns represent a fine-grained modular structure which can be expanded to provide an evolvable software architecture. In this section, we discuss how the use of NS patterns seems to even further enhance the reported design pattern benefits, when compared to design patterns.

A. Documentation

As NS-compliant applications based on the NS elements have basically five recurrent elementary structures, only these five elements have to be understood to grasp the structure of each instantiated element throughout the application. Because the patterns are detailed enough to be instantiated, no manual implementation of the patterns (as is the case with the design patterns proposed by Gamma et al. [10]) is required. Consequently, an identical code structure reoccurs in every application which is created using the NS expanders. The commonality of the structure of the patterns makes that once one understands the patterns, one understands all its instantiations as well. In this way, it could be argued that — at least partially — the pattern structure becomes the documentation. Therefore, no source code level documentation is required.

B. Using knowledge to build new applications

As (1) each violation of the NS theorems during any stage of the development stage results in a combinatorial effect, and (2) the systematic application of these theorems results in very fine-grained structures, it becomes extremely challenging for a human developer to consistently obtain

 ISBN: 978-1-61208-230-1

such modular structures. Indeed, the fine-grained modular structure might become a complexity-issue on its own. In this sense, the NS patterns might offer the necessary simplification by offering pre-constructed structures ("building blocks"), which can be parameterized during implementation efforts. This way the NS patterns do dictate the source code for implementing the pattern, contrary to the patterns of Gamma et al. [10].

An important characteristic of these structures is that they separate technology-dependent aspects from the actual implementation, resulting in the fact that one can easily switch the underlying technology stack of the software. One transition that has been performed, is changing the underlying implementation architecture from EJB 2 to EJB 3. Because these standards use a different way of communicating between agents and beans, this transition normally is a labor-intensive and difficult task. Using the architecture described in this paper, this transition can however be achieved rather easily by using the pattern expansion mechanism. This is because the expanders that perform the expansion are very similar for different technologies. This is done by clearly separating functional requirements of the system (i.e., input variables, transfer functions and output variables) from constructional aspects of the system (i.e., composition of the system). Whereas all constructional aspects are described in patterns and expanders, functional aspects are separately included in descriptor files (such as data elements, action elements, etc.). As each pattern can be conceived a recurring structure of programming constructs in a particular programming environment (e.g., classes), one can conclude that the functional/constructional transformation then becomes located at one abstraction level higher than before.

C. Incorporating new knowledge in existing applications

The purpose is to easily incorporate new knowledge of improvements, intrinsically by the use of the elements. This can be interpreted from two distinct perspectives. First, improvements or changes (e.g., typical bug fixing or a new kind of algorithm) regarding the actual functional parts of the system (i.e., the so-called 'tasks') are easily to be incorporated in the whole system as the properly separated change driver is the only place where any modifications have to be made and the remainder of the system can easily interact with the new task (and hence, use this knowledge). In NS terms, we could call these kind of changes and expertise inclusions, knowledge dispersion at the "*sub-modular level*" as only changes and new knowledge are incorporated at the sub-modular level of the tasks (and not in the modular structure of the elements). Second, however, knowledge can be incorporated at the "*modular level*" as well. This kind of knowledge inclusion would include change (e.g., an extra separated class in the pattern) and modifications (e.g., improved persistence mechanism) regarding the internal structure of an element (the pattern). Indeed, once the basic structure or cross-cutting concern implementation of an element is changed due to a certain identified need or improvement, the new best-practice knowledge can be expanded throughout the whole (existing) modular structure and used for new (i.e., additional) instantiations of the elements. In order to further illustrate this second kind of knowledge dispersion based on NS patterns, consider the following example, based on real-life experience from developers using NS.

For instance, one way to adopt a model-view-controller (MVC) architecture in a JEE distributed programming environment is by adopting (amongst others) the Struts framework. In such MVC architecture, a separated controller is responsible for handling an incoming request from the client (e.g., a user via a web interface) and will invoke (based on this request) the appropriate model (i.e., business logic) and view (i.e., presentation format), after which the result will eventually be returned to the client. Struts is a framework providing the controller (ActionServlet) and enabling the creation of templates for the presentation layer. Obviously, security issues need to be handled properly in such architecture as well. Applied to our example, these security issues in Struts were handled in the implementation of the Struts Action itself in a previous implementation of our elements. In other words, the implementation class itself was responsible for determining whether or not a particular operation was allowed to be executed (based on information such as the user's access rights, the screen in which the action was called, etcetera). As such, this "security function" became present in all instantiations of an action element type (i.e., each session). Moreover, this resulted in a combinatorial effect as the impact of a change such as switching towards an equivalent framework (i.e., handling similar functions as Struts), would entail a set of changes dependent on the number of instantiated action elements (and hence, on the size of the system). In order to solve the identified combinatorial effect, the Separation of Concern theorem has to be applied: separating the part of the implementation class responsible for the discussed security issues (i.e., a separate change driver) in its own module within the action element. In our example, a separate interceptor module was implemented, next to the already existing implementation class. This way, not only the combinatorial effect was excluded, but the new knowledge in terms of a separate interceptor class was applied to all action elements after isolating the relevant implementation class parts and executing the pattern expansion. Additionally, all new applications using the new action element structure automatically incorporate this new knowledge.

Hence, compared to traditional design patterns, the NS patterns offer a formally proven evolvable software architecture as well as an convenient knowledge distribution mechanism.

 ISBN: 978-1-61208-230-1

V. Conclusion and Future Work

In this paper, we indicated the application and usefulness of patterns in software development. It was also shown that Normalized Systems theory can readily be considered as a method of building stable and large-scale information systems. Furthermore it has been demonstrated how Normalized Systems theory uses patterns to facilitate the transfer and use of knowledge on software development. But far most we showed in this paper that the NS elements can be considered to be enhanced patterns for software development with benefits on three dimensions (i.e., less need for explicit documentation, more deterministic development of new applications and more convenient incorporation of new knowledge into existing applications). From interviews with developers, these benefits have shown to enhance the transfer of knowledge, success rate and the overall quality of NS developments. Although the discussion in this paper was limited to Normalized Systems theory for software, the theory has recently been applied to both Business Process Management and Enterprise Architecture domains. As part of future research, the possible formulation of patterns on the level of business processes and enterprise architecture will be studied.

Acknowledgment

P.D.B. is supported by a Research Grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[1] B. Wernerfelt, "A resource-based view of the firm," *Strategic Management Journal*, vol. 5, no. 2, pp. 171–180, 1984.

[2] R. M. Grant, "Toward a Knowledge-Based Theory of the Firm," *Strategic Management Journal*, vol. 17, pp. 109–122, 1996.

[3] F. O. Bjørnson and T. Dingsøyr, "Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used," *Information and Software Technology*, vol. 50, no. 11, pp. 1055–1068, 2008.

[4] P. Attewell, "Technology diffusion and organizational learning: The case of business computing," *Organization Science*, vol. 3, no. 1, pp. 1–19, 1992.

[5] B. Levitt and J. G. March, "Organizational learning," *Annual Review of Sociology*, vol. 14, pp. 319–340, 1988.

[6] C. Chewar and D. McCrickaerd, "Links for a human-centered science of design: integrated design knowledge environments for a software development process," in *Proceedings of the Hawaii International Conference on System Sciences*, 2005.

[7] T. Dingsøyr, H. K. Djarraya, and E. Røyrvik, "Practical knowledge management tool use in a software consulting company," *Communications of the ACM*, vol. 48, no. 12, pp. 96–100, 2005.

[8] F. Bjørnson and T. Dingsøyr, "A study of a mentoring program for knowledge transfer in a small software consultancy company," in *Product Focused Software Process Improvement*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3547, pp. 245–256.

[9] I. Rus and M. Lindvall, "Knowledge management in software engineering," *IEEE Software*, vol. 19, pp. 26–38, 2002.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[11] J. Coplien, "The culture of patterns," *Computer Science and Information Systems*, vol. 1, no. 2, pp. 1–26, 2004.

[12] D. C. Schmidt, "Using design patterns to develop reusable object-oriented communication software," *Commun. ACM*, vol. 38, no. 10, pp. 65–74, Oct. 1995.

[13] D. Riehle, "Transactions on pattern languages of programming ii," J. Noble and R. Johnson, Eds. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Lessons learned from using design patterns in industry projects, pp. 1–15.

[14] G. Odenthal and K. Quibeldey-Cirkel, "Using patterns for design and documentation," in *ECOOP*, 1997, pp. 511–529.

[15] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 595–606, 2002.

[16] C. Larman, *Applying UML and Patterns*. Prentice Hall, 1997.

[17] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03, 2003, pp. 16–27.

[18] R. L. Glass, "Maintenance: Less is not more," *IEEE Software*, vol. 15, no. 4, pp. 67–68, 1998.

[19] L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1134 –1144, dec 2001.

[20] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, pp. 89–116, 2011.

[21] Oracle. Java platform, enterprise edition. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[22] H. Mannaert and J. Verelst, *Normalized systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009.

[23] L. Welicki, J. Manuel, C. Lovelle, and L. J. Aguilar, "Patterns meta-specification and cataloging: towards knowledge management in software engineering," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06, 2006, pp. 679–680.

 ISBN: 978-1-61208-230-1