Towards the Explicitation of Hidden Dependencies in the Module Interface

Dirk van der Linden
Electro Mechanics Research Group
Artesis University College of Antwerp
Antwerp, Belgium
dirk.vanderlinden@artesis.be

Herwig Mannaert, Peter De Bruyn
Department of Management Information Systems
University of Antwerp
Antwerp, Belgium
herwig.mannaert, peter.debruyn@ua.ac.be

***Abstract*—Balancing between the desire for information-hiding and the risk of introducing undesired hidden dependencies is often not straightforward. Hiding important parts of the internal functionality of a module is known as the *black box* principle, and is associated with the property of reusability and consequently evolvability. An interface, which is roughly explaining the core functionality of a module, helps indeed the developer to use the functionality without being forced to concentrate on the implementation details. However, some implementation details should not be hidden if they hinder the use of the module when the environment changes. These kind of implementation details can be called undesired hidden dependencies. An interesting question then becomes, which information should be hidden and which not? In this paper, we use the Normalized Systems theorems as a base to evaluate which details should be hidden versus transparent in order to improve reusability. In other words, which kind of information encapsulation contributes towards safe black box reuse?**

***Keywords*-Normalized Systems; Reusability; Evolvability; Systems Theory; Modularity; Black Box.**

I. Introduction

Modern technologies provide us capabilities to build large, compact, powerful and complex systems. Without any doubt, one of the major key points is the concept of modularity. Systems are built as structured aggregations of lower-level subsystems, each of which have precisely defined interfaces and characteristics. In hardware for instance, a USB memory stick can be considered a module. The user of the memory stick only needs to know its interface, not its internal details, in order to connect it to a computer. In software, balancing between the desire for information-hiding and the risk of introducing undesired hidden dependencies is often not straightforward. Experience contributes in learning how to deal with this issue. In other words, best practices are rather derived from heuristic knowledge than based on a clear, unambiguous theory.

Normalized Systems theory has recently been proposed [1] to contribute in translating this heuristic knowledge into explicit design rules for modularity. In this paper, we want to evaluate which information-hiding is desired and which is not with regard to the theorems of Normalized Systems.

The authors of this paper have each a different implementation focus (business process software versus automation control software), with different programming languages and development environments (JAVA [2] versus IEC 61131-3 [3]). In this collaboration we want to study fundamental principles, which should be independent of implementation focus. With regard to this independence, the different implementation focus of the authors might be an advantage. Moreover, at some point the need for combining these disciplines is arising. Automation systems have to be upgraded to new communication protocols and to provide new processing rules, as the interconnection of different grids will be forced in future [4].

Doug McIlroy called for *families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able safely to regard components as black boxes* [5]. Decades after the publication of this vision, we have black boxes, but it is still difficult to guarantee that users can use them safely. However, we believe that all necessary knowledge is available, we only have to find all the necessary unambiguous rules to make this (partly tacit) knowledge explicit.

Scientific research groups contribute in converting tacit knowledge to theorems and fundamental rules, like the authors of Normalized Systems did. In addition, industrial working groups contribute in converting tacit knowledge to standards and specifications. For example, the OPC UA working groups provide the concept of OPC UA profiles. Profiles define the functionality of an OPC UA application [6]. Software Certificates contain information about the supported Profiles. OPC UA Clients and Servers can exchange these certificates via services.

The paper is structured as follows. In Section II, the Normalized Systems theory will be discussed. In Section lII, we give an overview of the most commonly discussed kinds of coupling, and evaluate whether they comply with the Normalized Systems theorems or not. In Section lV, we make suggestions on how we should deal with undesired hidded dependencies. Finally, conclusions and future research are discussed in Section V.

II. Normalized Systems

The current generation of systems faces many challenges, but arguable the most important one is evolvability [7]. The evolvability issue of a system is the result of the existence of Lehman's Law of Increasing Complexity which states: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" [8]. Starting from the concept of systems theoretic stability, the Normalized Systems theory is developed to contribute towards building systems, which are immune against Lehman's Law.

A. *Stability*

The postulate of Normalized Systems states that *a system needs to be stable with respect to a defined set of anticipated changes*. In systems theory, one of the most fundamental properties of a system is its stability: a bounded input function results in bounded output values, even for T $\rightarrow \infty$ (with T representing time).

The impact of a change should only depend on the nature of the change itself. Systems, built following this rule can be called stable systems. In the opposite case, changes causing impacts that are dependent on the size of the system, are called *combinatorial effects*. To attain stability, these combinatorial effects should be removed from the system. Systems that exhibit stability are defined as *Normalized Systems*. Stability can be seen as the requirement of a linear relation between the cumulative changes and the growing size of the system over time. Combinatorial effects or instabilities cause this relation to become exponential (Figure 1). The design theorems for Normalized Systems contribute to the long term goal of keeping this relation linear for an unlimited period of time, and an unlimited amount of changes to the system.

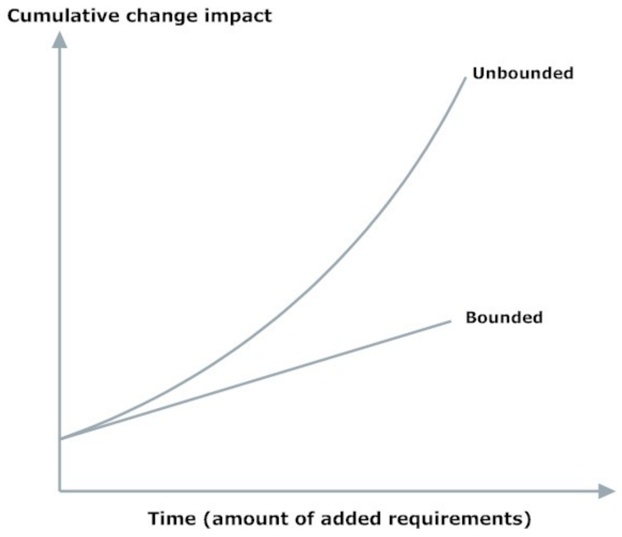

Figure 1. Cumulative impact over time

B. *Design Theorems for Normalized Systems*

In this section, we give an overview of the design theorems or principles of Normalized Systems, i.e., systems that are stable with respect to a defined set of anticipated changes:

- A new version of a data entity
- An additional data entity
- A new version of an action entity
- An additional action entity

Please note that these changes are associated with software primitives in their most elementary form. Real-life changes or changes with regard to 'high-level requirements' [9] should be converted to these abstract, elementary anticipated changes. We were able to convert all real-life changes in several case studies to one or more of these abstract anticipated changes [10][11][12]. However, the systematic transformation of real-life requirements to the elementary anticipated changes is outside the scope of this paper.

1) *Separation of concerns:*

An Action Entity can only contain a single task in Normalized Systems.

In this theorem, we focus on how tasks are structured within processing functions. Each set of functionality, which is expected to evolve or change indepently, is defined as a change driver. Change drivers are introducing anticipated changes into the system over time. The identification of a task should be based on these change drivers. A single change driver corresponds to a single *concern* in the application.

2) *Data version transparency:*

Data Entities that are received as input or produced as output by Action Entities, need to exhibit version transparency in Normalized Systems.

In this theorem, we focus on how data structures are passed to processing functions. Data structures or *Data Entities* need to be able to have multiple versions, without affecting the processing functions that use them. In other words, Data Entities having the property of data version transparency, can evolve without requiring a change of the interface of the action entities, which are consuming or producing them.

3) *Action version transparency:*

Action Entities that are called by other Action Entities, need to exhibit version transparency in Normalized Systems.

In this theorem, we focus on how processing functions are called by other processing functions. Action Entities need to be able to have multiple versions without affecting any of the other Action Entities that call them. In other words, Action Entities having the property of action version transparency, can evolve without requiring a change of one or more Action Entities, which are connected to them.

4) Separation of states: The calling of an Action Entity by another Action Entity needs to exhibit state keeping in Normalized Systems.

In this theorem, we focus on how calls between processing functions are handled. Coupling between modules, that is due to errors or exceptions, should be removed from the system to attain stability. This kind of coupling can be removed by exhibiting state keeping. The (error) state should be kept in a separate Data Entity.

III. Evaluation of kinds of coupling

Coupling is a measure for the dependencies between modules. Good design is associated with low coupling and better reusability. However, lowering the coupling only is not specific enough to guarantee reusability. Classifications of types of coupling were proposed in the context of structured design [13]. The key question of this paper is whether a hidden dependency and therefore coupling is affecting the reusability of a module? In general, the Normalized Systems theorems identify places in the software architecture where high coupling is threatening evolvability [14]. More specifically, we will focus in this paper on several kinds of coupling and evaluate which of them is lowering or improving reusability.

A. Content coupling

Content coupling occurs when module A refers directly to the content of module B. More specifically, this means that module A changes instructions or data of module B. When module A branches to instructions of module B, this is also considered as content coupling.

It is trivial that direct references between modules prevent them from being reused separately. In terms of Normalized Systems, content coupling is a violation of the first theorem, separation of concerns.

Avoiding content coupling is not new, other rules than those of the Normalized Systems already made this clear. Decades ago, Dijkstra suggested to abolish the Go To statement from all 'higher level' programming languages [15]. Together with restricting access to the memory space of other modules, Dijkstra's suggestion contributed to exile content coupling out of most modern programming languages.

B. Common coupling

Common coupling occurs when modules *communicate using global variables*. A global variable is accessible by all modules in the system. If a developer wants to reuse a module, analyzing the code of the module to determine which global variables are used is needed. In other words, a white box view is required. Consequently, black box use is not possible.

In terms of Normalized Systems, common coupling is a violation of the first theorem, separation of concerns.

We add however, that not the *existence* but the *way of use* of global variables violates the separation of concerns theorem. In earlier work we used global variables in a proof of principle with IEC 61131-3 code, which complies with Normalized Systems [11]. The existence of global variables was needed for other reasons than mutual communication between modules (i.e., connections with process hardware). In this project, the global variables were passed via an interface from one module to the other. Some authors state that the declaration of global variables in the IEC 61131-3 environment is somewhat ambiguous [16], although we do not think this is crucial to determine the characteristic of reusability of a module (as long as there is no common coupling!).

Since the use of global variables in case of common coupling is not visible through the module's interface, each use of these global variables is considered to be a hidden dependency. And since common coupling is a violation of separation of concerns, this is an undesired hidden dependency with respect to the safe use of black boxes.

C. Control coupling

Control coupling occurs when module A influences the execution of module B by passing data (parameters). Commonly, such parameters are called flags. Whether a module with such a flag can be used as a black box depends on the fact whether the interface is explaining sufficiently the meaning of this flag for use. Obviously, if a white box view is necessary to determine how to use the flag, black box use is not possible. The evaluation of control coupling in terms of reusability is twofold. Adding a flag can introduce a slightly different functionality and improve the reuse potential. For example, if a control module of a motor is supposed to control pumping until a level switch is reached, a flag can provide the flexibility to use both a positive level switch signal and an inverted one (positive versus negative logic). On the other hand, extending this approach to highly generic functions, would lead in its ultimate form to a single function *doIt*, that would implement all conceivable functionality, and select the appropriate functionality based on arguments. Obviously, the latter would not hit the spot of reusability.

One of the key questions during the evaluation of control coupling is: how many functionalities should be hosted in one module? In terms of Normalized Systems, the principle 'separation of concerns' should not be violated. The concept of change drivers brings clarity here. A module should contain only one core task, eventually surrounded by supporting tasks. Control coupling can help to realize theorem 2 (data version transparency) and theorem 3 (action version transparency). The calling action is able to select a version of the called action based on control coupling. We conclude that *control coupling should be used for version selection only*. More details about versions and their instances are part of very recent research of which the results are in a review process on the moment of the submissing of this paper.

D. Stamp coupling

Stamp coupling occurs when module A calls module B by passing a data structure as a parameter when module B does not require all the fields in the data structure.

It could be argued that using a data structure limits the reuse to other systems where this data structure exists, whereas only sending the required variables separately does not impose this constraint. However, we emphasize that the key point of this paper does not concern *reuse* in general. Rather, it focuses on *safe reuse* specifically. The research towards safe black box reuse is more about adding contraints or defining limitations than keeping or creating possibilities. When working with separated, simple datatypes as a set of parameters, every change requires a change of the interface of the module. Since we do not consider ‘changing the interface’ as one of our anticipated changes, stamp coupling is an acceptable form of coupling. With regard to the first theorem, separation of concerns, one should keep the parameter set (Data Entity), the functionality of the module (action) and the interface separated. Keeping the interface unaffected, while the Data Entity and Action Entity are changing, can be realized with stamp coupling.

E. Data coupling

Data coupling occurs when two modules pass data using simple data types (no data structures), and every parameter is used in both modules.

Realizing theorem 3 (action version transparency) is impossible with data coupling, since the introduction of a new parameter affects the interface of the module. This newer version of the interface would not be suitable for previous action versions, and could consequently not be called a version transparent update. The addition of a parameter in the module’s interface would violate the separation of concerns principle. Changing or removing a parameter is even worse.

Note that the disadvantage of data coupling, affecting the module’s interface in case of a change, does not apply on reusing modules, which are not evolving. This can be the case when working with system functions, e.g., aggregated in a system function library. However, problems can occur when the library is updated. We will give more details about this issue in the next section.

F. Message coupling

Message coupling occurs when communication between two or more modules is done via message passing. Message passing systems have been called ‘shared nothing’ systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

The property ‘sharing nothing’ makes message coupling a very good incarnation of the separation of concerns principle. Please note that asynchronous message passing is highly preferable above synchronous message passing, which violates the separation of states principle.

IV. Implicit Dependencies

We consider the case of one developer, who has programmed a part of a modular system with an acceptable amount of coupling. Every entity of the *subsystem* complies to the theorems of Normalized Systems. The code is fulfilling a *part* of the requirements of a bigger system, which has to be built by several developers. Another part of the system is programmed by a second developer, using the same programming language and the same platform. The question is, can both developers exchange the source code of their modules and reuse them as *safe black boxes*, i.e., without the need of a white box view of their colleagues’ code? In this paper, we concentrate on the case where both developers used the same programming language, the same platform, but a different set-up of programming environment. In other words, they do not share the same - company standard - system functions in their respective programming environments.

We start our discussion with a second case, where two or more hardware developers share several ICs (Integrated Circuits) to build for example embedded systems. Can they just pick an IC out of the box and safely use it as a black box? On one hand, hardware engineers did a remarkable job with regard to the production of safe black boxes, because they do not need to know the internal details of the IC to use it. However, before usage, the user needs the information *that is printed on the IC* to estimate its expected behaviour. If the IC is very recently built as a prototype, with nothing printed on it, the user needs information of the prototype-builder to know how to use it. In other words, information about the interface has to be available in order to use the black box. Besides, the information — rarely more than a type number — which is typically printed on an IC, is referring to data sheets, explaining in more detail the black box use of the (hardware) module.

In software, we have the advantage that interfaces can be made roughly self-explaining. But in comparison with hardware, possible dependencies are introduced. Our two or more software developers who want to (re)use each others modules, made in another programming environment, should inform each other about the libraries they used. But even if they do this well, they might end up in a so-called dependency-hell. This is a colloquial term for the frustration of some software users who have installed software packages, which depend on specific versions of other software packages. It involves for example package A needing package B & C, and package B needing package F, while package C is incompatible with package F.

This kind of compatibility conflicts is — in terms of Normalized Systems — caused by a violation of the first theorem, separation of concerns. Every external technology should be separated by a connector element. Note that a connector element is an Action Entity dedicated to connect a module with an external technology. To prevent version conflicts, every connector element should exhibit version transparency, like every other Action Entity should.

Let us consider the situation in which highly qualified software developers indeed implemented connector elements for every package or library they used. This makes the integrated system robust against anticipated changes, although it is obviously not delivering any garantee regarding the proper functioning of that external technology itself. In other words, if one or more packages or libraries are not properly linked to the development environment, the connector elements will deliver (on an asynchronous way) an error status.

Safe black box (re)use includes that a developer should be able to anticipate which conditions are necessary for (re)use. A self-explaining interface is a good start, but typically dependencies like packages or libraries are not included in the interface. We conclude that it should, and phrase the following rule.

In order to design safe black box (re)useable software components, every (re)use of a library or package in a module, should include a reference, path or link to the identification of the dependency, accompanied with the used version.

We make the reflection that there is a similarity between global variables and dependencies, which are not passed through the module's interface. Consequently, these dependencies cause common coupling. Remember it is not the *existence* of global variables, libraries or packages which is causing common coupling, but rather the fact that these variables, libraries or packages are *not passed via the module's interface*. As a result, these kind of dependencies violate the separation of concerns principle.

In searching ways to identify dependencies through the module's interface, the concept of OPC UA Profiles is interesting [17]. OPC UA Profiles define the functionality of an OPC UA application. As human-readable announcements, they inform users which parts of the OPC UA standard are implemented. In addition, this information can also be exchanged between OPC UA applications. This allows applications to accept or reject connection requests depending on which Profiles their counterpart is supporting. The concept of OPC UA Profiles is dedicated to exchange information about the interface and communication concepts of OPC UA applications. The functionality behind such an OPC UA interface is not exchangeable via OPC UA Profiles. In other words, the functionality, which can be expressed in standardized OPC UA Profiles, is limited to interface and communication functionality. This principle should be extended to a more general form to provide information about dependencies on different levels. For modules, directly connected to the internet, a worldwide accessible website could provide standardized information about well defined dependencies. For other modules, the same concept of reference could be introduced for specific application domains, or even vendor-dependent dependency information.

Remember the case of hardware engineers willing to share ICs for the development of embedded systems. The code printed on the ICs is referring to data sheets. This situation is similar to software modules, accompanied with a reference to dependency information in their interface. Whenever a user can not find the dependency information through a reference in the interface of a black box, it should be possible to reject the possible use of this module. This would result in an enforcement of the separation of states principle.

V. Conclusion

The reasons why properties like evolvability, (re)usability and safe black box design are difficult to achieve, have most likely something to do with a lack of making the existing knowledge and experience-based guidelines explicit. Undoubtedly, the theorems of Normalized Systems contribute on this issue by formulating unambiguous design rules at the elementary level of software primitives. However, on a higher implementation level, it is expected that not all implementation questions like those related to e.g., a dependency-hell, are easy to answer. Experienced engineers will find that these are violations of the theorems 'separation of concerns' and 'separation of states'. However, we aim that — on top of these fundamental principles — some derived rules can make these violations easier to catch, also for less experienced engineers.

In this paper we introduced the derived rule that, based on the 1st and 4th principle of Normalized Systems, any dependency should be visable in the module's interface, accompanied by its state and version. Of course, the way how this information is included in the interface, should be done in a version transparent way, to prevent violations of the 2nd and 3rd principle of Normalized Systems.

Acknowledgment

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

References

[1] Mannaert H. and Verelst J., "Normalized Systems Re-creating Information Technology Based on Laws for Software Evolvability", Koppa, 2009.

[2] Java platform enterprise edition. http://java.sun.com/javaee/.

[3] International Electrotechnical Commission, “IEC 61131-3, Programmable controllers - part 3: Programming languages”, 2003.

[4] Kuhl I. and Fay A., “A Middleware for Software Evolution of Automation Software”, IEEE Conference on Emerging Technologies and Factory Automation, 2011.

[5] McIlroy M.D., “Mass produced software components”, NATO Conference on Software Engineering, Scientific Affairs Division, 1968.

[6] Mahnke W., Leitner S., and Damm M., “OPC Unified Architecture”, Springer, 2009.

[7] Mannaert H., Verelst J., and Ven K., “Exploring the Concept of Systems Theoretic Stability as a Starting Point for a Unified Theory on Software Engineering”, IARIA ICSEA 2008.

[8] Lehman M.M., “Programs, life cycles, and laws of software evolution”, Proceedings of the IEEE, Vol 68, pp. 1060-1076, 1980.

[9] Mannaert H., Verelst J., and Ven K., “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability”, Science of Computer Programming, 2010.

[10] Mannaert H., Verelst J., and Ven K., “Towards evolvable software architectures based on systems theoretic stability”, Software, Practice and Experience, vol. 41, 2011.

[11] van der Linden D., Mannaert H., and de Laet J., “Towards evolvable Control Modules in an industrial production process”, $6th$ International Conference on Internet and Web Applications and Services, pp. 112-117, 2011.

[12] van der Linden D., Mannaert H., Kastner W., Vanderputten V., Peremans H. and Verelst J.,“An OPC UA Interface for an Evolvable ISA88 Control Module”, IEEE Conference on Emerging Technologies and Factory Automation, 2011.

[13] Myers G., “Reliable Software through Composite Design”, Van Nostrand Reinhold Company, 1975.

[14] van Nuffel D., Mannaert H., de Backer C., and Verelst J., “Towards a deterministic business process modelling method based on normalized theory”, International journal on advances in software, 3:1/2, pp. 54-69, 2010.

[15] Dijkstra E., “Go to statement considered harmful”, Communications of the ACM 11(3), pp. 147-148, 1968.

[16] de Sousa M., “Proposed corrections to the IEC 61131-3 standard”, Computer Standards & Interfaces, pp. 312-320, 2010.

[17] OPC Foundation. “OPC Unified Architecture, Part7: Profiles”, Version 1.01, Draft 1, september 2010.