The Role of Concept Management in System Development

– A practical and a theoretical perspective

A.I. Bleeker[1], H.A. Proper[2] and S.J.B.A. Hoppenbrouwers[2]

[1] Luminis, IJsselburcht 3, 6825 BS Arnhem, The Netherlands, EU araminte.bleeker@luminis.nl
[2] University of Nijmegen*, Sub-faculty of Informatics, IRIS Group, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, EU e.proper@acm.org, stijnh@cs.kun.nl

Published as:

A.I. Bleeker, H.A. Proper, and S.J.B.A. Hoppenbrouwers. The role of concept management in system development – a practical and a theoretical perspective. In J. Gravis, A. Persson, and J. Stirna, editors, *Forum proceedings of the 16th Conference on Advanced Information Systems 2004 (CAiSE 2004)*, pages 73–82, Riga, Latvia, EU, June 2004. Faculty of Computer Science and Information Technology.

Abstract. In this article we argue the need for proper concept management during the development of software systems. It is observed how, during system development, a lot of "concept handling" occurs without proper management. We define concept management as the deliberate activity of introducing, evolving and retiring concepts. It is argued that concept management plays an important role during the entire system development life cycle.
The notion of concept management is discussed and elaborated from both a theoretical perspective and a practical perspective. The latter perspective considers concept management in the context of the software development practice of a Dutch IT consultancy firm.

1 Introduction

Software systems are developed in order to support the activities that occur in some (class of) business domain(s)[3]. As a direct consequence, concepts from the business domain(s) are bound to play an important role in the deliverables that are produced in the course of system development, such as requirements and design documents, the constructed system, as well as the manuals for using the system. When, for instance, developing a software system to assist in the handling of claims in the context of a health insurance company, concepts such as "claim", "treatment", "processing of claims", "policy", etc., are bound to play a crucial role. During system development, requirements on the software system are likely to be expressed in terms of these concepts, while the design of the system is bound to comprise a class or entity type "claim" and "policy" and some activity/process "claim processing". Needless to say that these concepts will even be reflected in the (user) manuals of the system.

* This paper results from the ArchiMate project (http://archimate.telin.nl), a research consortium that aims to provide concepts and techniques to support enterprise architects in the visualisation, communication and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Instituut, Centrum voor Wiskunde en Informatica, Katholieke Universiteit Nijmegen, and the Leiden Institute of Advanced Computer Science.

[3] In [1] a distinction is made between *usage world*, *subject world*, *system world* and *development world*, when discoursing about information system development. What we refer to as business domain is corresponds to the subject world from [1].

The concepts in the business domain are not the only concepts that play a role during system development. The software system will be implemented using several forms of technologies and pre-existing infrastructures. This gives rise to an additional class of concepts: the concepts from the implementation domain. These concepts deal with the mapping of the concepts from the business domain to the technological infrastructure underlying the software system. Examples of such concepts would be: "claim queue handler", "claim scheduler", etc. Some of the concepts in the implementation domain are likely to be application dependent while others will be of a more infrastructural/generic nature. In this article we mainly focus on concepts that are native to the *business domain*.

In sum, one could state that during system development, a lot of "concept handling" occurs. At times we may engage in it without explicitly realizing we do so. Concept handling may occur when analyzing the structure and nature of the business domain, when formulating the software needs, or during the design and realization of the implementation of the system and its documentation. Business domain concepts are introduced, evolved and retired for different reasons. Initially they are introduced with the aim of scoping and understanding the business domain for which the software system is to be built. During requirements engineering as well as the design and realization of the system, additional insights may be gained into the structure and nature of the business domain. These insights are bound to lead to the evolution of the concepts used thus far.

The fields of conceptual modelling, requiremens specification, ontology engineering, etc, all contribute towards the "handling of concepts" during system development. It is our believe, however, that one should not just *handle* concepts, but rather consciously *manage* them. We regard the proper management of concepts during system development as an essential cornerstone for the development of systems that indeed fit the needs of the business domain. With the notion of concept management we refer to: *the deliberate activity of introducing, evolving and retiring concepts*, where *deliberate* hints at the goal-driven nature of the management of the concepts. Concept management plays a role during all phases of a system's life cycle:

Requirements engineering Requirements on the software system are formulated in terms of the business domain concepts. For example, the use cases and activity diagrams from the UML [2], may be used to define the functional requirements of a system, and can be used to more precisely specify any of the non-functional requirements. These diagramming techniques all refer, using labels, to concepts from the business domain.
It is argued by several authors [3, 4] that requirements should be managed well during system development. However, if the definitions of the underlying domain concepts are not managed equally well, the meaning of the concepts may change unnoticed. Changes in the meaning of the concepts will automatically lead to changes in the precise meaning of the requirements as well. Even more, the additional insight into the concepts of the business domain may lead to further refinement of the requirements on the future software system.
Requirements management, however important, should be able to rely on proper management of the underlying business domain concepts.

Design & realization During design and realization, the concepts of the business domain are mapped to specific features of the future system in terms of the high-level and low-level designs of the system. In doing so, the proper usage of the original definitions of the concepts should be guarded, otherwise there is likely to be a mismatch between the functionality as it was intended and the functionality as it is actually provided by the system.
At the same time, we should allow for additional insights into the business domain as it may be gained during the design and realization. As a result, we should allow the set of business domain concepts to evolve; the set of concepts is never really finished. This evolution should, however, be managed. Note that during design and realization, it is likely that new concepts for the implementation domain are introduced as well.

Operational use & maintenance The user documentation, as well as maintenance documentation, of the system will be formulated in terms of the business domain concepts as well. The use of these concepts should honor their original definitions.

Evolution of concepts is just one of the complexities facing concept management. Another complexity is the fact that in practice a software system does not simply have to deal with a homogenous and unified business domain. It rather has to deal with different contexts and groups of users; for example a claim handling system needs to deal with the financial administration, claim assessment, as well as the clients who hand in their claims. This heterogeneity is bound to lead to different sets of business domain concepts, and will even lead to a myriad of homonyms and synonyms. The heterogeneity of the business domain, and the impact it has on the business domain concepts, should not be denied; it should be managed instead.

We are not alone in arguing in favor of proper concept management. The Rational Unified Process [5] (RUP) explicitly underlines its importance as well. The RUP identifies the notion of a *glossary*, which is used to define terminology specific to the business domain, explaining terms which may be unfamiliar to the reader of use-case descriptions or other project documents. Quite often, this document is used as an informal data dictionary, capturing the definitions of the terms. At the same time, we believe that the notion of a glossary, in particular when viewed as a "list of terms", is a too light weight mechanism for proper concept management. In this article we will propose a more rigorous approach, based on pre-existing techniques for the modeling of domains.

Development of software systems is actually not the only engineering discipline where concept management is called for. The aeronautical industry has long since identified the need for proper concept management. For example, Boeing [6] uses a strict form of concept management, based on the restricted language as defined by the European Association of Aerospace Industries [7], to safeguard that concepts are used in the same way throughout the documentation of one of its airplane types.

The structure of the remainder of this article is as follows. Given the above discussed motivation for concept management section 2 aims to provide an overview, from a *theoretical perspective*, of the activities involved in concept management. Using these activities as a framework of reference, section 3 aims to offer a *practical perspective* on concept management by discussing how a Dutch software development firm uses elements from concept management in their software development practices.

2 Activities in concept management

As mentioned before, concept management is not limited to the development of software systems alone. In the context of system development, concept management can aid in achieving the following goals:

1. articulate clear and concise meanings of business domain concepts,
2. achieve a shared understanding of the concepts among relevant stakeholders and
3. guard the stability of a concept's meaning during system development.

Based on the results reported in [8], we consider concept management in the context of system development to revolve around four streams of (mutually influencing) activities:

1. Scoping environments of discourse.
2. Concept specification.
3. Concept integration.
4. Concept enforcement.

These streams are explained in more detail below. Note that these are four *streams* of activities. It is not likely that they can be executed in a linear order.

2.1 Scoping environments of discourse

Traditionally, when developing a model of the relevant concepts from an business domain, the term *universe of discourse* is used to refer to "*the world (or universe) we are interested in talking (or discoursing) about*" [9, 10, 8]. It is a "conceptual world": a world about which people communicate, and which may be described through the naming of entities and relations. This view implies that if one makes a model of the domain, one also (almost implicitly) makes a model of the language used to describe that domain.

In [8], the theoretical concept of *environment of discourse* was introduced, to extend the notion of *universe of discourse* [9][4]. The *environment of discourse* is essentially defined as a group of people who discourse in, or about, part of a universe of discourse. In other words, it links a group of individuals to a universe of discourse, and in doing so allows for the recognition of the fact that it is this combination of language users to which a language may be associated. In addition, recognizing environments of discourse allows for a diferentiated look at the conceptual needs of various groups within one universe of discourse [11, 8].

In line with the earlier mentioned heterogeneity of business domains, the development of a software system typically touches multiple discourse environments. For example, the development of a claim handling system would probably have at least four key environments of discourse: the *project members* of the system being developed, the *financial administration*, the *claim assessment department*, and the *clients*.

A key activity in concept management is the identification, and proper scoping, of the environments of discourse that are most important to the success/failure of the system to be developed. In summary, we have:

- A software system has *one* envisaged (class of) business domain(s) for which it is intended.
- Multiple environments of discourse can be associated to this (class of) business domain(s).
- Each environment of discourse embeds its own universe of discourse.

2.2 Concept specification

For each of the identified environments of discourse, the relevant business domain concepts should be specified in terms of their:

- meaning,
- relationships to other concepts (and the constraints governing these relationships and
- possible names used to refer to them.

The process of arriving at concepts, together with their meaning and names, is referred to as a *conceptualization process* [8]. When, as in the context of software development, the conceptualization is done with an explicit goal in mind, it is referred to as an *explicit conceptualization process*. In [8] a reference model for explicit conceptualization processes is provided. This reference model distinguishes five key steps:

1. *Acquire raw material.*
   This step aims to bring together input documents of all sorts that provide a basic understanding of the universe of discourse that is relevant to the environment of discourse under consideration.

[4] The term *environment of discourse* was first introduced in [11]. However, we use it in a somewhat different sense.

2. *Capture the universe of discourse.*
   In this step, formal decisions are to be made regarding the concepts that play a role in the universe of discourse and how these concepts interrelate. Note that explicit conceptualization can occur in two extreme forms: analyzing a pre-existing universe of discourse or designing a future (as yet non-existent) universe of discourse.
3. *Select relevant concepts.*
   The goal of this step is to focus on those concepts in the universe of discourse that bare some relevance to the system to be developed. These are the concepts that should be defined and named formally in the next step.
4. *Name and define concepts.*
   All of the concepts selected in the previous step, should be named and defined. Homonyms and synonyms within one environment of discourse should be allowed. However, homonyms within one environment of discourse should be discouraged if it concerns concepts that are to be used to formally describe system requirements or parts of the system's design.
5. *Quality checks.*
   Final quality checks on the validity, consistency and completeness of the set of defined concepts.

It should be noted that during the initial stages of system development, one may chose not to provide a precise meaning for relevant concepts (yet). For example, using a metaphorical meaning of concepts pertaining to a future business/system, may actually stimulate innovative thinking. Focussing on a precise definition to early may stifle innovation.

A conceptualization process is not a desk activity; far from it. It is important that all stakeholders of an environment of discourse participate in some form in order to arrive at a commonly accepted and understood set of concepts. All stakeholders should be actively involved. Moreover, stakeholders should be well aware of the impact of defining concepts that are to be implemented in the software system. Once implemented, they will be difficult to change (in [8], this is referred to as *freezing language*).

In conceptualization processes, several stakeholders play a role. It is important to be aware of the roles played by five specific classes of stakeholders [8]:

- Concept informants: those who take part in discussions about language/terminology.
- Concept authors: those who are authors of certain language specifying artifacts.
- Concept authority: those who decide upon official terminology and meaning of concepts.
- Concept managers: those who are responsible for concept management (including conceptualization processes).
- Conceptualization facilitators: those who are a facilitators for the conceptualization process.

2.3 Concept integration

Analyzing the concepts that play a role in a business domain by "simply" specifying the concepts for each of the environments of discourse is not enough when developing a software system. The concepts as identified in the different environments of discourse may quite well contradict each other. There are likely to exist homonyms and synonyms between different environments of discourse.

Since the software system that is being developed needs to interface with these differing environments of discourse, some integration between these environments should occur. On one extreme we could require the development project, as well as the resulting software system, to be able to use the concepts as used natively in the different environment of discourse. This would require the development project, as well as the software system, to be able to translate between the different "languages" spoken in the different environments! On the other extreme, we could require a harmonization of all relevant concepts across all involved environments of discourse. In practice a balance needs to be struck [8].

When harmonizing concepts between environments of discourse, one may also need to backtrack on steps 2–5 of the concept specification stream.

2.4 Concept enforcement

Finally, the definitions of the concepts should be enforced as specified. The meaning of the concepts should be the same for the requirements, the design of the system, as well as in the documentation of the system.

As mentioned before, there may be reasons to change the definition of concepts during the development of a system. For example, during the design of a system, additional insight may be gained into the structure of the business domain. However, when changing the definition of a concept, one should realize the consequences, as it is bound to lead to widespread changes. For instance, due to the changed meaning of a requirement expressed in terms of this concept. Furthermore, care should be taken in obtaining agreement with the concept authorities when modifying the definition.

3 Concept management in practice

In the previous sections a theoretical framework for concept management has been introduced. This section discusses a practical viewpoint on this matter. This discussion is illustrated by the practical experiences of Luminis, a Dutch IT consultancy firm. Luminis provides IT products and services to organizations producing products with a high complexity. All of Luminis' products and services are aimed at staging innovation in terms of technology adoption and improvement of both skills and processes.

Although concept management is not necessarily restricted to the development of purely computerized systems, the discussion in this section does focus on modeling for computerized systems as it is the primary business Luminis operates in. Furthermore, the discussion focuses on concrete systems for specific business domains, leaving the additional complexity of product families out of consideration.

3.1 Domain modeling

Software developers tend to be good at thinking in terms of technology enabled solutions and focused on sound engineering. Software projects often fail because it is unclear what the underlying business need is and what problem is actually being solved. Concept management is regarded as playing an important role in preventing this.

Luminis' approach to requirements engineering and software development is inspired by the RUP [5], while concept management has been adopted as an integral part of the development process. Luminis adopted the term *domain modeling* as a generic term to refer to the first three steps of concept management: scoping environments of discourse, concept specification and concept integration. The result of this process is referred to as the *domain model*.

The purpose of a domain model is to uniformly define and scope the business domain in terms which the stakeholders understand and agree upon. Towards the stakeholders in the business environment, the domain model plays the role of a unified vocabulary and an understanding of the scope of the system; towards software architects and engineers it provides guidance in making implementation decisions (for example database design). For the project leader it can be an aid in planning and prioritization of the project.

3.2 Way of working

In order to develop *successful* software systems, that is, software systems that meet the business needs, the following aspects need to be considered:

– The intrinsic structure and boundaries of the business domain in terms of its composing environments of discourse.
– The use of the concepts from the business domain by the stakeholders in the context of the planned system.
– The non-functional qualities of the planned system.

The structure of the business domain is captured in the aforementioned domain model, which is a concrete project deliverable. This model defines the relevant concepts in the business domain and the relationships between them. The use of the business domain concepts, as it is to be supported by the planned system, is laid down in the functional requirement specification (e.g. in terms of use cases from the UML). Finally, it is defined, in terms of the non-functional qualities, *how well* the system is supposed to do what it does. Concepts play a role in all these facets; concepts are the vocabulary.

Running example To illustrate the way of working the following running example is used. One of the elements in the Dutch healthcare system is the notion of *domestic aid*. People in need of medical attention can request a domestic aid worker to regularly visit their homes to assist with medical care or house work. Domestic aid organizations receive payment from health insurance companies for the hours of aid provided to their clients. Consequently, it is very important that domestic aid workers maintain an administration of the hours spent at their clients. To this end, the Dutch company Nedap (www.nedap.nl) developed a dedicated device called iO. The underlying idea is quite simple: Clients receive a personal badge and each domestic aid worker carries an iO. When the domestic aid worker arrives at the client's home she holds the iO against the client's badge and presses a button. The iO has now registered the starting time and the identity of the client. When the worker leaves she does the same and the iO registers the completion time. In the office the workers "empty" their iOs into a central computer which takes care of invoicing.

Scoping environments of discourse When a new project starts off, the first step is scoping. In terms of RUP [5] this takes place in the inception phase. During scoping the area of interest is explored on the basis of the product vision (see RUP) and a first rough outline of the (non)functional requirements of the system is made. This step also leads to a rough outline of the business domain. Scoping is a heterogeneous activity, it involves analysis and high level design but it is often also a negotiation process between different stakeholder interests, such as content versus costs.

Domain modeling plays an important role during scoping. There is an interesting correlation between modeling the domain and specifying the requirements on one hand, and the intended scope of the system as defined in the vision on the other hand. On one hand the scope of what is modeled specifies the boundaries of the system; on the other hand the scope of the system determines what is modeled. The domain modeling activity ties both ends together.

In the domestic aid example the following environments of discourse were identified: domestic aid organization, domestic aid worker, Nedap product development and technical support. Each has a different interest in the product.

Concept specification Once the scope of the project has been outlined, it is clear what the relevant environments of discourse are. The next step is to start identifying and specifying the relevant concepts in the environments of discourse. In RUP terms this starts during the inception phase and carries on in the elaboration phase. Concept specification is done in parallel to and in close interaction with requirements elicitation and specification. The relation to requirements specification is important: Requirements are expressed in terms of concepts and on the basis of their understanding of concepts stakeholders make assumptions on what the system will do. This is independent of the technique used for requirements specification; this holds for any verbal

requirements specification, for example use cases. During concept specification, modelers should be extra cautious about concepts that the domain experts consider to be "trivial". If during requirements specification concepts surface that have no place yet in the domain model one should start wondering whether it concerns a new concept or a synonym of an already identified concept.

RUP employs a *glossary* to define concepts from the business domain. Such a glossary does not specify the relationships between concepts nor the constraints on the relationships. To remedy this, Luminis has chosen to use Object Role Modeling (ORM) as a method for concept specification. ORM [10], and its many variations such as NIAM, PSM and NORM, has a rich theoretical foundation dating back to the 1980's and 1990's [12, 13, 14, 15, 16, 17, 18, 19, 20]. Even though the UML [2] is used intensively during the development of software systems, ORM still has a important role to play in the early stages of system development. An active community of ORM users exists (see e.g. www.orm.net and www.inconcept.com), while Microsoft's Visio Modeler even provides advanced support for ORM diagrams [21]. One of the important features of ORM is its foundation in natural language analysis. ORM views the world in terms of objects playing roles, and approaches a domain by verbalizing examples of such objects in natural language. These verbalizations are used as a starting point to further develop and refine the model. ORM also does not make an initial use of the notion of attribute; hence the modeler is not required to agonize over whether some feature ought to be modeled in terms of attributes rather than an entity and a relationship.

Another important feature of ORM is its elaborated modeling procedure which guide modelers in their task. In [10] an elaborate discussion of this procedure can be found. However, the ORM modeling procedure as such is too rich for the purposes of domain modeling, as it is originally geared towards conceptual *design* of a database system rather than the *analysis* of concepts playing a role in a business domain. Nevertheless, the general structure of the procedure can still be applied. The trick is to be less rigid about some of the modeling details, such as the specification of domains for value types, identification mechanisms for objects, advanced constraints.

Furthermore, even though verbalization of familiar examples from the business domain is a very powerful mechanism and the basis of proper conceptual modeling, it also involves a lot of documenting. In projects the more experienced conceptual modeler will generally not explicitly document the verbalizations but rather produce ORM or UML diagrams directly.

One might wonder why the use of ORM diagrams is preferred over the use of UML class diagrams to express domain models. Even though UML class diagrams can indeed be used for this purpose (ignoring the forced identification of attributes), UML diagrams have an implementation oriented co-notation from the perspective of the stakeholders in the business domain. This co-notation disqualifies UML class diagrams for the purpose of domain modeling as domain modeling requires intensive communication with the stakeholders about the structures of their domain.

Finally, as with all methods, ORM's modeling procedure should be used wisely. Given the characteristics of a specific project and business domain, it may be necessary to iteratively evolve a model, using short cycles, into a "good enough" model. This will be dictated by the situation at hand. In other words: *A method should never become an excuse to stop thinking.*

Concept integration An important stream of activities in the creation of a domain model is concept integration. As mentioned before, the four identified *streams* of activities in concept management are not likely to be executed in a linear order. In practice, this holds particularly for the integration of concepts between different environments of discourse. During modeling, the integration takes place while going along due to the iterative nature of modeling; i.e. it is hard to single out specific integration activities.

An important aspect of this stream is the continual (and driving!) validation whether the resulting domain model spans all environments of discourse (identified thus far). For each synonym and homonym it must be decided which term is used as the "official" one in the model.

Practical experience shows that the specification of concepts does not occur on a per environment-of-discourse basis, but rather on the basis of logical groupings of concepts, which are considered from the perspective of each environment of discourse they pertain to.

Documenting a domain model There are different ways to document the various stages of the domain model. The scope of the business domain may be documented as a concept map [22] or "information landscape" in the product vision document. The domain model itself may be documented in:

- Natural language.
- A graphic language, such as UML or ORM diagrams.
- A mathematical language, such as logic or set theory.

While ORM is used as a conceptualization method, Luminis uses UML class diagrams when communicating the domain model to software engineers. More specifically, the software engineers receive a domain model as a UML class diagram, in combination with natural language descriptions and a glossary. Natural language is used to communicate modeling decisions to the domain experts. The UML is used to add the details and specify the domain model in a format that architects and engineers can deal with in object-oriented technology. Either way, it is important to be concise and to the point. Too many projects suffer an overload of too many and too bulky documents.

Architecting the software system The domain model is one of the foundations for architecting and engineering as it determines the scope of the system, which is eventually reflected in the software design and functionality. Throughout the engineering process the model is used to manage concepts and as such monitor whether the right software is being built.

Concept specification and integration yields a conceptual model of the business domain. Sometimes there may be modeling choices present in the domain model that may be hard to realize in a specific programming environment or target platform. For such reasons, it is practical to translate the domain model into an *implementation model* (a process quite similar to the one as described in [23]). The implementation model can be seen as a more pragmatic view on the domain. However, it does *not* change the actual definition of the domain. The implementation model is consistent with the original domain model. For example, complex relationships can be translated into multiple binary relationships.

When transforming the domain model into an implementation model, one will usually cross-over from ORM diagrams to UML diagrams, as this is the lingua-franca of software developers. This is also the stage where other UML diagrams, such as sequence diagrams, collaboration diagrams and activity diagrams come into play. All of these diagrams play a role in the design of the software system that implements the business domain.

3.3 Reflections

ORM, just as any other modeling methodology, explains *how* to model but not *what* to model and this precisely is the challenge in modeling. Making the right decisions about what to model, what terms to use, when to stop, who to involve and how to document and communicate it is mostly a matter of talent and experience of the modeler(s).

An important issue, in this respect, is whether the *current* situation is modeled or some desired *future* situation. If the current situation is being modeled the resulting model may not yield any new insights to the domain experts and the more skeptical ones might say the model is "trivial" or not really adding anything new. In a way this is actually a good sign because the model

apparently succeeded in specifying the relevant concepts in a recognizable way. A model of the current situation structures domain knowledge and unifies terms but is not intended to teach the domain experts new insights on their specialism. Modeling the future situation, however, puts other demands on the domain experts, as they are now asked to think about how they believe their business is going to develop and what is important in future (software) systems.

Another issue is which term to use in case of synonyms. This may be decided by the stakeholders with authority but reuse may also play a role. For example, Nedap also develops other device oriented products (for example access control solutions) and there is a need to reuse certain parts of the software. So it is important to choose more generic terms, rather than domestic aid specific terms. Another way to deal with synonyms is localization in the user interface: The model remains an internal deliverable while the stakeholders of different environments of discourse find their local terminology in a personalized user interface.

Finally, in almost every business there is some form of evolution. The nature and frequency of evolution depends on the type of business. It is important to decide how heavily concept management is impacted by evolution. Should the system and therefore the domain model anticipate for it right from the beginning or is it a matter of "*time will tell*"?

The domestic aid device shows an example of evolution. The original iO has been developed to register hours spent at client homes, so called *external* hours, and so has the software. However, during requirements specification the concept of *internal hours* surfaced. Internal hours are hours spent around the office, for example, for meetings or education, but also holidays and sick leave. The vision behind the iO and client badge did not anticipate for this extension. Even more, the intruduction of this concept required changes to both software and hardware.

4 Conclusions and future research

In this article we have argued the need for proper concept management in the context of system development. We have discussed how, during system development, a lot of "concept handling" occurs and that this quite often occurs without proper management. We have introduced the explicit notion of concept management as the deliberate activity of introducing, evolving and retiring concepts. The notion of concept management has been discussed and elaborated from both a theoretical perspective as well as a practical perspective.

In the near future, we are planning on a more explicit integration between concept management, the ORM conceptualization procedure and software development using the RUP and the UML. We will also aim to formulating explicit heuristics on dealing with the integration of concepts between different environments of discourse. When possible, a prototype for a concept management tool will be considered as well.

References

1. Mylopoulos, J.: Techniques and languages for the description of information systems. In Bernus, P., Mertins, K., Schmidt, G., eds.: Handbook on Architectures of Information Systems. International Handbooks on Information Systems. Springer, Berlin, Germany, EU (1998). ISBN 3-540-64453-9
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modelling Language User Guide. Addison-Wesley, Reading, Massachusetts, USA (1999). ISBN 0-201-57168-4
3. Robertson, S., Robertson, J.: Mastering the Requirements Process. Addison-Wesley, Reading, Massachusetts, USA (1999). ISBN 0201360462
4. Leffingwell, D., Widrig, D.: Managing Software Requirements: A Use Case Approach. 2nd edn. Addison-Wesley, Reading, Massachusetts, USA (2003). ISBN 032112247X
5. Kruchten, P.: The Rational Unified Process: An Introduction. 2nd edn. Addison-Wesley, Reading, Massachusetts, USA (2000). ISBN 0201707101

6. Farrington, G.: An Overview of the International Aerospace Language. (1996).
7. AECMA – The European Association of Aerospace Industries: AECMA Simplified English – A guide for the preparation of aircraft maintenance documentation in the international maintenance language, Brussels, Belgium. (2001) Issue 1, Revision 2.
`http://www.aecma.org`
8. Hoppenbrouwers, S.: Freezing Language; Conceptualisation processes in ICT supported organisations. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, EU (2003). ISBN 9090173188
9. ISO: Information processing systems – Concepts and Terminology for the Conceptual Schema and the Information Base. (1987) ISO/TR 9007:1987.
`http://www.iso.org`
10. Halpin, T.: Information Modeling and Relational Databases, From Conceptual Analysis to Logical Design. Morgan Kaufman, San Mateo, California, USA (2001). ISBN 1-55860-672-6
11. Weigand, H.: Linguistically Motivated Principles of Knowledge Base Systems. Foris, Dordrecht, The Netherlands, EU (1990).
12. Verheijen, G., Bekkum, J.v.: NIAM: an Information Analysis Method. In Olle, T., Sol, H., Verrijn-Stuart, A., eds.: Information Systems Design Methodologies: A Comparative Review. North-Holland/IFIP WG8.1, Amsterdam, The Netherlands, EU (1982) 537–590.
13. Nijssen, G., Halpin, T.: Conceptual Schema and Relational Database Design: a fact oriented approach. Prentice-Hall, Sydney, Australia (1989). ASIN 0131672630
14. Wintraecken, J.: The NIAM Information Analysis Method: Theory and Practice. Kluwer, Deventer, The Netherlands, EU (1990).
15. Bommel, P.v., Hofstede, A.t., Weide, T.v.d.: Semantics and verification of object-role models. Information Systems **16** (1991) 471–495.
16. Halpin, T., Orlowska, M.: Fact-oriented modelling for data analysis. Journal of Information Systems **2** (1992) 97–119.
17. Hofstede, A.t., Weide, T.v.d.: Expressiveness in conceptual data modelling. Data & Knowledge Engineering **10** (1993) 65–100.
18. Shoval, P., Zohn, S.: Binary-Relationship integration methodology. Data & Knowledge Engineering **6** (1991) 225–250.
19. De Troyer, O.: The OO-Binary Relationship Model: A Truly Object Oriented Conceptual Model. In Andersen, R., Bubenko, J., Sølvberg, A., eds.: Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering. Volume 498 of Lecture Notes in Computer Science., Trondheim, Norway, Springer-Verlag (1991) 561–578.
20. Habrias, H.: Normalized Object Oriented Method. In: Encyclopedia of Microcomputers. Volume 12. Marcel Dekker, New York, New York, USA (1993) 271–285.
21. Halpin, T., Evans, K., Hallock, P., Maclean, B.: Database Modeling with Microsoft's Visio for Enterprise Architects. Morgan Kaufmann, San Mateo, California (2003). ISBN 1558609199
22. Novak, J.: Concept maps and Vee diagrams: Two metacognitive tools for science and mathematics education. Instructional Science (1990) 29–52.
23. Halpin, T., Proper, H.: Database schema transformation and optimization. In Papazoglou, M., ed.: Proceedings of the OOER'95, 14th International Object-Oriented and Entity-Relationship Modelling Conference. Volume 1021 of Lecture Notes in Computer Science., Gold Coast, Australia, Springer Verlag, Berlin, Germany, EU (1995) 191–203. ISBN 3540606726