Electronic Communications of the EASST
Volume X (2007)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

An Active Domain Meta Layer for Documented Software Malleability

Dirk Deridder, Sofie Goderis, Isabel Michiels and Viviane Jonckers

10 pages

Guest Editors: Tom Mens, Maja D'Hondt, Kim Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Active Domain Meta Layer for Documented Software Malleability

Dirk Deridder[1], Sofie Goderis[2], Isabel Michiels[2] and Viviane Jonckers[1]

[1] dderidde@vub.ac.be, http://ssel.vub.ac.be/
System and Software Engineering Lab
Vrije Universiteit Brussel, Belgium

[2] http://prog.vub.ac.be/
Programming Technology Lab
Vrije Universiteit Brussel, Belgium

Abstract: In order to cope with the ever-increasing adaptability and flexibility requirements for software we introduce the notion of a Concept-Centric Coding (C3) environment. The aim is to provide an active domain meta layer that can be used as a basic infrastructure to set up a documented malleable implementation. We illustrate this by means of CoBro, a proof-of-concept C3 environment developed in Smalltalk.

Keywords: Run-time Adaptation, Documentation, Agile Development, Co-Evolution

1 Introduction

In order to survive in today's highly dynamic marketplace, companies must show a continuous and ever-increasing ability to adapt. This level of agility, necessary to keep up with the business side of the adaptability challenge, consequently imposes itself on the software side. As a result the current generation of software developers needs to service extreme adaptability and flexibility requirements in an environment of extremely strict deadlines, short time-to-market expectations, and limited availability of project resources [CDBD04].

The kind of systems targeted by this research are E-type object-oriented applications written in a class-based programming language. Such E-type applications are known to be in a constant state of change [Leh96] and typically call for a malleable implementation to facilitate adaptations as swiftly as possible. Additionally such systems are closely connected to a real world domain, which means that there exists a tight two-way interaction between changes in the real world and changes to the application [Pfl98]. As a result of this close connection to the real world, a lot of domain expertise is involved in the development of E-type systems.

Since a lot of this domain expertise is lost over time or remains implicit in a software implementation, we present an approach that provides support to make it explicit in connection to the code. Moreover, to support a programmer in developing a malleable implementation, we incorporate a mechanism that renders the active use of this knowledge transparent to the programmer. Both elements will enable developers to cope with the adaptability challenge referred to above.

The approach is labelled *Concept-Centric Coding (C3)*[1] and is supported by a tool suite named the *Concept to Code Browser (*COBRO*)* [Der06].

In Section 2 we identify three main sources of problems that form a bottleneck for evolving object-oriented E-type applications. These are described from a worst case perspective and result in a considerable overhead for the developers.

In Section 3 we introduce the C3 environment to counter the problems identified. In essence we propose an environment that provides support to make domain knowledge explicit, to couple it to the code, and to use it in an active fashion. The domain knowledge thus forms an integral and essential part of the implementation and can be used as a basic building block for creating a malleable implementation. This results in an active domain meta layer where special care is taken to reduce the overhead perceived by developers to an absolute minimum. This is based upon a symbiotic integration[2] of the domain meta layer with the underlying programming language which lets programmers handle domain concepts and implementation objects transparently.

2 The Need for a Concept-Centric Coding Environment

2.1 Implicit Domain Knowledge

Up to 60 percent of the time spent in software evolution is dedicated to program comprehension [ABDT04]. Canfora et al. even mention estimates as high as 90 percent [CC05]. So before any adaptation actually takes place, a lot of effort is put in trying to understand the software. Therefore in program comprehension a major amount of time is dedicated to rediscovering and reconstructing the domain knowledge[3] that was available to the original developer. The fact that a lot of it remains implicit is clearly problematic for evolution.

Even though this is a rather plain observation, there is also empirical evidence that domain concept descriptions are amongst the most essential information needed by software maintainers [KSP04]. Koskinen et al. for example observed that *domain concept descriptions* and *connected domain-program-situation knowledge* are among the top three most frequent information types needed by software maintainers.

Broadly speaking, the main factors why knowledge about a software system becomes implicit can be summarised as follows [Der06]:

- **High personnel turnover**
  Software developers and domain experts are a volatile asset. They switch companies or project teams frequently. Moreover depending on the stage the software is in, it will become less likely to find people that are well-acquainted with the system.

- **Complexity of software systems**
  As a result of the complexity and the size of contemporary software systems, the peculiar-

[1] http://ssel.vub.ac.be/c3/

[2] With *symbiotic* we refer to a mutually beneficial relationship between the domain meta layer and the code layer. With *integration* we refer to the fact that both are closely combined so they form a whole

[3] The set of knowledge that captures the concepts and rules that govern a particular domain. This includes business domain knowledge which is rooted in the business (e.g., insurance, banking) and application domain knowledge which is rooted in the software used in a particular business.

ities of a particular component or implementation are often lost over time. This is not only a result of the difficulty people have to grasp the entire system. It is also a consequence of the large number of dependencies between the different elements.

- **Short-term software development economics**
  Spending time on documenting a system is time that cannot be spent on development. As a consequence code documentation is often neglected by developers in favor of producing code that contributes to the next release of the system.

- **Kind of knowledge**
  Even in a model-centric approach to software development not all knowledge is made explicit. For instance the motivation for a particular design choice is often left undocumented. Also common sense knowledge about the business domain in which the system operates is often unavailable in the code and the models. This is especially true in the case of a malleable generic implementation.

- **Genericity of the implementation**
  Continued development is also about refactoring the software to make it more susceptible to future changes. A basis for enhancing the flexibility of an implementation lies in writing generic code. Unfortunately generic code results in less explicit code since a lot of details become hidden in the data that is loaded at runtime. Moreover less explicit code is not helpful for code comprehension since it has a negative impact on readability [Fow01].

- **Reduced information quality**
  Each time information is passed on in a communication chain, there is a deterioration of the quality. Such a deterioration also occurs when an evolution request is passed on from a domain expert to a developer since the developer will have to map the business need onto the implementation. Moreover each evolution step results in certain parts of the implementation to be rewritten which also causes a recurrent loss of information.

We have briefly illustrated that domain knowledge becomes implicit for different reasons. This is problematic in the context of software evolution, since it is one of the main assets for program comprehension. Given this observation we conclude that there is a clear *need for an explicit medium to capture the relevant domain knowledge*. Next we discuss the importance of having a connection between the code and the domain knowledge.

2.2 Detached Domain Knowledge

Making the domain knowledge available in some explicit form would only be a partial solution to the problem. Developers also need to know to which part of the implementation the knowledge applies to. Traditionally however, when knowledge is made explicit about an implementation, it is typically done detached from the implementation. This means that there is no explicit link available between the knowledge and the code. The evidence by Koskinen et al. mentioned before also confirms that program-domain-situation knowledge is highly ranked by developers. This is not so surprising since developers are actually performing a continuous mapping of elements from the problem domain into the programming domain [MV95].

Mapping the domain knowledge back to the code is difficult in nature. This is in part because the domain knowledge cannot always be brought back to one specific location in the code. This manifests itself especially in what is known as distributed domains since the corresponding domain functionality typically crosscuts the entire application. As the mapping process between the knowledge and the code is difficult and time-consuming, it is important that a developer can attach his findings to the code once the (manual) mapping is complete.

To worsen the situation there also still exists a clear *dichotomy* of the traditional analysis-design-implementation phases. Broadly speaking this refers to the fact that the artifacts are decoupled each time control is passed from one phase to another. As a consequence, knowledge that is available in an explicit form at a certain point in time will become detached from the implementation in the end.

In this section we have seen that developers not only require explicit domain knowledge. They also need to know to which part of the implementation the knowledge applies to. This indicates *the need for a mechanism that helps developers to make the link between their domain knowledge and the implementation explicit*. Next we discuss why it is important that developers do not perceive domain knowledge as a passive asset and as an overhead to their development activities.

2.3 Passive Domain Knowledge and Overhead for the Developer

When domain knowledge is made explicit and when it is linked to the implementation, developers need to keep this information up-to-date. Yet this is not evident as a result of short-term software development economics. First of all this is due to the fact that developers do not perceive documenting as an activity that contributes to meeting their next deadline. As a consequence it is often neglected in favor of writing code. This is because the contribution of writing code to the next release of the system is clearly visible. You could say that the domain knowledge plays a *passive role* with respect to the execution of the system. We mean this in the sense that it does not participate actively to provide the functionality of the system in any way.

Secondly, writing down the domain knowledge associated with a system is mostly done outside a development environment. This is perceived as an overhead by developers since they need to *switch frequently between environments*. A seamless integration of tools in such case is beneficial for reducing the overhead introduced by the documentation system.

From this you could conclude that the domain knowledge documentation and the implementation live in a state of *antibiosis*. This means that an adverse relationship exists between both sides in which one is adversely affected by the other. In essence this boils down to the fact that spending time on the documentation results in less time available for the code level.

In the context of agile software development, the fact that domain knowledge plays a passive role poses an even bigger problem. Here, requirements are elicited and refined on the go during the implementation of the system. Hence the software grows incrementally and is under constant revision. As a consequence it is unfeasible to have a conventional documentation system. If domain knowledge is made explicit in such a context, it will always be out-of-date unless the developers spend a lot of time on updating it each time the code is changed. It is clear that unless the domain knowledge contributes to the implementation, that developers will not or cannot dedicate any time to it.

In this section we have seen that one of the main reasons why domain knowledge becomes implicit and detached is because it plays a *passive* role. As a result it is not kept up-to-date and its usefulness as documentation will quickly degrade. This indicates *the need for a mechanism that enables and promotes the activation of the domain knowledge. Moreover the mechanism should not be perceived as an overhead to the developer.*

2.4 Summary

In summary the problems discussed bring us to the following requirements that need to be addressed by the concept-centric coding environment we describe in the next section:

Implicit Domain Knowledge A C3 environment must provide a mechanism to capture domain knowledge in an explicit form. This will reduce the possible loss of domain knowledge.

Detached Domain Knowledge A C3 environment must provide a mechanism to couple the domain knowledge to the implementation. This will reduce the effort spent by developers on retroactively matching the knowledge to the corresponding implementation.

Passive Domain Knowledge and Overhead for the Developer A C3 environment must provide a way to involve the domain knowledge actively in order to provide the functionality of the software system. This empowers a developer to improve the malleability of the implementation by devoting time to the domain knowledge documentation. Moreover a C3 environment must be built in a way that it is not perceived by developers as an overhead.

3 CoBro: a Concept-Centric Coding Environment

In this section we couple the four key points related to the absence of domain knowledge to the different elements of our solution. In short, the C3 environment is based upon a symbiotic integration between an active domain meta layer and a programming environment. In the following subsections we discuss how the C3 solution takes us from a situation where domain knowledge is implicit detached and passive towards a situation where it is explicit coupled and active. Moreover special care is taken to make the use of the C3 solution by a developer as transparent as possible. Throughout the discussion we refer to CoBro, which is a proof-of-concept C3 tool suite developed in VisualWorks Smalltalk. For a detailed discussion of C3 and CoBro we refer to [Der06].

3.1 Setting up an Explicit Domain Meta Layer

Making implicit knowledge explicit requires some kind of formalism in which the developer can express this knowledge. For our application context we opted for an open and lightweight formalism. This is intended to minimise the overhead perceived by developers when using it. Moreover, the openness of the formalism is required to enable the developer to customise it according to a particular task at hand. As we will discuss in section 3.4, we prefer to use the same syntax and interaction mechanisms of the programming language (i.e. Smalltalk) to manipulate and access the domain meta layer. This is in line with minimising the difficulties encountered by users as described by Shipman and Marshall [SM99]. First of all they suggest to minimize the *cognitive overhead* perceived by the users (e.g. by using a syntax that is familiar to the user).

Secondly they stress that you should not enforce the user to commit to a *premature structure* (e.g. by not enforcing the use of class hierarchy if it is better to model in a different way). Finally they highlight the importance of enabling the user to *tailor the representation* according to the situation in which it is being used (e.g. by providing an open formalism that can be extended if necessary).

We developed a *frame-based representation* named CoBro-CML in which domain concepts are related to each other within a domain meta layer. This layer can be considered to act as an ontology for both the developers and the implementation since it provides an explicit shared representation of a domain conceptualisation. In CoBro a concept is defined by a *definition frame* that contains *definition entries*. We present an example below in CoBro-CML:

```
(Concepts new:  #{Adult})
   hasLabel:  'Adults age category';
   superconcept:  Concepts.AgeCategory;
   ageRange:  '#(25 64)';
   save.
```

In the example we illustrate the definition of the `Adult` concept which is an `AgeCategory` concept. Note that the syntax and interaction mechanisms of CoBro-CML are identical to Smalltalk for the purpose of transparency. The concepts from the domain meta layer are stored in a relational database, but this is not visible to the developer since they are first-class citizens.

The definition entries consist of two elements: a *relation* (e.g. `superconcept`) and a *destination* (e.g. `Concepts.AgeCategory`). Destination values can be another *concept* or a *terminal*. A terminal represents a value about which we do not wish to record extra information in the concept network other than its value (e.g. `#(25 64)`). Hence its definition frame does not contain definition entries but just the value that it represents. In essence the choice of representing an element as a terminal or a concept defines the *ontological granularity* of the domain meta layer. The types of terminals that are supported by CoBro are extensible. Hence it is possible to adapt the concept representation according to the particular needs of the developer (e.g. to relate a concept to a drawing). This is realised through the notion of `valueInterpreter`'s in the domain meta layer, which define how a particular terminal value should be handled. Note that the relations that are used in the definition entries are also defined as concepts. As a consequence the domain meta layer is said to be self-describing. So in the example, the `hasLabel`, `superconcept`, and `ageRange` relations are also defined as concepts. This results in a highly extensible environment in which developers can tailor the meta layer to their needs.

A *prototype-based approach* is followed for the domain meta layer. In a class-based approach the concepts would be represented as classes. Hence a concept could only exist unless there already existed a class that described the general characteristics of that type of concept. In contrast the prototype-based approach simply represents the actual concepts without this restriction. This is analogously to prototype-based programming languages and results in a more natural and highly dynamic way of interaction. For instance, you do not have to switch between class and object views to alter a concept definition. The choice for a prototype-based approach follows

naturally from our goal not to enforce users to a premature structure. A class-based approach on the contrary would force the user to commit to a taxonomy of concepts from the very beginning. As a consequence a mismatch is often created if the pre-defined interpretation of for example `is-a` is not in line with the user's intention [WF94].

Often a distinction is made between *ontology-aware*, and *ontology-driven* applications [Gua98]. This distinction is based upon whether the concepts from the ontology are used either passively or actively by the application. As we shall see in the following sections, we move towards an environment that is best characterised as ontology-driven.

3.2 Coupling the Domain Meta Layer to the Implementation

The coupling between domain concepts and code entities is done by representing the code entities (e.g. classes, methods) at the domain meta layer. The process of generating a concept representation for a code level entity is referred to as *code level entity conceptification*. In CoBro this is implemented through an extension of the Smalltalk language to support an up-down mechanism. For example you can 'up' a `GameRating` class to the domain meta layer by sending the message `asConcept` to it. This triggers the CoBro conceptification mechanism which will compute the concept representation for the meta layer. Note that the link between the code level entity and its domain concept counterpart remains explicit at all times.

The conceptification process requires access to information about the code level. Hence it requires support for reflection from the programming language. Smalltalk excells in its reflective capabilities which was to our advantage when building the CoBro kernel. Deciding what to represent at the domain meta layer is the first choice to make with respect to the granularity of conceptification. You can restrict, for example, the available slots at the meta layer to instance variables, methods, and a superclass. A second granularity decision is how to represent the information you compute for the code entity. This boils down to specifying whether the value of a computed slot refers to a terminal or a concept. You could for example represent the value for a superclass slot as a concept and the value for an instance variable slot as a terminal.

Since conceptification is an automated process, it follows a prescription based on the granularity that was chosen. This prescription defines what to represent and how to represent it, and is referred to as the *intension* of a concept. In CoBro this prescription can be easily customised since part of the CoBro implementation was realised by using the domain meta layer in an active fashion .

At the domain meta layer, there is no difference between code level concepts and domain level concepts. This implies that they can be related to each other by adding a relation between them. Thus you can annotate the class `GameRating` at the domain meta layer by relating it to domain concepts such as `Adult`, `Youngster`, or `Infant`.

3.3 Activating the Domain Meta Layer for Malleability

The way we activate the concepts in the meta layer is by making them accessible from within the code level. For this purpose CoBro extends the Smalltalk language kernel so that it becomes possible to write concept statements within normal code that consults, creates, updates or deletes domain concepts. We only illustrate this for consulting the domain meta layer by the following

code snippet of the `ageCategory` method from the `LibraryMember` class:

```
LibraryMember>>ageCategory
    ^ Concepts.AgeCategory allSubconcepts
      detect:
        [:each| (self age >= each ageRange first) &
              (self age <= each ageRange second)]
```

The different `AgeCategory` concepts are first retrieved from the domain meta layer, after which they are iterated over to find the one that matches the age of a particular library member. Another method can now make use of this information to verify whether the user is allowed to borrow an item:

```
MediaLibrary>>canBorrowItem:  aMember
    ^ aMember ageCategory is:  Concepts.Adult
```

This shows that domain concepts can indeed be involved in the computation at the code level. The resulting code is more generic and changes can be applied at the domain meta layer (e.g. adding age categories) without having to change the underlying code. Moreover instead of loosing the domain knowledge as a result of the genericity, we have actually embedded the domain knowledge explicitly within the implementation.

CoBRO also provides support to invoke conceptification by writing a statement at the code level (by sending the message `asConcept`). This gives a developer the means to 'up' a code level entity to the meta layer during the execution of a method. This enables a developer to write code that consults its own concept representation to perform a given task (e.g. to consult its annotations at the domain meta layer). A corresponding 'down' operation also exists (`asSmalltalk`), which returns the code level counterpart of a conceptified entity.

When an active meta layer is installed, a developer can begin to refactor the implementation so it becomes driven by the domain meta layer. As a consequence, the concepts play an active role in the functioning of the software system and certain adaptations can be realised by changing the concepts instead of the code. This results in developer support for run-time malleability which was already used for ensuring a flexible and extensible implementation of the CoBRO tool base.

Note that the activation of the meta layer is the enabler to turn the antibiosis between the concept and code level into a symbiosis. Spending time on the domain concepts thus becomes advantageous to the code level and vice versa. As a consequence, the domain knowledge documentation is no longer tied to a passive promise of benefits for future evolution efforts.

3.4 Transparency for the Developer

As already illustrated, a first step towards transparency is to use the same syntax to represent concepts as the syntax of the programming language. As a result, the developer is not required to learn another one. Moreover there is no need to switch between syntax each time a developer

works on either the concepts or the code. Hence the programming experience is not disturbed by the concept language, since syntax-wise there is no visible difference between concepts or code. This is the reason why we did not use an existing ontology language such as OWL, or RDF.

A second step to obtain transparency is to ensure that interacting with concepts and objects is done by using the same mechanisms. Hence, COBRO manipulates concepts by sending messages. This was already indicated in the code snippets, where the `ageRange` slot of the `Adult` concept was accessed by sending the corresponding message. Moreover, in order to be able to refer to concepts, a referencing scheme is needed that is similar to the one used by the programming language (e.g. `Concepts.Adult`).

The third step to achieve transparency lies in the close integration between the programming environment and the domain meta layer. This means that all the tools available for interacting with concepts are closely integrated with the existing programming tools. Consequently, there is no need for a developer to continually switch between environments. Moreover it is desirable to provide an open malleable implementation of the concept environment. This enables the developer to adapt the environment to fit more closely to the particular task at hand. As a consequence, the openness of the environment contributes to the fact that developers will no longer experience the domain knowledge documentation as an overhead.

4 Conclusion

We discussed COBRO, a concept-centric coding environment, which is based upon a symbiotic integration between a domain meta layer and a programming environment. The following requirements were detailed for our C3 environment:

Explicit Domain Meta Layer: COBRO provides a mechanism that makes it possible to capture domain knowledge in an explicit form. This reduces the possible loss of domain knowledge. The domain meta layer that captures the knowledge is represented in a frame-based, prototype-based ontology.

Coupling between the Domain Meta Layer and the Implementation: COBRO provides a mechanism to couple the domain knowledge to the implementation. This reduces the effort spent by developers on retroactively matching the knowledge to the corresponding implementation. Code level entities are deified towards the domain meta layer by a conceptification process that follows a prescription to generate the concept representation. This is referred to as the intension of the concept, which is based on the granularity chosen for what is represented at the domain meta layer and how it is represented.

Activation of the Domain Meta Layer: COBRO provides a way to involve the domain knowledge actively so as to provide the functionality of the software system. This motivates developers not to neglect the domain knowledge, since it potentially yields a short-term benefit. Moreover an enhanced malleability of the software can be achieved by devoting time on the domain meta layer. Concept statements can be written within normal code so that domain concepts and conceptified code entities can be involved in the computation at the code level.

Transparency and Reduced Overhead for the Developer: COBRO is built in a way that it is not perceived by developers as an overhead. This means that the interaction with the concept environment is set up as transparent as possible. The syntax of Smalltalk is used to represent con-

cepts at the code level. The mechanisms to interact with concepts are based on the mechanisms to interact with normal objects. The programming environment and the concept environment are implemented in close symbiotic integration with each other.

Acknowledgements: This research is funded by the IAP Programme of the Belgian State.

Bibliography

[ABDT04] A. Abran, P. Bourque, R. Dupuis, L. Tripp. Guide to the Software Engineering Body of Knowledge (Ironman Version). Technical report, IEEE Computer Society, 2004.

[CC05] G. Canfora, A. Cimitile. Software Maintenance. *IT Metrics and Productivity Journal*, November 2005. Online article - e-Zine.

[CDBD04] T. Cleenewerck, D. Deridder, J. Brichau, T. D'Hondt. On the evolution of IMedia Implementations. *Proceedings of the European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology*, 2004.

[Der06] D. Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Context*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, 2006.

[Fow01] M. Fowler. To Be Explicit. *IEEE Software*, November, December 2001.

[Gua98] N. Guarino. Formal Ontology and Information Systems. In *Proceedings of FOIS 1998, Trento, Italy*. IOS Press, Amsterdam, June 1998.

[KSP04] J. Koskinen, A. Salminen, J. Paakki. Hypertext Support for the Information Needs of Software Maintainers. *Journal of Software Maintenance and Evolution: Research and Practice* 16:187–215, 2004.

[Leh96] M. Lehman. Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*. Pp. 108–124. 1996.

[MV95] A. von Mayrhauser, A. Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, August 1995.

[Pfl98] S. L. Pfleeger. The Nature of System Change. *IEEE Software*, pp. 87–90, May/June 1998.

[SM99] F. M. Shipman III, C. C. Marshall. Formality Considered Harmful: Experiences, Emerging Themes, and Directions. *Computer-Supported Cooperative Work* 8(4):333–352, 1999.

[WF94] C. A. Welty, D. A. Ferrucci. What's in an Instance? Technical report, RPI Computer Science, 1994.