Maximizing Reuse with an Evolution Oriented Domain Engineering

Guillermo Mayobre

Hewlett Packard
Grenoble Networks Division
5, Ave Raymond Chanas
38053 Grenoble CEDEX 9
France
Email: gm@hpgntol1.grenoble.hp.com

Abstract

Software development on the context of domain of application (domain focussed software development) may be seen as evolutionary development were software representing core domain concepts are extended/adapted to meet new product requirements. On such a context, managing the evolution of existing software is key to keep development costs under control. An evolution oriented domain engineering including a domain analysis phase with special focus on the identification/prediction and characterization of the variability, provides a framework to successfully master the evolution of software.

This paper summarizes some of the results carried out on the context of the European Research project PROTEUS (ESPRIT project 6086), together with some practical results from the software reuse program at the Hewlett Packard Grenoble Networks Division.

Keywords: Domain, variant, invariant, impact analysis, evolutiveness, adaptive, cohesiveness with respect to variability, model predictability, model coverage, taxonomy of variability, specialization attribute, factors of variability.

1 Background

Domain oriented product development is today one of the prevalent ways of developing products. Corporations, companies, must of the organizations are almost organized around product lines or domains of applications. Simply because this is the most natural way of centralizing, capitalizing and maximizing the reuse of knowledge. Naturally the solution space corresponding to this domain oriented organization of the problem space is also domain oriented.

Applications of the domain represents specific solutions to the problem space. They are usually built on an incremental manner, evolving from the existing software. A part of their software represents core functionality of the domain, usually shared between several applications, and another part represents specific functionality.

What is important, is to abstract, from those specific solutions, generic specifications to derivate reusable, generic, highly evolutive, solutions to be able to implement new requirements/functionality within the domain at the lowest possible cost.

Up to now, most of the existing domain analysis approaches aim at preparing reuse by focussing on the identification of the common, invariant parts of the systems in the domain. The approach presented in this paper puts a strong emphasis on the identification/prediction and characterization of the variability, essential to master evolution. (Mechanisms dedicated to the identification of the invariant part are not described here, they are based on the common methodological framework that underlies current methods).

2 Position

2.1 Classifying Functionality

Functionality involved in the development of new systems may be classified in two categories.: Invariant and Variant. [1] Invariant functionality is the set of components or components fragments that are used without changes. It provides the kernel of functionality around which new systems are built. By definition invariant functionality cannot create new systems since it lacks the customization needed to meet new requirements. Variant functionality is the set new functionality that must be added/changed to customize invariant components. It provides novel functionality to address new needs. On domain focussed software development, invariant functionality usually represents domain commonalities, core functionality of the domain, that are reproduced from one application to another. Variant functionality is usually added or replaced in an incremental manner and represents a small percentage of the total implemented functionality of the system. This way of development is usually called an incremental evolutionary software development process.

An adaptive software development strategy is extremely well adapted to be used in this context. Such a strategy uses large frame structures as invariants (supporting the invariant functionality) and restricts variability to low level isolated locations within the overall structure. As it attempts to keep most of the overall structure invariant, adaptive strategy helps to keep development costs under control when reusing or leveraging invariant functionality from one existing system to the new one. However, because it also tend to be application specific and relatively inflexible, it may burden the cost of addition of variant functionality, increasing the overall costs of development of new systems.

To avoid this situation and minimize the costs of addition of variant functionality we need to add flexibility to those selected locations within the overall structure where variant functionality is to be plugged. In other words, the key point to master evolutionary software development is the creation of an evolution infrastructure (set of methods, tools and operating practices) that minimizes both, the cost of reusing/leveraging invariant functionality and the cost of addition of variant functionality. Methods and tools are those involved in the domain engineering activity. By operating practices we understood the set of well known rules and mechanisms that allow to obtain results on a systematic and repeatable way.

2.2 The Evolution Oriented Domain Engineering

It consist basically on three main activities:

1. Evolution specification,
2. Evolution control,
3. Instance implementation.

The evolution specification activity consists on the domain analysis, the domain modeling and the domain architectural design processes. We made the distinction between the domain analysis process referring to the activities of collecting and classifying data, and the domain modelling process as referring to the design of a formal structure for the descriptions. The combined results of those activities are the domain specifications model, and the taxonomy of domain variability. By domain specifications model we understood the definition of entities, operations, relationships and events that abstracts domain commonalities and variances across the systems, together with a classification of them. [2].

The result of the architectural design activity is the domain architectural model that represents the generic architecture (or set of architectures) of the domain from where specific instances are derived. The contribution of our approach is that a particular emphasis is put on the identification/prediction and characterization of variability, as we consider it is crucial to specify evolution. The taxonomy of domain variability is used to encode variance on the model through the specialization attributes. Each variance is represented by a set of different values of the specialization attributes. An instance (specific application of the domain) may be obtained by assigning a vector of values to the specialization attributes.

The evolution control activity characterizes the evolution induced by a new requirement according to a taxonomy of variability. In that activity, the new requirement is first translated into a set of specialization attributes. (i.e. on the network management domain of application, extending the management to support a new type of device, may impact specialization attributes like: management protocol, buffer management subsystem, line throughput performance,...) Then, a vector of values is assigned to the identified specialization attributes characterizing the instance to be implemented. (i.e. associates values to the already identified specialization attributes: management protocol = CMIP, buffer management subsystem = class 4, line throughput performance = 800 Mbit/sec, ...) Finally, if needed, models (domain specification and domain architectural) and taxonomy of variability are updated to include un-predicted, partially predicted or predicted and partially implemented variances. Note that this last operation is done only if the evaluation of benefits induced by the modifications appears to be greater than the costs of updating.

The instance implementation activity, is responsible for the implementation of the specific application to meet the new requirement. It extracts from the domain architectural model the specific architecture of the application to be implemented, identified in the previous step by the set of values assigned to the specialization attributes, and gets the corresponding specific design. This specific design is generally incomplete. It is composed of an implemented part representing the already implemented core functionality of the domain, and a not implemented part representing the not yet implemented core functionality of the domain and the not yet implemented variant functionality corresponding to the variability induced by the new requirement.

According to this situation, an implementation strategy is defined that may be summarized as follows:

- reuse already implemented large frames of software, corresponding to the invariant functionality, involved in the implementation of this specific instance,
- develop reusable software corresponding to the not yet implemented core functionality (typically by using a classical engineering for reuse approach, supported by the existing domain models), and
- implement specific variability maximizing the external reuse level. At this stage, a classical Design With Reuse approach should be used. Components to be reused may come for the software associated to the domain of application, from others domains or from general purpose libraries.

2.3 Identifying/Predicting and Characterizing Variability

The identification/prediction and characterization of variability are key mechanisms of the evolution oriented domain engineering process.

Identification/Prediction deals with evaluating factors of variability that influences variations in requirements within the domain of application. Among the most important are:

Market characterization : evolution and potential for expansion. Variations in functionality are mainly induced by market trends and market opportunities on mature domains, and by customer requirements on immature or new domains. Market expansion may also be a factor inducing changes in requirements: a growing market may result on a more competitive environment were the reduction of implementation costs of already existing functionality may conduct to changes on requirements (changes in hardware platforms, performances ...). Market analysis and customer characterization (of representative customers) are valuable mechanisms to predict changes/evolutions in functionality.

Technology evolution : Provides information on the number of instances that may be reused without any technology change and/or the level of abstraction required by the descriptions on the domain models to be technology independent.

Level of standardization : Is at the inverse of the two precedent a factor of stability. The highest the level the lower the variability. It may be used to isolate areas of the domain were the evolution will be none, a few or at least easily predicted. Those areas represents kernels of stable functionality around which variability may be articulated.

The taxonomy of variability results from the classification of variations identified during the evaluation of factors of variability. To each of the elements of the taxonomy, a factor of risk is associated, that evaluates its probability of occurrence. During the characterization, each of the elements of the taxonomy of variability is mapped to the set of corresponding specification attributes in the domain models. As a result, an internal variability map associating a level of variability to each specialization attribute and risk of occurrence to each of the instances of the specialization attribute is obtained. The level of variability of a specialization attribute is the number of different instances (represented by different values) associated to it. The risk of occurrence of an instance of the specialization attribute is the level of confidence associated to the prediction of occurrence of that instance.

What is extremely important is to build/modify the models, according to the internal variability map, to isolate specialization attributes with a high level of variability, group specialization attributes with low level of variability and define a strategy of implementation. The level of variability provides useful information to:

- Isolate specialization attributes with high level of variability into separate locations, what tends to minimize the cost of implementation of variability. The property of restricting high level of variability to isolated locations within the overall structure is what we call the model cohesiveness against variability.
- Group specialization attributes with low level of variability, what tends to increase the invariants areas of the model, defined when identifying common/stables parts of the systems in the domain, and by the way, contributes to maximize the reuse of large frames of software between applications. The risk of occurrence of an instance of a specialization attribute provides information to establish a strategy of implementation:
  - the higher the level of confidence associated to a prediction of variability, the lower the level of abstraction in the specification of the corresponding software.
  - the lower the level of confidence associated to a prediction of variability, the higher the level of abstraction in the specification of the corresponding software.
- Prediction of occurence of variability in the time provides complementary information for the implementation schedule.

3 Comparison

3.1 Modeling and Representing Domain Concepts

It is extremely recommended to adopt an object oriented approach to build domain models. Essentially because the concepts of abstraction/specialization (to capture commonalities and encapsulate variances), aggregation/decomposition (to master the complexity) and differentiation, critical to represent domain knowledge, are inherent to the OO representation. [3]

3.2 Qualifying Domain Models

The rules of construction of the models described above, leads naturally to define the following metrics, to qualify domain models:

Model coverage : Is the ability to provide reusable software measured by the amount of core functionality to be implemented by new applications that may be reused from the invariant part of the model.

Model predictability : Is the capability to predict/anticipate variance, measured by the cohesiveness (ability to locate variability on isolated areas of the model) and the costs of development anticipated by the strategy of implementation.

The already defined metrics involves lower level metrics such as software development productivity and component and system costs of reusable software [4, 5, 6]. Note that the model coverage and model predictability, inderectly measures , the cost benefits of reusing invariants and the cost of implementing variant, both key costs of the evolutionary software development.

A natural question rises here: what if for a new un-predicted variant, models show a poor coverage and a bad predictability? The first point to verify, is if the variant is outside of the predefined domain boundaries. If it is not the case and the variant is not an isolated example, but rather than that, it predicts a set of variabilities of the same type to arrive, domain models should be re-considered and a new step of domain engineering will probably be necessary. [7]

3.3 Conclusion and Further Investigation Directions

Up to now at Hewlett Packard Grenoble Networks Division we mostly experienced two approaches of reuse, the a-posteriori (or reverse engineering) and the a-priori (or domain reuse). They are essentially different on the process and results.

The a-posteriori approach is simpler, provides immediate return on investment and requires a lower level of reuse culture on the organization. As a drawback, the return on investment is burden by the adaptation costs induced by the re-engineering of components not initially designed to be reusable. The a-priori approach is more difficult and risky and requires a higher level of reuse culture of the organization. But it presents the advantage of providing high return on investment.

A natural way to introduce reuse in an organization is to start with the a-posteriori approach to build confidence and create success stories and to migrate incrementally to the a-priori as the culture on reuse increase and the change is accepted. We are now using both approaches and by comparison the a-priori approach is about three times better in terms of ROI than the a-posteriori. However, even in the best cases using an a-priori approach we were able to establish that the most limitative factor of the ROI are the un-predicted domain variances, that invalidates domain models by inducing important costs of addition of variant functionality. And obviously, the longer the domain life cycle the higher the risk. We believe, as confirmed by several first results, that an evolutionary software development approach, supported by an evolution oriented domain engineering, with a special focus on the identification/prediction and characterization of variability, really contributes to maximize reuse return on investments on the context of domain focussed software development.

Among the further directions of the research are:

- enhance the mechanisms of identification/prediction and characterization of variability together with their formalization,
- enhance the traceability of variability to increase the accuracy of impact analysis and record the evolution knowledge,

- provide appropriate tools to support the traceability of variability (research on the area of hypertext based tools),
- extension/tuning of the reuse economical costs models, used as decisions tools by the intensive computation of object oriented productivity metrics involved on them.

Further research activities will be done on the context of the European Research ESPRIT project PROTEUS, and in close collaboration with Hewlett Packard Corporate Engineering and Hewlett Packard Laboratories. The areas of technology transfer are essentially Hewlett Packard operational divisions and some European companies member of the PROTEUS consortium. Domain of applications were this evolutionary approach is being applied today are Telecommunication and Datatcommunication Networks, and Aerospace applications.

References

[1] Barness and Bollinger, "Making Reuse Cost Effective," *IEEE Software*, January 1991.

[2] R. P. Diaz and G. Arango, "Domain Analysis and Software Modelling," in *IEEE Computer Society Press Tutorial*, IEEE Computer Society Press, 1991.

[3] J. Rambough and All, *Object Oriented Modelling and Design.* Prentice-Hall, 1991.

[4] D. Balda and D. A. Gustafson, "Cost Estimation Models for the Reuse and Prototype Software Development Life Cycle," *ACM Sigsoft Software Engineering Notes*, vol. 15, p. 42, July 1990.

[5] G. Mayobre, "Using Code Reusability Analysis to Identify Reusable Components from SW Related to an Application Domain," in *WISR 1991 Proceedings*, Department of Computer Science, University of Maine, 1991.

[6] Caldeira and Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, February 1991.

[7] B. Boehm, "A Spiral Model of Software Development and Improvment," *IEEE Computer*, vol. 21, May 1988.