Dealing with Unstable Domains in Product-Line Architecture Development

Valentino Vranić *
vranic@fiit.stuba.sk, http://www.fiit.stuba.sk/ vranic/

Vladimír Marko*
marko@fiit.stuba.sk

Abstract: Application of domain engineering approaches, which represents the basis for establishing product lines, normally subsumes a stable and well understood domain. This may prevent many projects from gaining a benefit of the organized development for reuse enabled by domain engineering techniques. This article explores how to develop the architecture of a domain under a change. The approach is based on a thorough exploration of well-understood part of the domain by the means of feature and use case modeling. This is followed by a generalization of the use case view and interactive development of the subsystem and component view. The approach is illustrated by examples from the project on knowledge management whose development part is performed concurrently with the ongoing research activities.

Keywords: Archetype, Component, Feature Modeling, Product Line, Subsystem, UML, Unstable Domain, Use Case

1 Introduction

Reusability is one of the most desirable properties software can have. To achieve this, software must be developed with reuse in mind from the start. Even then, real reuse is possible only for a group of related software products. Such groups are being denoted as software product families.

Domain in the sense of domain engineering represents an area of knowledge scoped to the needs of its stakeholders; it includes a set of concepts and terminology of the respective area and the knowledge how to build software systems in that area [5]. This means that a domain is expected to be stable and well-understood, which is not always achievable.

This article addresses the issue of dealing with unstable domains in product-line architecture development. One may encounter such domains in research projects which embrace a development component (mostly for evaluation purposes). In such projects, in order to meet the schedule, development has to start early, which is often before the research has been completed. Here we speak of product lines rather than product families because products are grouped mainly by the need of the specific project, which is an analogue of the market demand in industrial projects.

The rest of the article is structured as follows. Section 2 gives an overview of the approach. Section 3 describes how to approach the unstable domain by first exploring a limited portion of it by the means of feature and use case modeling. Section 4 describes the subsequent generalization of the feature and use case model. Section 5 is devoted to evolving the structure of the systems in the domain based on the feature and use case model. Section 6 discusses the proposed approach in the context of related approaches. Section 7 concludes the article.

* Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology, Ilkovičova 3, 84216 Bratislava 4, Slovakia

2 Approach Overview

Figure 1 gives an overview of our approach to developing product-line architecture in an unstable domain. It shows the main artifacts and their dependencies. The whole process starts by exploring a limited portion of the domain in order to obtain a partial domain model. Typically, this part of a domain should represent one possible product in the domain. First, commonality and variability in the domain has to be explored at least at a coarse level. A useful technique to do this is feature modeling. Functionality of this domain part is captured by use cases. Each use case should be linked to related features in order to document functionality variants.

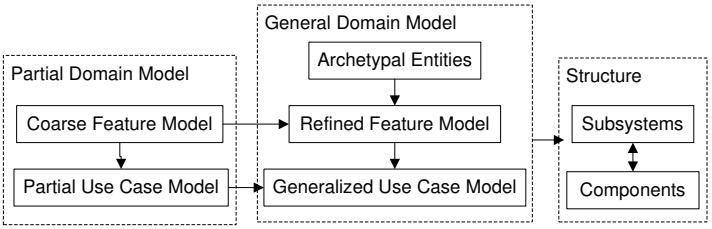

Fig. 1: Overview of the approach.

The next step is generalization. The feature model should be further refined as its purpose is to control the configuration of individual products. However, the use case model will undergo more significant changes as we have to switch from the specialized viewpoint to a general one based on archetypal entities (explained in Sect. 4.1) and their interactions.

Finally, the structure of the products in the domain based on the feature and use case model is evolved in the form of subsystem and component views. In this step, an opportunity to apply architecture styles or patterns should be sought.

3 Exploring the Domain

Configurability is crucial to product lines. In our approach, we start by a coarse examination of commonality and variability using feature modeling. Subsequently, focusing on one special product, we try to express the expected functionality in greater detail in a use case model.

3.1 Feature Model

Feature modeling is a conceptual domain modeling technique in which concepts in a domain, understood broadly as an area of interest [4], are being expressed by their features taking into account feature interdependencies and variability in order to capture concept configurability. Introduced by FODA [6], feature modeling was later improved by Czarnecki and Eisenecker [5], as well as by subsequent work (see [10] for an overview).

In feature modeling, a concept represents an understanding of a class or category of elements in a domain. A feature is an important property of a concept [5]. Any feature may be isolated and modeled further as a concept, therefore being a feature is actually a relationship between two concepts, but the concepts identified only in the context of other concepts, i.e. as their features, are usually referred to as features [10]. In general, a feature may be *common*, which means it is present in all concept instances, or *variable*, which means it is present only in some concept instances.

The most important part of the information in feature models is presented graphically by feature diagrams. Figure 2 shows the *Knowledge Acquisition* concept feature diagram. This concept is a part of the domain of knowledge management which encompasses application for acquisition, organization, and maintenance of knowledge in the web. This domain has been analyzed in our project on tools for acquisition, organization, and maintenance of knowledge in an environment of heterogeneous information resources. The project is spanned over four organizations and it encompasses a significant development component whose objective is to create a knowledge portal based on the sophisticated methods being explored in this project.

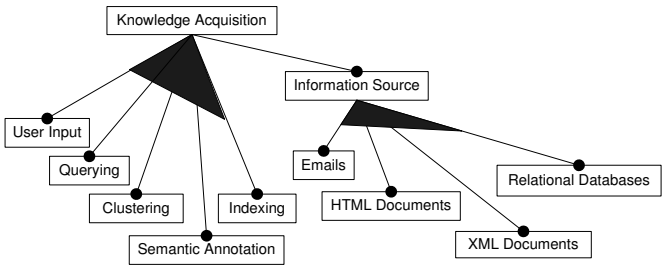

Fig. 2: Knowledge acquisition concept.

The *Knowledge Acquisition* concept represents approaches to knowledge acquisition. We identified several such approaches ranging from the direct user input to semantic annotation. Multiple approaches can be employed in the final product, but at least one has to be present. This is expressed by grouping features that represent approaches to knowledge acquisition into an or-feature group (indicated by a filled arc).

Information Source is another important feature of the *Knowledge Acquisition* concept. This feature is mandatory, i.e. it has to be present in all products (indicated by a filled circle). Several types of information sources are possible and they also constitute a group of or-features.

Other concepts identified in the domain include, for example, information domain and presentation options. These two concepts along with *Knowledge Acquisition* actually represent mandatory features of the *Knowledge Portal*, which represents the main concept in the domain. User adaptability is another concept that appears as an optional feature of the *Knowledge Portal*.

3.2 Use Case Model

Use case modeling has been used to capture stakeholders and functional requirements imposed onto system under development. The objective is to achieve a grasp of core functions while abstracting from realization details to avoid premature breakage into functional blocks misregarded for functional development units before all of the functionality has been understood.

More specifically, variations in functionality, such as support of various input methods and output formats, are not reflected as use cases of their own. Internal working and methods employed to achieve required functionality hidden from the standpoint of an stakeholder is also omitted from the description of use cases. Stakeholders are modeled as actors of the use case model. Some internal mechanisms are modeled as worker actors as well, as is the case of *Job tracker* internal process which represents proactive behavior on behalf of the system (see Fig. 3).

Each use case has to be provided with description, possibly specified by a set of scenarios, and a list of related concepts and features. The mapping of use cases to features allows to capture the variations in functional requirements. As the mapping to the features captures the

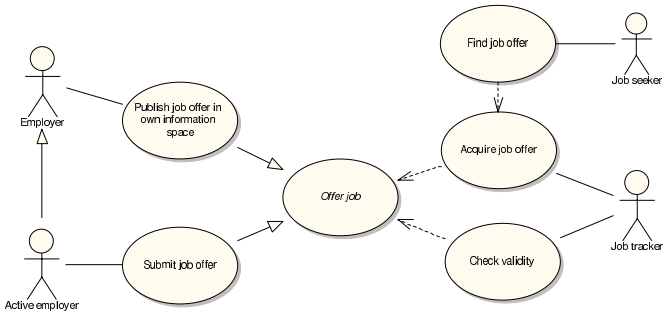

Fig. 3: A domain exploration level use case diagram.

possible variations in the functionality modeled by a use case, there is no need to describe such variations further at this level, be it in the form of specialized use cases or variants of a scenario. The general rule is not to expose the realization of a functionality as *Optional Use Case* or *Use Case Sequence* pattern [7].

The use case diagram in Fig. 3 represents only a part of the functionality required for acquisition and presentation of job offers. *Make offer* abstract use case is specialized into two concrete ways of publishing job offers for use in our system. However these two use cases do not differ only in their realization, but in their mode of operation in the first place. They are activated by different stakeholders: by an employer who is not aware of our system and by an employer who actively uses the system. *Acquire a job offer*, *Check validity*, and *Find a job offer* use cases do not form a use case sequence as they can exist independently apart from a possible interdependency ensuing from their mutual dependence on *Offer job* use case.

Find job offer use case represents a query of a *Job seeker* on the accumulated job offer store. A variety of realizations were to be considered for this function, which has been modeled by mapping the use case to the *Knowledge acquisition* concept (recall Fig. 2).

4 Domain Generalization

Domain generalization is based on the partial domain model: the objective is to identify the archetypal entities of the domain and relations between them, refine the feature model, and generalize use cases.

4.1 Archetypal Entities

Identification of archetypal entities and their interactions is a major transition the domain model must undergo. The level of abstraction we are looking for lies in between the abstract feature model and concrete use cases that constitute our domain model presented so far. Though it cannot be expected for archetypal entities just to drop out of these two views, a good starting point is to analyze the important concepts from the feature model and corresponding use cases.

In our example, knowledge acquisition is such a concept. Use cases that correspond to this concept are about job offer acquisition in a special case of knowledge acquisition or, more precisely, a special case of *offer acquisition*. Thus, we actually may narrow our domain to offer acquisition as such.

In a domain of offer acquisition, we speak of an abstract *offer* which will, with the development of concrete components, become a job offer, travel offer, apartment renting offer, etc.

Each offer has two faces: the one is of the offer source—a *producer*—and the other of the offer target—a *consumer*. Thus, our archetypal entities include an offer, producer, and consumer.

It is important to note the relativity of these archetypal entities. A job offer can be perceived as an offer of a job to potential employees. In this case employers are producers, while employees are consumers of the offer. However, it is also possible that an employee will seek the job by exposing his offer which will turn him into a producer, and employers into consumers.

4.2 Feature Model Refinement

Identification of archetype entities represents a major shift in domain understanding. This requires the feature model refinement. One of the consequences for the *Knowledge Acquisition* concept in our feature model is the separation of the information content independent acquisition (denoted as *Data Acquisition*) from the dependent one based on the identified archetypal entities (denoted as *Offer Acquisition*). Figure 4 shows the refined feature diagram of the *Knowledge Acquisition* concept.

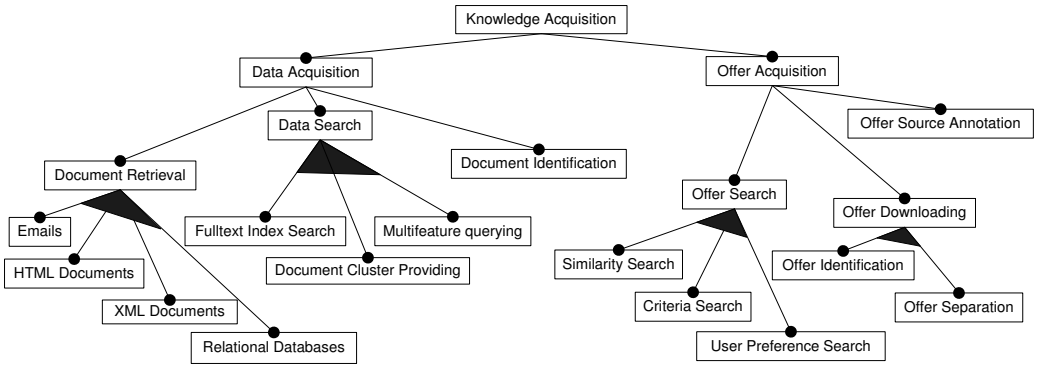

Fig. 4: Knowledge acquisition concept refinement.

4.3 Generalized Use Cases

Subsequently, generalized use cases based on archetypal entities can be developed. In addition to introduction of archetypal roles, the use case model has to be evolved further using the refined feature model. The identified features should be used to split the functionality into more manageable units. The objective of this process is to obtain a use case model which can be mapped to structural view of the software. Therefore, each use case has to be considered as a collaboration of several actors, some of which represent users of the system, while others are internal system components. The use case itself then represents a concrete usage of a subsystem by another actor.

Figure 5 shows a view derived from the specialized use case presented in Sect. 3.2. The *Active producer* user role collaborates with the *Offer database* subsystem actor to facilitate the *Enter offer* use case. The directionality of the usage relation is not established at this stage as it needs to be addressed only after overall requirements are analyzed in order to select suitable architectural style (discussed in Sect. 5).

Again, it is advisable to avoid unnecessary functional decomposition at this stage that can be carried out later within subsystem and/or component view of the system. Now it would

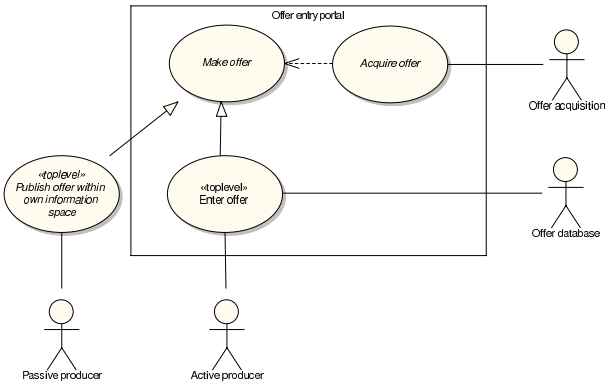

Fig. 5: Generalizing use cases.

lead to proliferation of actors and single-actor collaborations which are not bearing significant information on the subject of identification of components of the system.

Representing variations of requirements as separate use cases should also be avoided where possible. The only exception from this rule is the case when variations in functionality require different collaboration or different collaborating actors constitutes (see in Fig. 6). Specializations of the *Acquire offer* use case are performed as collaborations of different subsystem actors.

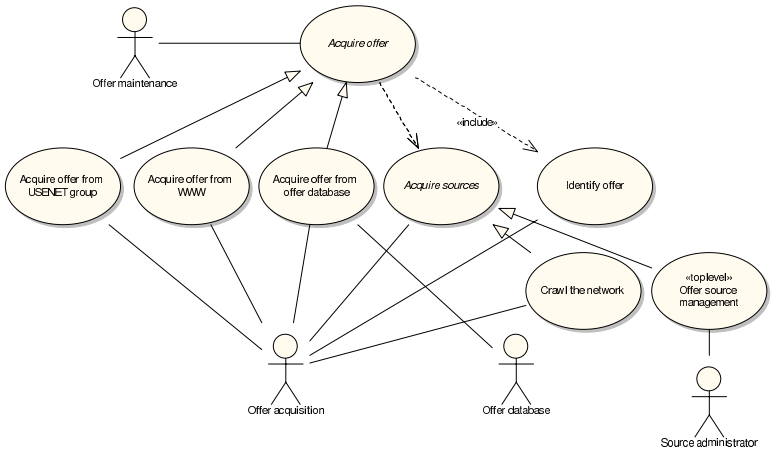

Fig. 6: Generalizing use cases.

5 Structure View

Finally, the system structure can be derived from the functionality captured in the use case model. Letting the behavior form the structure enables to avoid the unpleasant consequence of the Conway's law [3] by which the structure of the developing organization ultimately shapes the system being developed.

In our approach, two distinct levels of structural decomposition are employed. The first level comprises decomposition to subsystem according to logical cohesion among offered functionality. This view corresponds to the *module viewtype* style [2] and is derived directly from the refined use case model of the system. The internal actors from the refined use case model rep-

resent subsystems. As an example, consider the subsystem diagram in Fig. 7. *Offer acquisition*, *Offer database*, and *Offer maintenance* subsystems correspond to subsystem actors in Fig. 6.

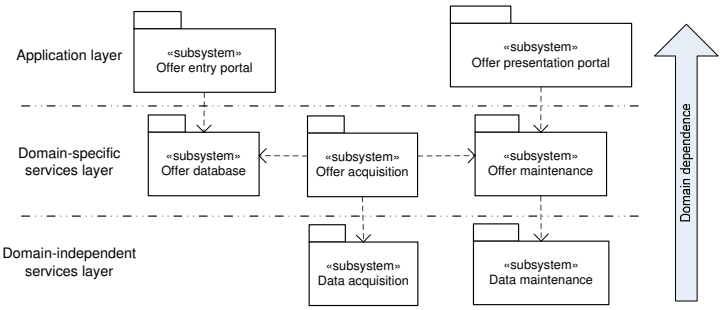

Fig. 7: A fragment of the subsystem diagram.

Some subsystems do not collaborate in any functionality specified by use cases, but merely pose interfaces to the external environment and most notably user interaction. In our example, this is facilitated by the top layer subsystems which were derived from the system boundaries introduced in the use case model.

Subsystems can be understood as a foundation for building components. The component view prefers functional cohesion resulting into packages that offer the functionality adhering to prescribed interfaces. Often, multiple components implement a common interface, and this usually ensues from variations in functionality identified as variable features in a feature model.

The two levels allow for distribution of development among teams by splitting system into reasonably sized functional components while preserving clear insight of the system as a whole through subsystem dependencies. Subsystem structure can be utilized for source package structuring and dependency management whereas component view reflects into runtime modules.

6 Related Work

The risks of developing product lines in immature domains have been analyzed by Voget and Becker [9]. More specifically, they deal with the risks of *uncertain technological evolutions*, which actually correspond to the domain instability in projects with ongoing research addressed in our approach, and propose to resolve them by employing light-weight domain engineering, stable subdomains start-up, commonality-oriented assets, and sound variability treatment support. While Voget and Becker's stable subdomains start-up strategy is merely about isolating unstable subdomains and excluding them from the product line, we focus on stabilizing such subdomains in order to preserve them as a part of the product line.

The notion of an archetypal entity in the context of the approach proposed in this article is related to Bosch's archetypes, the core abstractions on which the system will be structured [1]. These are further described as highly abstract and stable, which are the properties of our archetypal entities, too. However, while Bosch warns of deriving archetypes from the concrete instances in the domain and proposes to rely on the good understanding of the domain and developer's insight, our experience is that in an unstable domain one has to take the concrete (and specific) instances into account along with the abstract domain view.

Furthermore, Bosch's archetypes are more structural in their nature; concrete systems are populated by instantiation of archetypes. Our archetypal entities do not necessarily represent abstract structure of systems and may include external entities such as users.

7 Conclusions

In this article, we proposed an approach to dealing with the architecture design in unstable domains in order to enable exploiting the benefits of product lines in such domains. The approach is presented on examples from a project on knowledge management whose development part is performed concurrently with the ongoing research activities causing the instability in a respective domain of knowledge acquisition.

The approach shows that improved understanding of a specific—but important—part of a domain in terms of its functionality and configurability can be translated to the whole domain. This process is intrinsically functionality driven; structural decomposition is postponed until the behavior of the systems in the domain is sufficiently explored.

The most critical step of the approach—identification of archetypes and their interactions—is principally highly dependent on the insight of developers. However, it is our experience that the very existence of a partial domain model significantly improves the communication with domain stakeholders, which is inevitable for the process of generalization.

This work was supported by the Science and Technology Assistance Agency of Slovak Republic under the contract No. APVT-20-007104, State Programme of Research and Development "Establishing of Information Society" under the contract No. 1025/04, and Scientific Grant Agency of Slovak Republic (VEGA) by the grant No. VG 1/3102/06.

Bibliography

1. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
2. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.
3. J. O. Coplien. A development process generative pattern language. AT&T, 1995. Available at `http://users.rcn.com/jcoplien/Patterns/Process/` (accessed in Aug. 2005).
4. J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.
5. K. Czarnecki and U. W. Eisenecker. *Generative Programing: Methods, Tools, and Applications*. Addison-Wesley, 2000.
6. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, Nov. 1990. Available at [8] (accessed in Mar. 2002).
7. G. Overgaard and K. Palmkvist. *Use Cases Patterns and Blueprints*. Addison Wesley, 2004.
8. Software Engineering Institute, Carnegie Mellon University. Home page. `http://www.sei.cmu.edu`. Accessed in Mar. 2004.
9. S. Voget and M. Becker. Establishing a software product line in an immature domain. In G. J. Chastek, editor, *Proceedings of 2nd International Software Product Line Conference (SPLC2)*, LNCS 2379, pages 62–77, San Diego, USA, Aug. 2002. Springer.
10. V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.