A survey on domain engineering

Maarit Harsu
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere
e-mail: firstname.lastname at tut.fi

Contents

1 Introduction

Software product lines provide a way to design and implement several closely related systems together. Such a *family of applications* share a lot of common features, and thus, it is reasonable to produce the members of the same family together, in order to take full advantage of reusable requirements and other common elements.

To make reuse possible, software engineering is divided into two parts: *domain engineering* to find and implement the common features and *application engineering* to produce the individual applications. This report concentrates on domain engineering. The purpose is to describe the research area concerning domain engineering, to find a consensus between inconsistent terminology of the field, and to find the most important reseach problems in the field.

2 Domain engineering overview

There are several, varying definitions for domain engineering, for example [CE00, JGJ97, Sof00a]. Common for these definitions is that the purpose of domain engineering is to provide reuse among similar applications (i.e. an application family). Domain engineering is a systematic process to provide a common core architecture for these applications. It can be applied both to existing systems and to newly-engineered systems. Thus, domain engineering covers analysis of either existing applications or the concepts of a problem area.

The get a conception of domain engineering, this section considers the term ”domain” more precisely, discusses the context of domain engineering, divides domain engineering into parts, distincts domain engineering from closely related terms, introduces some domain engineering methods, and compares them to the methods concerning object-oriented analysis and design.

2.1 Domain

This subsection concentrates on domains in general. It gives a definition for the term, shows how domains and systems can be related with each other, and considers different types of domains.

2.1.1 Definition of domain

A *domain* can be defined by a set of problems or functions that applications in that domain can solve [Tra94]. A domain can also be characterized by a common jargon for describing the concepts and problems in that domain [Tra94].

The term ”domain” can be used in several associations [Sch00]:

- business area
- collection of problems (problem domain)
- collection of applications (solution domain)
- area of knowledge with common terminology.

To determine a domain, several approaches exist. The view can be focused on describing what is inside the domain, what is the boundary of the domain, or what is outside the domain [Sch00]. The first case describes the items that constitute

the domain, or it identifies other domains that together form the actual domain (domains can have sub-domains). The second case describes the rules of inclusion and exclusion. The third case (as well as the first one) can be depicted by structure and context diagrams. However, domain boundaries are not necessarily easy to find [LM97]. This may be due to the inexperience of the domain analyst and to such domains that use or provide services to other domains.

Domains can be considered at least from two point of views [Sim97a]. First, as considered in object-oriented context, a domain can be seen as real world. This approach concentrates on the phenomena and their processes. The requirements of different applications are not cared about. Thus, this kind of domain engineering is rather similar to conceptual modeling, and it covers the first two items of the previous list. The domain model, derived from this alternative, is highly reusable, because the concepts of a problem area are rather stable. They do not necessarily vary, although the requirements of the systems change.

The other way to consider a domain is to see it as a set of systems. This point of view concentrates on application families. The domain is bordered according to the similarities between the applications. Thus, this approach covers the third item of the previous list. This latter alternative is more close to product-line engineering, while the former one suits also single-system engineering.

2.1.2 The relationship between domains and applications

Applications may belong to several domains. For example, an application concerning distributed banking practices covers at least the following domains: banking practices, commercial bank information systems, workflow management, user interfaces, database management systems, and networking [CN02]. In addition, applications do not necessarily cover a whole domain or several entire domains. On the contrary, several domains may be scattered under one application. The both situations are shown in Figure 1, when applications are presented as boxes and domains as areas.

Moreover, the relationship between domains and applications is considered in [WS94]. If the domain functionality is covered by a single subsystem in one system, the domain is called to be *encapsulated*. Instead, if the domain functionality is scattered through several subsystems of one system, the domain is said to be *distributed*. Distributed domains can also be called *diffused* domains [CE00].

Forward engineering usually deals with encapsulated domains, because it is pos-

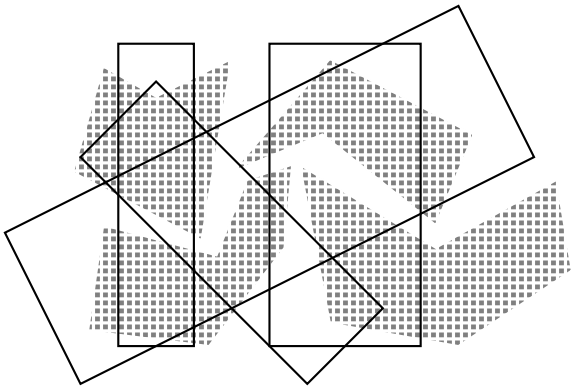

Figure 1: Relationship between domains and applications [Sch00]

sible to divide the system to be engineered according to existing domains. However, applying domain engineering into legacy systems leads more probably to distributed domains. In such a case, it is necessary to study several systems to find out the desired domain architecture [WS94].

2.1.3 Vertical and horizontal domains

Domains can be divided into vertical and horizontal domains [CE00]. In *vertical domains*, software systems are classified according to the busines area. Such systems are, for example, airline reservation systems, medical record systems, portfolio management systems, order processing systems, and inventory management systems. In *horizontal domains*, parts of a software system are classified according to their functionality. Examples are database systems, container libraries, workflow systems, GUI (graphical user interface) libraries, and numerical code libraries.

As shown in Figure 2, vertical domains are domains of systems, while horizontal domains are domains of parts of systems [CE00]. When applying domain engineering to a vertical domain, the result could be reusable software that can be presented as a form of a reusable framework and components. In the case of horizontal systems, domain engineering may yield reusable components.

As in the case of distributed domains, also horizontal domains are more typically found in legacy systems. However, at least large or complex systems can be scat-

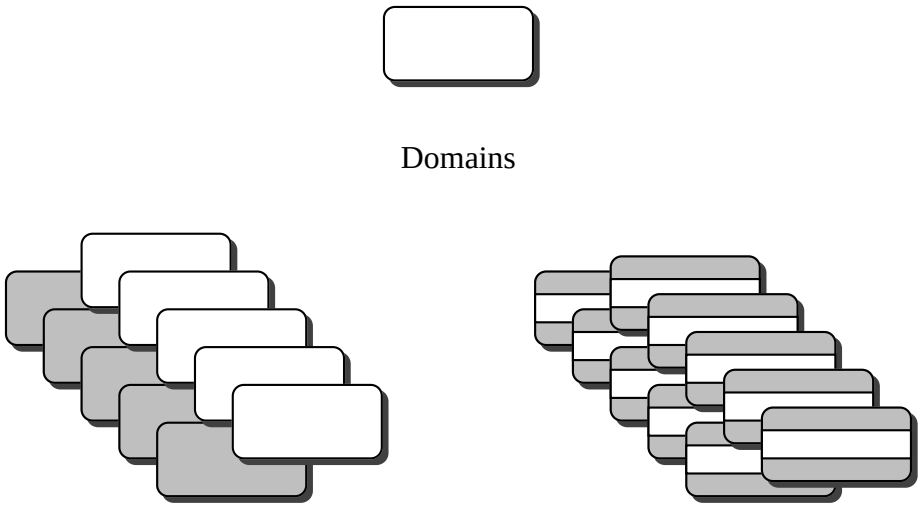

Figure 2: Vertical vs. horizontal domains [WS94]

tered above several domains even from the beginning, as described in the example of the distributed banking system in Subsection 2.1.2.

Horizontal domains can be considered from the point of view of lateral domains [Sim97b]. *Lateral domains* are horizontal domains in a subsystem level, and thus, they can also be called *component domains*. They correspond to a set of closely related components (or asset bases) that can belong to several systems as their subsystems.

2.2 Domain engineering and application engineering

To enable product-line engineering, a well-accepted convention is to divide the engineering process into two different processes: domain engineering and application engineering (for example, see [CE00, Sof00a, TP00, WL99]). These processes can be followed in parallel, as depicted in Figure 3.

Domain engineering and application engineering can be called *engineering-for-reuse* and *engineering-with-reuse*, respectively. The purpose of domain engineering is to provide the reusable core assets that are exploited during application engineering when assembling or customizing individual applications. Thus, these

phases can also be called *assets engineering* and *product engineering* [TP00], or *core asset development* and *product development* [CN02], respectively.

The subprocesses of domain engineering (considered more precisely in Subsection 2.3) and application engineering correspond to each other. The phases of domain engineering concentrate on the design and implementation of the core architecture, while the phases of application engineering deal with the design and implemention of individual applications. Note that the names of the phases vary in different references. For example, in [Sof00a], the phases of application engineering are called *requirements engineering*, *design analysis*, and *integration and testing*. However, most authors agree that the phases of domain engineering are domain analysis, domain design, and domain implementation (as depicted in Figure 3).

Domain engineering can also be compared with conventional software engineering [CE00]. *Requirements analysis* corresponds to domain analysis such that requirements analysis produces requirements for a single system while domain analysis produces reusable configurable requirements for a family of systems (more precisely in Subsection 2.3.1). Similarly, *systems design* finds design of one system, while domain design concentrates on reusable design for a class of systems and a production plan (more precisely in Subsection 2.3.2). Finally, *system implementation* implements one system, while domain implementation produces reusable components, core architecture, and production process (more precisely in Subsection 2.3.3).

2.3 Domain analysis, domain design, and domain implementation

Domain engineering is most often divided into three phases: domain analysis, domain design, and domain implementation (for example [CE00, Sof00a]). This is also shown in Figure 3. However, some references (for example [WL99]) divides domain engineering into two parts called domain analysis and domain implementation. In this case, domain analysis covers also tasks from domain design. This report follows the aforementioned division into three phases (as shown in Figure 3) and applies it in the division of the current subsection.

In the literature, the terms concerning the phases has been used inconsistently. For example, domain engineering and domain analysis has been used interchangeably. However, more recently domain analysis is established as a part of domain engi-

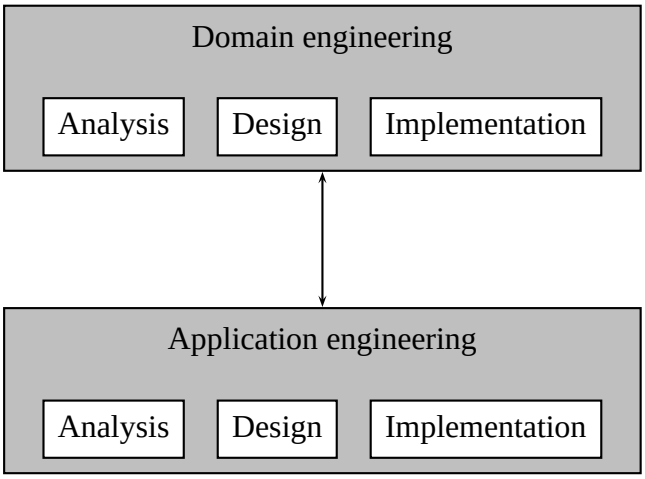

Figure 3: Product-line engineering phases

---

neering.

2.3.1 Domain analysis

Domain analysis is first introduced by Neighbors to denote studying the problem domain of a family of applications [Nei80]. (The corresponding term for single systems is *systems analysis*.) Domain analysis is associated with reuse, its purpose is to capture information involved with the domain to be reused in developing further applications of the same domain [Pri90]. However, domain analysis is not necessarily restricted to product-line engineering, instead, it can be used in the context of single-system engineering, too.

The output of domain analysis is *domain model*. However, different references in the literature are inconsistent about the artifacts or processes belonging to domain model. Trying to find the most common elements from several domain models, the ingredients of domain model can be sketched as follows:

- domain scoping (domain definition, context analysis)
- commonality analysis

- domain dictionary (domain lexicon)
- notations (concept modeling, concept representation)
- requirements engineering (feature modeling)

In the above list, the first term after each bullet denotes the term used in this report, while parenthesis enclose alternative terms for the same concept. A brief description is given for each item below.

Domain scoping finds out the boundaries of the domain. This activity provides examples of the applications belonging to the domain, as well as counter-examples of such applications that are outside the domain. In addition, it gives the rules of inclusion and exclusion.

Note that scoping can be divided into domain scoping, product-line scoping, and asset scoping [Sch00]. Domain scoping is associated with domain engineering (as described above). *Product-line scoping* identifies requirements and products that should belong to a certain product line. *Asset scoping* identifies reusable core assets.

Commonality analysis considers both the commonalities and variabilities of the application in the domain. It studies the requirements and properties of the applications and the concepts in the domain. Commonality analysis produces a *commonality document* that could be like presented in Table 1 [ADH+00].

Domain dictionary provides and defines the terms concerning the domain. Its purpose is to make communication among developers and other stakeholders easier and more precise. Note that domain dictionary may be part of commonality document (see Table 1, point Definitions).

Notations provide a uniform way to represent the concepts used in domain modeling. Examples of such notations are object diagrams, state-transition diagrams, entity-relationship diagrams, and data-flow diagrams. According to [LM97], it is reasonable to use such a notation that is familiar for most of the stakeholders.

Some notations are meant to describe a system, and thus, they can be called *system-level* notations. It is not easy or even desired to use these notations to describe *domain-level* concepts. For example, UML (Unified Modeling Language) [RJB99] can be used to describe variability within a single system. However, according to some authors, it could be confusing to use (or expand) the same notations to describe also variability among systems [SB98]. Thus, both familiar and

Table 1: Commonality document

Overview	Description of the domain and its relation to other domains
Definitions	A standard set of technical terms
Commonalities	A structured list of assumptions that are true for every member of the family
Variabilities	A structured list of assumptions about how family members differ
Parameters of variation	A list of parameters that refine the variabilities, adding a value range and binding time for each
Issues	A record of important decisions and alternatives
Scenarios	Examples used in describing commonalities and variabilities

new notations have their advantages and disadvantages. On one hand, familiar notations and their extensions are easy to learn. On the other hand, they may lead to misunderstandings if they are used in inappropriate situations or abstraction levels. However, choosing a notation is an important decision because notations have a key role in giving an understanding of a system or a domain.

Requirements engineering comprises gathering, defining, documenting, verifying, and managing the requirements that specify the applications in the domain [CN02]. In the product-line context, the purpose is to reuse and configure the requirements among individual applications. Such requirements can also be called *features* [CE00]. A feature can also be defined as a characteristic of the system that is visible to the end user [KCH+90], or it can comprise also characteristics that are relevant to some stakeholder (not necessarily visible) [SCK+96].

Features can be depicted by feature models that also tell which combinations of features are meaningful. Feature models provide notations for different kinds of features such as the FODA-like features, i.e. mandatory, optional, and alternative features [KCH+90]. However, these three feature types are not always adequate. For example, from alternative features you can select exactly one. In some situations, it should be possible to choose any number of features from a given set of features. Besides different kinds of features, feature models may provide a way to present constraints among features or rationales to select features.

Figure 4 shows a feature diagram of a simple car. Mandatory features are marked with filled circle in the head of the line. For example, a car must have a body,

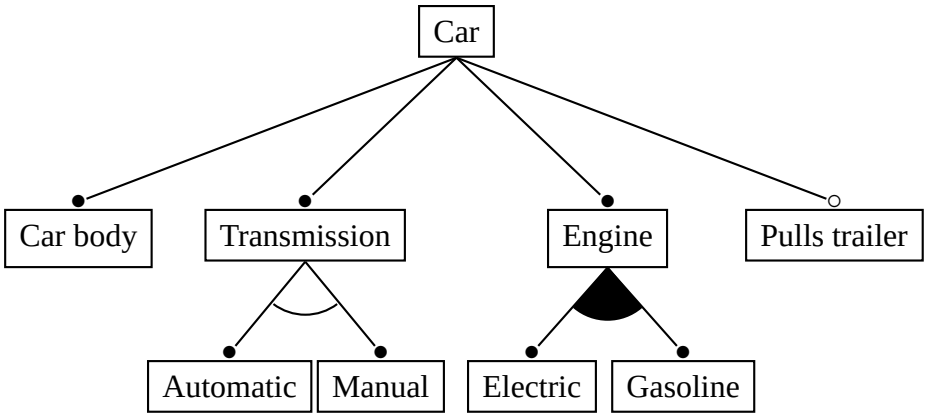

Figure 4: Feature diagram of a simple car [CE00]

transmission, and an engine. Optional features have an empty circle in the head of the line. For example, a car can either pull a trailer or cannot. Alternative features are connected with an empty arc. For example, the transmission of a car can either be automatic or manual, but not both. A filled arc connecting features denotes or-features [CE00], meaning that you can choose one or several of such features. For example, a car may have an electric engine, a gasoline engine, or both.

Feature modeling is very close to conceptual modeling. Actually, the top item in a feature diagram (Car in Figure 4) is a concept having several properties called features. Each feature may have subfeatures (as can be seen in Figure 4). In addition, features have a very close connection to variabilities. Division into the subfeatures corresponds to a variation point. There are different kinds of variation points according to the division points of subfeatures. For example, a variation point corresponding to the division point of optional subfeatures is different from the one corresponding to the division point of or-features.

Besides the aforelisted tasks concerning domain model, domain analysis usually comprises information gathering from different sources such as several documents and domain experts. In addition, it may cover clustering, abstracting, classifying, and generalizing of domain concepts. Note that domain analysis is different for different applications and for different domains [CE00]. For example, interactive systems may require use cases and scenarios. In addition, data-centric applications can be more easily described by entity-relationship diagrams or object diagrams. Special properties such as real-time support, distribution, and fault tolerance may require special modeling techniques. Properties peculiar to a certain

organization may set additional requirements.

Note that, domain analysis can also be called as *domain modeling* referring to the output i.e. domain model. In addition, there are other inconsistencies in the terminology and in the contents of the subareas concerning domain analysis. According to this report (which uses the most common subarea division), domain scoping is a part of domain analysis. However, both domain scoping and domain analysis can be represented as parts of domain engineering [Sof00b]. Moreover, this report follows the approach that both domain scoping and requirements engineering are parts of domain analysis. However, domain scoping may be considered as a part of requirements engineering that, in turn, belongs to domain analysis [Sof00a].

2.3.2 Domain design

Domain design means designing the core architecture for a family of applications. It comprises the selection of the architectural style [BMR+96, SG96]. In addition, the common architecture under design should be represented using different views (for example [Kru95]). The core architecture should also provide variability between applications. In this phase, it is decided how to enable this variability or configurability. According to feature models and commonality documents, it should also be selected which components or items (such as requirements) are provided in the core architecture and which items are implemented as variations in individual applications.

Domain design produces also *production plan* telling how the concerete application can be derived from the core architecture and from the reusable components. Production plan comprises descriptions of the systems with their interfaces per each customer, guidelines for the assembling process of the components, and guidelines for managing the change requests from the customers [CE00]. The assembling process may be manual, semi-automatic, or automatic.

Domain design may be involved in assessing the core architecture, i.e. analyzing the architecture against its quality requirements to reveal potential risks concerning the architecture. (Architecture evaluation is considered more precisely, for example, in [CN02] and in [Lah02]). There are different opinions of the most suitable time to assess an architecture. According to [ABC+97], an architecture should be evaluated at an early stage of the development of a product-line architecture, because the sooner the problems are detected the easier they are to solve. Thus, as soon as there are artifacts to be evaluated, it should be checked how well the architecture meets its requirements.

Composing and other actions to provide final applications may require special tools that are outlined in the design phase. An environment constituted of such tools can be called an *application engineering environment* [WL99].

2.3.3 Domain implementation

Domain implementation covers the implementation of the architecture, components, and tools designed in the previous phase. This comprises, for example, writing documentation and implementing domain-specific languages and generators.

The purpose of domain engineering is to produce reusable assets that are implemented in this phase. Thus, the result of whole domain engineering phase comprises components, feature models, analysis and design models, architectures, patterns, frameworks, domain-specific languages, production plans, and generators.

Note that domain engineering is a continuous process. The knowlegde concerning the domain should be maintained and updated all the time according to new experience, scope broadening, and new trends [Sim91]. In addition, domain engineering should adapt according to the feedback from application engineering. Thus, domain model can never be completed, it could always be refined to be more accurate. In addition, domain model usually contains some kind of compromize about different and perhaps inconsistent views from several experts.

2.4 Distinction from related terms

The field concerning domain engineering and product-line architectures has several concepts the contents of which are not very clear. This subsection compares the terms ”domain”, ”domain engineering”, and ”product-line architectures” with some very related terms.

2.4.1 Product line

The terms ”domain” and ”product line” are very close to each other. However, the difference is that a domain consists of conceptual items, while a product line comprises concrete products or applications to be developed [Sch00]. Moreover, these terms can be considered to refer to the same level of abstraction, but different terms are meant for different audience [Pou97]. Technical persons (software

developers) understand and use the term ”domain”, while for managers it is easier to talk about product lines, because it is more like a business or a marketing term, as noted also in [CE00].

2.4.2 Requirements engineering

Requirements engineering can be divided into subtasks such as *requirements elicitation* to discover and understand the user's needs, *requirements analysis* to refine the user's needs, *requirements specification* to document the user's needs, *requirements verification* to ensure that the system requirements are complete, correct, and consistent, and *requirements management* to schedule and coordinate the above activities concerning requirements [DT97]. However, like domain analysis and domain engineering, also requirements analysis and requirements engineering are sometimes used interchangeably.

Note that requirements engineering is involved in Figure 3 and in Subsection 2.2 in a sense that requirements analysis for single systems corresponds to domain analysis for a family of systems. In addition, requirements engineering is a part of commonality document (in Subsection 2.3.1), and it has a close connection to features.

Although requirements analysis can be considered as domain analysis for single systems, it is also involved in families of systems. In product-line engineering, requirements engineering concerning the whole product line should be handled separately from the individual requirements of each system [CN02]. Thus, product-line architectures considers requirements both from the point of view of product families and of single systems. Actually, engineering the common requirements defines the commonalities, while the individual requirements concentrates on variabilities. This is the explanation why requirements engineering is on one hand connected with commonality analysis and on the other hand it means very much the same as domain engineering for single systems.

2.4.3 Reuse engineering

Reuse engineering means identification of reusable parts in software and changing software to make it more reusable [Mül96]. Tasks involved in reuse engineering are object recovery, identification of abstract data types and reusable components, and building libraries of the identified components. Thus, the reuse aspect in reuse engineering is concentrated on existing software. To even more emphasize

the legacy aspect, a very close term, *reuse re-engineering* can be used for the same purpose, i.e. to extract reusable components from existing software and to collect them into repositories [CCM93].

Both of the terms (reuse engineering and reuse re-engineering) are associated with promoting reuse of existing software. Domain engineering, instead, deals both with existing software and with newly-developed software. Thus, reuse engineering can be considered as a part of domain engineering.

2.4.4 Domain-specific software architectures

Domain engineering is the process of developing and implementing a domain-specific software architecture [Tra94, Cza97]. A *domain-specific software architecture* (DSSA) is an architecture for a specific domain, however, it should still be general enough to support several applications of the domain [JGJ97]. A DSSA consists of common requirements and the process how to refine it [Tra94]. Thus, a DSSA denotes very much the same as a product-line architecture. (See also the next subsection about the relationship between DSSAs and reference architectures.)

2.4.5 Reference architectures

A *reference architecture* is a software architecture for a family of application systems [Tra94]. The result of refining or instantiating a reference architecture is an *application architecture*. Thus, a reference architecture corresponds to a common core architecture and an application architecture corresponds to the architecture of an individual application in product-line terminology.

When comparing DSSAs and reference architectures, DSSAs actually concentrate on the process how to provide the common and variable features and how to derive the final architectures for each application. Reference architectures and application architectures, instead, denote actual architectures or artifacts. However, DSSAs and reference architectures have a close connection with each other. A DSSA is a process to develop a reference architecture in order to generate applications within a particular domain [Tra94].

2.5 Domain engineering methods

This subsection introduces some of the most famous domain engineering methods. Most of the methods have originally concentrated on purely domain engineering. Later they have been extended or connected to other methods to cover the whole product-line engineering process.

FODA (Feature-Oriented Domain Analysis) [KCH+90] considers the features of similar applications in a domain. Features are capabilities of the applications considered from the end-user's point of view. Features cover both common and variable aspects among related systems. They are divided into mandatory (or common), alternative, and optional features. FODA includes different analyses such as requirements analysis and feature analysis. It produces a domain model covering the differences between related applications. FODA is currently a part of the model-based approach of domain engineering [Sof00a]. This approach covers both domain engineering and application engineering. The domain engineering part consists of domain analysis (FODA), domain design, and domain implementation. In addition, FODA is further extended into FORM (Feature-Oriented Reuse Method) [KKLL99] to include also software design and implementation phases. FORM covers both analysis of domain features and using these features to develop reusable domain artifacts.

ODM (Organization Domain Modeling) [SCK+96] mainly concentrates on the domain engineering of legacy systems. However, it can be applied to the requirements for new systems. ODM is tailorable and configurable, and it can be integrated with other software engineering technologies. It combines different artifacts such as requirements, design, code, and processes from several legacy systems into reusable common assets. ODM is supported by DAGAR (Domain Architecture-based Generation for Ada Reuse) [KS96]. DAGAR process does not cover domain modeling. Thus, it applies ODM or other methods for this purpose. Instead, DAGAR process includes activities both for domain engineering and application engineering.

RSEB (Reuse-Driven Software Engineering Business) [JGJ97] is a systematic, model-driven reuse method. It composes sets of related applications from sets of reusable components. RSEB uses UML [RJB99] to specify application systems, reusable component systems, and layered architectures. Variabilities between systems are expressed with variation points and attached variants. FeatuRSEB (Featured RSEB) [GFd98] connects features (from FODA) with RSEB. Actually, FODA and RSEB have much in common. Both of them are model-driven methods providing several models corresponding the different view points of the domain.

Thus, they are compatible with each other.

PuLSE (Product Line Software Engineering) [BDF+99] divides product-line life cycle into three parts. In the initialization phase, PuLSE is customized to fit the particular application. Adaptation is affected by the nature of the domain, the project structure, the organizational context, and the reuse aims. In the second phase, product-line infrastructure is constructed. This step includes scoping, modeling, and architecting the product line. In the third phase, the product-line infrastructure is used to create individual products. This concerns instantiating the product-line model and architecture. Each of these phases is associated with product-line infrastructure evolution. Each phase should consider changing requirements and changing concepts within the domain. PuLSE has several components. PuLSE-DSSA (PuLSE - Domain-Specific Software Architecture) [ABFG00], for example, develops a domain specific architecture based on the product-line model. As other examples, PuLSE-Eco concentrates on economic scoping, and PuLSE-EM on evolution and management.

FAST (Family-Oriented Abstraction, Specification, and Translation) process covers the whole product-line engineering process [WL99]. However, it divides domain engineering into two parts: domain analysis and domain implementation. Thus, the issues involving domain design are considered in the domain analysis phase. FAST provides systematic guidance to each step during product-line engineering. These steps can be proceeded as transitions between states. FAST also describes which artifacts are produced in each phase.

2.6 Domain engineering and OOA/D methods

Object-oriented analysis and design (OOA/D) methods are widely used in software engineering. However, they mainly concentrate on single system analysis and design. Object-orientation was first thought to inherently bring reuse, however, nowadays it has been agreed that to enable reuse, it has to be invested and planned beforehand.

This subsection first considers the shortcomings of existing OOA/D methods and then introduces such OOA/D methods that support domain engineering.

2.6.1 Shortcomings of existing OOA/D methods

Reuse is connected to object-orientation at least in the form of frameworks and design patterns. A framework comprises an abstract design which several applications can be instantiated from. A design pattern, in turn, provides a solution (design) for recurring problems [GHJV95]. However, OOA/Ds do not support reuse, and they do not guide in designing families of applications. In [CE00], the problems OOA/D methods have been listed as follows:

- no distinction between domain engineering and application engineering
- no domain scoping phase
- no differentiation between modeling variability within one application and between several applications
- no implementation-independent means of variability modeling.

To solve the reuse problem in OOA/Ds, there are several approaches [CE00]. The first alternative is to upgrade older domain engineering methods. For example, the structured analysis and design used in FODA [KCH+90] can be replaced with OOA/D methods [VAM+98]. Second, customizable domain engineering methods can be specialized into object-oriented direction. For example, in ODM [SCK+96], a system engineering method can be given as a parameter [Sim97a]. Third, existing OOA/D methods can be extended with the concepts of domain engineering. For example, RSEB [JGJ97] is based on object-oriented modeling notation UML [RJB99] and OOSE [JCJO92], to which it adds reuse aspect. The fourth alternative is to integrate existing methods. For example, FeatuRSEB [GFd98] is developed from FODA [KCH+90] and RSEB [JGJ97], the former applying domain engineering and the latter object-oriented concepts.

2.6.2 OOA/D methods supporting domain engineering

OOram is an OOA/D method that also emphasizes domain engineering [RWL96]. OOram does not believe in a single methodology, instead, it is a generic method that provides a framework for creating a variety of methodologies. Real software systems are typically very large and complex. Thus, OOram collects objects having a common goal to the same group called *collaboration*. In addition, different viewpoints are provided to understand the system more easily.

As conventional OOA/D concentrates on class dimension, OOram emphasizes role model dimension. *Role model* collects collaborating objects together according to their common goal. The goal defines an *area of concern*. A *role* describes

an object and its responsibility to achieve the goal in the area of concern. Role models can be hierarchically related to each other as a base role model and a derived role model. Collaborations are connected to frameworks and design patterns. A framework design can be represented as a composition of collaborations. In addition, collaborations can be instances of design patterns.

OOram identifies the following three processes to achieve the given goal. First, the *model creation process* concentrates on creation a model for a certain phenomenon. Examples are creating a role model, performing role model synthesis, and creating object specifications. Second, the *system development process* covers the typical software life cycle, from the specification of the user's needs to the installation and maintenance of the system. Third, the *reusable asset building process* is needed in continuous production of several closely related systems. Creating a system is mainly configuring and reusing robust and proven components. To complete the system, adding a few new components may be necessary.

Besides OOram, JODA (JIAWG Object-Oriented Domain Analysis) [Hol93] — where JIAWG stands for Joint Integrated Avionics Working Group — integrates domain analysis with Object Oriented Analysis (OOA) [CY91]. Domain analysis provides a domain model to be used in producing reusable elements especially reusable requirements, while OOA brings the object analysis techniques and notations. According to [Hol93], objects are more stable than functions. Although the operationality of an object may change, the object itself more often stays. This is the reason why OOA/D methods are preferred to functional methods.

JODA divides domain analysis process into three phases: domain preparation, domain definition, and domain modeling. Domain modeling is extended from OOA, and it consists of three steps. First, object life-histories and state-event response are examined. This step is derived directly from OOA, and it comprises identification of objects, definition of attributes and services, and finding relationships between objects. Second, domain scenarios are identified, defined, and simulated. In this phase, the behaviour of objects is identified to provide services to users and other systems. Third, objects are abstracted and grouped to enable reuse. For example, if there are repeating instances of similar problems such as the processing of many message formats, the problems are analyzed to find an abstraction to cover all cases. In addition, this phase identifies commonalities and variabilities such as stable interfaces with variable implementations. For example, interfaces of device drivers for different graphical packages can be similar, although the implementations vary according to the capabilities of the device. The three steps are iterated and the resulting domain model is reviewed and updated as long as the model is accurate enough to define the domain.

As a third example of methods connecting object orientation and domain engineering, FeatuRSEB [GFd98] is considered. As mentioned in Subchapters 2.5 and 2.6.1, FeatuRSEB is developed from FODA [KCH+90] and RSEB [JGJ97]. FeatuRSEB connects use cases of RSEB and features of FODA to each other. The use case model is user-oriented, while the feature model is reuser-oriented. The former provides a description of what systems in the domain do. The latter, instead, shows which functionality can be selected when engineering new systems in the domain.

FeatuRSEB analyzes several exemplar systems, finds out their commonalities and variabilities, and produces a feature model. The feature model construction process can be divided into four steps. First, individual exemplar use case models are merged into a domain use case model (or family use case model). The differencies are expressed by using variation points. The model is formed such that the original exemplars can be traced. Second, an initial feature model is created. Functional features are derived from the domain use case model. Mandatory and optional features are identified according their frequency of occurence in exemplar systems. Identified features are decomposed into subfeatures. The domain use case model may contain variation points. The features corresponding such variation points are divided into subfeatures. Derived subfeatures are traced back to their occurencies in the domain use case model. The third step adds architectural features to the feature model. Instead of specific function, architectural features relate to system structure and configuration. Finally, the fourth steps adds implementation features to the feature model.

3 Emerging problems

As a conclusion, this section collects the emerged problems concerning domain engineering. However, the problems listed below are not necessarily reseach problems:

Non-established terminology
: is mainly due to the immaturity of the research area (considered throughout the report).

Incompleteness of domain engineering
: means that domain modeling can never be complete, instead, the model can always be made more accurate. In addition, domain knowledge from different sources can be inconsistent with each other (considered in Subsection 2.3.3).

Distributed domains
: are typically associated with domain engineering of existing applications, i.e. reengineering aspects (considered in Subsections 2.1.2 and 2.1.3).

Overlapping domains
: are due to the difficulties in determining the borders of domains or in dividing domains into subdomains (considered in Subsection 2.1.1).

Lacking support for domain engineering in well-accepted object modeling techniques
: is due to the immaturity of the research area of domain engineering and to the belief that object orientation inherently makes applications reusable (considered in Subsection 2.6).

The last problem is probably the most severe one of the above problems. In addition to object modeling techniques, the similar deficience concerns the modeling language UML. It does not support domain engineering, neither. However, the technique concerning MDA (Model Driven Architecture) that is based on UML, may provide support for domain engineering, too [dMJS02, Arc01]. Thus, that topic is an important subject for further research.

References

[ABC+97] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie-Mellon University, 1997.

[ABFG00] Michalis Anastasopoulos, Joachim Bayer, Oliver Flege, and Christina Gacek. A process for product line architecture creation and evaluation, PuLSE-DSSA–version 2.0. Technical Report IESE-038.00/E, Fraunhofer Institut Experimentelles Software Engineering, June 2000.

[ADH+00] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software — Practice and Experience*, 30(7):825–847, 2000.

[Arc01] Architecture Board ORMSC. *Model Driven Architecture (MDA)*, July 2001. Document number ormsc/2001-07-01.

[BDF+99] Joachim Bayer, Jean-Marc DeBaud, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, and Tanya Widen. PuLSE: A methodology to develop software product lines. In *Symposium on Software Reusability (SSR'99)*, pages 122–131, May 1999.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[CCM93] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Working Conference on Reverse Engineering*, pages 73–82, Baltimore, Maryland, May 1993.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice-Hall, 1991.

[Cza97] Krzysztof Czarnecki. Leveraging reuse through domain-specific software architectures. In *Workshop on Institutionalizing Software Reuse (WISR'8)*, Columbus, Ohio, March 1997. Position paper.

[dMJS02] Miguel de Miguel, Jean Jourdan, and Serge Salicki. Practical experiences in the application of MDA. In *Fifth International Conference on the Unified Modeling Language — the Language and its applications (UML 2002)*, volume 2460 of *Lecture Notes in Computer Science*, pages 128–139, Dresden, Germany, Septermber–October 2002. Springer.

[DT97] Merlin Dorfman and Richard H. Thayer, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Fifth International Conference on Software Reuse (ICSR'98)*, pages 76–85, June 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. O. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hol93] Robert Holibaugh. Joint integrated avionics working group (JIAWG) object-oriented domain analysis method (JODA). Technical Report CMU/SEI-92-SR-3, Software Engineering Institute, Carnegie-Mellon University, November 1993.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[KKLL99] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, and Kwandoo Lee. Feature-oriented engineering of PBX software for adaptability and reuseability. *Software — Practice and Experience*, 29(10):875–896, 1999.

[Kru95] Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KS96] Carol Diane Klingler and James Solderitsch. DAGAR: A process for domain architecture definition and asset implementation. In *ACM TriAda'96 Conference*, December 1996. Available from: http://source.asset.com/stars/darpa/Papers/ArchPapers.html.

[Lah02] Essi Lahtinen. Scenario-based assessment of software architectures. Master's thesis, Tampere University of Technology, Department of Information Technology, April 2002.

[LM97] W. Lam and J. A. McDermid. A summary of domain analysis experience by way of heuristics. *Software Engineering Notes*, 22(3):54–64, 1997.

[Mül96] Hausi A. Müller. Understanding software systems using reverse engineering technologies research and practice (tutorial). In *18th International Conference on Software Engineering (ICSE-18)*, Berlin, Germany, March 1996. Available from: http://www.rigi.csc.uvic.ca/UVicRevTut/UVicRevTut.html.

[Nei80] James M. Neighbors. *Software Construction Using Components*. PhD thesis, University of California, Irvine, Department of Information and Computer Science, 1980.

[Pou97] Jeffrey S. Poulin. Software architectures, product lines, and DSSAs: Choosing the appropriate level of abstraction. In *Workshop on Institutionalizing Software Reuse (WISR'8)*, Columbus, Ohio, March 1997. Position paper.

[Pri90] Rubén Prieto-Díaz. Domain analysis: an introduction. *Software Engineering Notes*, 15(2):47–54, 1990.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with objects: The OOram Software Engineering Method*. Manning/Prentice Hall, 1996. Available from http://www.ifi.uio.no/∼trygver/documents/book11d.pdf.

[SB98] Mark Simos and Alan F. Blackwell. Pruning the tree of trees: The evaluation of notations for domain modeling. In *10th Workshop of the Psychology of Programming Interest Group*, January 1998.

[Sch00] Klaus Schmid. Scoping software product lines. In Patrick Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 513–532. Kluwer Academic Publisher, 2000.

[SCK+96] Mark Simos, Dick Creps, Carol Klingler, Larry Levine, and Dean Allemang. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, June 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Sim91] Mark A. Simos. The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse. In Rubén Prieto-Díaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 204–221. IEEE Computer Society Press, 1991.

[Sim97a] M. Simos. Organization domain modeling and OO analysis and design: Distinctions, integration, new directions. In *STJA'97 Conference Proceedings*, pages 126–132, Erfurt, Germany, 1997.

[Sim97b] Mark A. Simos. Lateral domains: Beyond product-line thinking. In *Workshop on Institutionalizing Software Reuse (WISR'8)*, Columbus, Ohio, March 1997. Position paper.

[Sof00a] Software Engineering Institute, Carnegie-Mellon University. *Domain Engineering: A Model-Based Approach*, January 2000. Available from: http://www.sei.cmu.edu/domain-engineering/.

[Sof00b] Software Engineering Institute, Carnegie-Mellon University. *Domain Engineering and Domain Analysis*, September 2000. Available from: http://www.sei.cmu.edu/str/descriptions/deda-body.html.

[TP00] Steffen Thiel and Fabio Peruzzi. Starting a product line approach for an envisioned market. In Patric Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 495–512. Kluwer Academic Publishers, 2000.

[Tra94] Will Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *Software Engineering Notes*, 19(2):52–56, 1994.

[VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro. FODAcom: An experience with domain analysis in the Italian telecom industry. In *The Fifth International Conference on Software Reuse*, pages 166–175, Victoria, Canada, June 1998.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[WS94] Grant R. Wickman and James Solderitsch. A systematic software reuse program based on an architecture-centric domain analysis. In *Software Technology Conference*, Salt Lake City, Utah, April 1994. Available from: http://www.asset.com/stars/lmtds/Papers/ReusePapers.html.