A Concrete Method for Developing and Applying Product Line Architectures

Michalis Anastasopoulos, Colin Atkinson, and Dirk Muthig

Fraunhofer Institute Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{anastaso, atkinson, muthig}@iese.fhg.de

Abstract. Software development organizations are often deterred from introducing product line architectures by the lack of simple, ready-to-use methods for developing and applying them. The well-known, published product-line-engineering methods tend to focus on the early stages of the software life cycle and address product line issues at a high-level of abstraction. Connecting product-line concepts with established implementation technologies is thus largely left to the user.
This paper introduces a method, known as the KobrA method, which addresses this problem by enabling product line concerns to be coupled with regular (non-product line) architectural artifacts, and thus introduced incrementally. By explaining how the method can be understood as a concrete instantiation of the well-established PuLSE-DSSA product-line architecture approach, the paper clarifies the product line features of the KobrA method and illustrates how they can be used in tandem with established, general-purpose product line methods.

1 Introduction

Software development organizations are becoming increasingly aware of the benefits of product line engineering in general, and product-line architectures in particular. However, they are often deterred from exploiting these ideas in practice due the lack of concrete, ready-to-use approaches for applying them. Well known methods for product-line engineering, such as FAST [1], FODA [2], PuLSE [3], and Bosch [4], cover many of the central issues involved in supporting families of systems, but they all tend to address them at a high-level of abstraction, and focus on the earlier phases of the life-cycle. They also typically require some degree of tailoring before they are ready for use in a specific project. This makes the approaches difficult to apply effectively in practice, and limits their usability to experienced engineers.
The KobrA method [5], recently developed at the Fraunhofer Institute for Experimental Software Engineering, was designed to address this problem. It provides an approach that adds product line concerns to the regular technical description of software architectures, and is thus essentially ready to use "out of the box". It also avoids the quantum paradigm shift associated with most methods by enabling product line capabilities to be adopted incrementally. The method therefore offers a simple and practical way of migrating to and leveraging product line architectures.

Extending a normal (non-product-line) software architecture to cover a family of systems requires product line concerns to be integrated with the core concepts, techniques and processes used to describe the architecture of a single member of the family. In the case of the KobrA method, the core architectural concepts are based on the principles of component-based and model-driven development. The former focuses on components as the primary vehicle for decomposing and organizing architectural information, while the latter emphasizes the creation of platform-independent models as a way of shielding architectural assets from the idiosyncrasies and volatility of implementation platforms. KobrA therefore integrates the ideas of product line engineering with the principles of component-based development and model driven architectures. The product line engineering aspects of the method are adapted from the PuLSE approach, especially from PuLSE-DSSA the PuLSE component for developing product-line architectures, since this is one of the most mature and complete methods for product line engineering. In fact, the KobrA method can be viewed as a direct and concrete instance of PuLSE-DSSA.

In this paper we explain how KobrA's approach to product-line architecture creation and application relate to those of PuLSE-DSSA, and how model-driven (i.e. UML-based) architectural artifacts can be augmented by descriptions of common and variable features. This has two basic goals. The first is to provide a clear presentation of how the KobrA method can be viewed as an instance of the PuLSE-DSSA approach, and thus how it captures all the important issued raised by product-line architectures. The second is to explain how KobrA's concrete and pragmatic solution for product line architecture documentation complements the higher-level approaches of previous methods, and thus how it can be used together with them. Among other things, this clarifies how the KobrA method can be used with the other parts of the PuLSE approach.

The next section of the paper presents the two methods that PuLSE-DSSA and KobrA and motivates their combination. Section 3 then describes the combined, concrete architectural artifacts and processes. Section 4 gives an example application of the method before section 5 concludes with some final remarks.

2 Background

As a basis for the discussion, in this section we provide a brief overview of the two main ingredients of the paper: PuLSE-DSSA's abstract approach for the development and evaluation of product-line architectures and the (non-product line) architectural concepts of the KobrA method. In section 2.3, we then discuss the benefits of combining these two approaches into a concrete method for applying product line architectures.

2.1 Creation and Evaluation of Product Line Architectures: PuLSE DSSA

PuLSE™ (Product Line Software Engineering)[1] is a framework for developing and managing software product lines [3], which has been developed at the Fraunhofer Institute Experimental Software Engineering (IESE). An overview of the PuLSE framework is given in Figure 1.

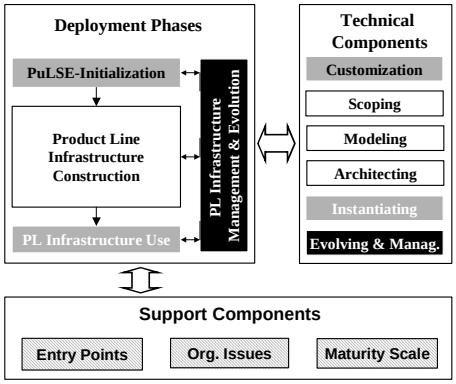


Figure 1: Overview of the PuLSE framework

PuLSE is composed of three main parts: the deployment phases, the technical components, and the support components. The deployment phases describe the life-cycle stages of a software product line infrastructure (i.e. its initialization, its construction, its use, and its evolution), the technical components provide the technologies needed to operationalize the deployment phases, and the support components capture the experience with the PuLSE approach in the context of different organizational structures, different mature organizations, and different initial contexts (or entry points).
PuLSE-DSSA (Domain-Specific Software Architecture) is one of PuLSE's technical components that provide a general approach for creating architectures for software product lines [6]. The general DSSA approach is customizable with respect to the following factors: the mechanisms that ensure traceability from features in the scope definition down to elements of the product line architecture, the process and techniques used to integrate existing components, the supported architectural viewpoints with respect to the application domain, and the role that prototyping plays throughout the architecture creation process.

The process covers the scope definition and the domain model including a decision model that describes the dependencies and inter-relationships among variabilities in the domain model. Both products are the output of other technical PuLSE

[1] PuLSE is a registered trademark of Fraunhofer IESE.
For more information see www.iese.fhg.de/PuLSE

components, that is, the scoping (Eco) and the modeling (Customizable Domain Analysis) component. **Figure 2** depicts the general DSSA process.

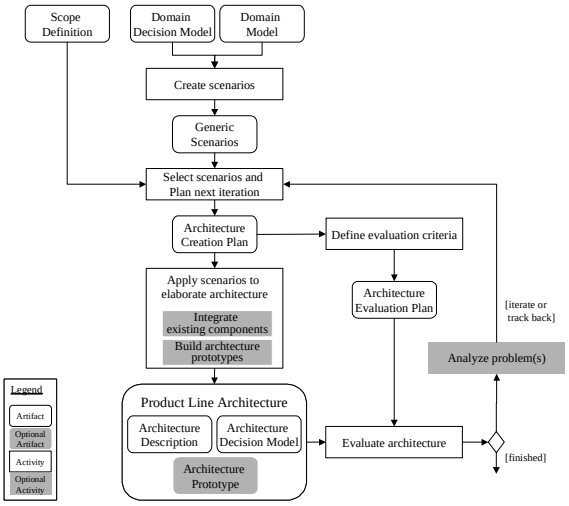


Figure 2 Process for creating product line architectures defined by PuLSE-DSSA

The set of generic scenarios that must be supported by the product line architecture is derived from the domain model. The grouping and prioritization of these scenarios is documented in an architecture creation plan, enabling the architecture to be developed incrementally group-by-group, scenario-by-scenario. After each group the intermediate architecture is evaluated and in the event of problems the process is reiterated before it continues with the integration of further scenarios.
The creation of the product line architecture is accompanied by continuous quality assurance activities. The architecture is evaluated according to an evaluation plan, which may be supported by the development of prototypes, especially in relation to the non-functional requirements.

2.2 Component-based Product Line Engineering with the UML: KobrA

The KobrA method's approach to the visualization and description of software architectures integrates two increasingly important paradigms: component-based development and model-driven development. Both are expected to play a central role in the future of the software industry.
Component-based development (CBD) is motivated by the desire of software organizations to avoid reinventing the wheel every time a new system is developed, but to package commonly occurring software into reusable building blocks that can be simply and straightforwardly integrated into new applications [7]. Although simple in principle, however, achieving a truly component-based approach to development has proven difficult in practice. One of the problems has been to extend the concept of

components beyond the implementation and deployment phases of development into the earlier, analysis and design-oriented phases.
Model-driven development is motivated by the desire of software organizations to shield their key software assets from the constant change in implementation technologies and platforms by capturing them in the form of platform independent models. When applied to architecture level artifacts this leads to the high-profile Model Driven Architecture (MDA) approach of the OMG [8]. Although not strictly tied to any particular notation, the MDA approach is oriented towards the Unified Modeling Language (UML) [9]. Successful application of the MDA approach requires more than just the creation of UML models, however. It also requires an underlying architectural strategy for structuring and organizing the models.
KobrA integrates the CBD and MDA approaches by using models as a way of capturing components in the early phases of development, and using components as a way of organizing and structuring the models within a software architecture. Treating all behavior-rich abstractions as components, regardless of their size or location, and modeling them in the same way, in turn, achieves this. The resulting synergy addresses the weakness of each paradigm individually.

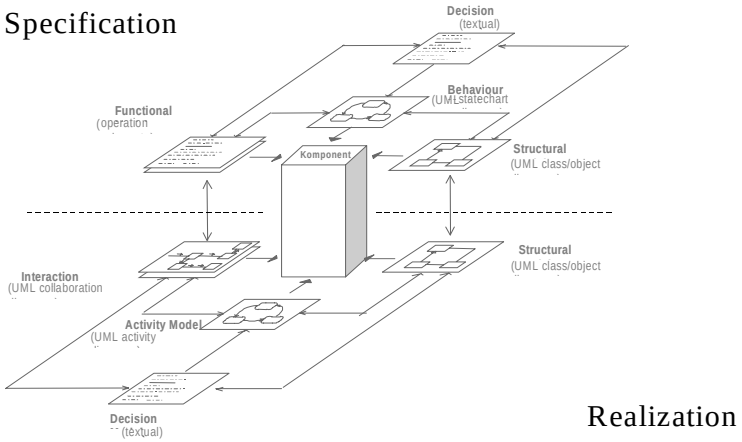


Figure 3 Model-based Representation of a Component

Figure 3 illustrates the general way in which a logical component is modeled in KobrA. This description of a component consists of two basic parts: the specification, which describes the externally visible properties of the component, and the realization, which describes how these properties are realized.
Three primary models make up the specification: the structural model, which describes the information needed to use a component in terms of class and object diagrams, the behavioral model, which describes the externally visible states of the components in terms of a state chart diagram, and the functional model which describes the effects of the component's operations in terms of textual pre and post conditions (often written in OCL). Together these provide different views onto what may be viewed as the interface of the component.

Three primary models also make up the realization: the structural model, which is a refinement of the specification structural model, describes all types and structural information related to how the component works, the interaction model which describes how instances of the component interact with other component instances (or objects) in order to realize the component's operations, and the activity model which describes the algorithms used to implement the operations in an alternative flow-chart oriented style. Together these provide different views onto what might be called the design or architecture of the component.
This basic modeling strategy is applied to all behavior-rich abstractions, whether they be the system as a whole or smaller abstractions that deal with only a part of the overall functionality. As illustrated in Figure 4, the descriptions of these separate components are organized in a tree-like structure reflecting the logical nesting or containment of components. Thus, its parent logically contains the component.

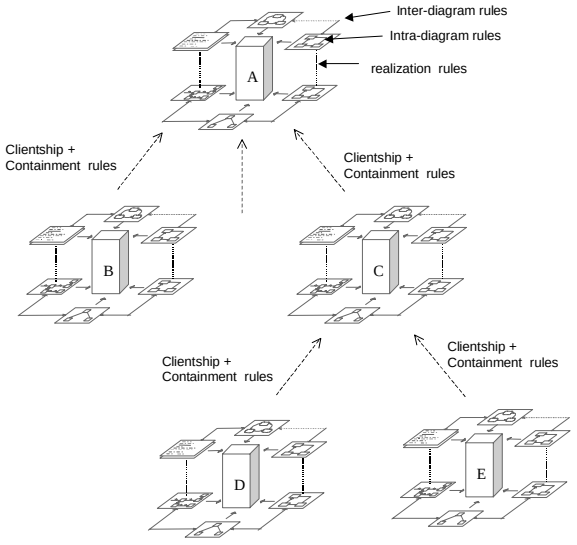


Figure 4 Component Containment Tree

Figure 4 illustrates some important aspects of KobrA's representation of a software architecture. First, and foremost, it conveys the principle that every diagram is related to just one component. This reflects the so-called "principle of locality" in KobrA, which requires that each diagram describe one and only one component. An immediate consequence of this principle is that no single component "knows" about the whole system. Even the component at the root of the tree, which represents the system to be developed, only knows about the logical properties of its subcomponents. How these are internally realized is described by their private realization models, which are not visible externally. In principle, therefore, a complete view of the overall architecture can only be attained by walking through the tree and

collecting the architectural information from each component realization.
This approach works because very strict consistency rules are defined within and between the models documenting components. The main categories of rules are depicted on Figure 4. As well as guiding the development process these consistency relationships are very valuable during maintenance activities for tracing requirements and realization decisions between components.
Although it may at first seem unfamiliar, this way of documenting a system's architecture actually subsumes most traditional views of software architectures. Although it is conceptually treated in the same way as other components, the realization of the "system" component at the root of the containment tree resembles what most people would traditionally view as an architecture [10], [11]. It basically describes the nature of the subsystems (i.e. subcomponents) making up the system, and the nature of the relationships and interactions between them. Applying this concept recursively is a natural (and often implied) consequence of the subsystem concept.
Organizing specification and realization information in this way also resolves the ubiquitous problem of understanding the role of analysis and design activities in a component-based development. Analysis of a component takes place in the context of the realizations of the clients (e.g. the super component) which use it, and design of a component takes place in the context of the specification it is supposed to be realizing. Defining analysis and design in a context-sensitive way not only clarifies the difference between them, but also provides a natural and effective strategy for their incremental development.
This uniform approach also removes the often-fuzzy boundary between architecture and design. In KobrA, architectures and designs are conceptually the same thing, although it is still possible to distinguish them based on the granularity of the entities they embrace. However, the usual distinction based on abstraction level is removed.
As well as being simple and conceptually clean, another important benefit of a nested, tree like system architecture of this form is that it opens the way for a recursive process based on the principle of step-wise refinement. Thus, the KobrA process (for single system development) is based on the recursive applications of a few core activities (e.g. specification, realization etc.). This top down approach is balanced by a bottom up approach that allows prefabricated (e.g. COTS) components to be inserted into a component tree based on the nature of the specification that it supports.

2.3 Benefits of combining PuLSE-DSSA and the KobrA Method

In this paper, PuLSE-DSSA and the KobrA method are combined to form a concrete method for applying and developing product line architectures. In this section, the benefits of this combination with respect to each of the individual approaches are discussed.
Table 1 gives an overview on the strengths and weaknesses of PuLSE-DSSA and the KobrA method, respectively. This overview shows that the strengths of the one approach repair the weaknesses of the other.

Table 1 Strengths and weaknesses of PuLSE-DSSA and KobrA

	Strengths	Weaknesses
PuLSE-DSSA	▪ Focused on product line issues ▪ Closely integrated with PuLSE, that is, with non-architectural product-line activities (e.g. scoping, domain analysis) ▪ Customizable wrt. application domain and application context	▪ Abstract process description ▪ Abstract artifact definitions
KobrA	▪ Systematic method for component-based software development ▪ Well-defined processes, artifacts, and consistency rules among these artifacts ▪ Supports technology transfer by adding product line concepts to single-system practices ▪ Compatible and linked with all major implementation technologies	▪ Processes focus primarily on (generic) component frameworks and only secondarily on product line architectures

More concretely, PuLSE-DSSA, on the one hand, fully focuses on product line issues and thus especially tackles product line issues concerned in the context of product line architectures and is also closely connected to other, non-architectural product line activities, such as scoping and domain analysis. PuLSE-DSSA is a generally applicable approach because it is customizable to different application domains and application contexts. Hence, it describes processes and artifacts at a general, abstract level only and thus cannot immediately deployed by an organization.
On the other hand, the KobrA method provides precisely defined processes and artifacts for developing generic component frameworks that describe product line architectures. The product line concepts are designed as an add-on to a single-system component-based development approach so that the migration of organization towards a product line approach are well supported. Additionally, the KobrA method is integrated and compatible with all major implementation technologies, which makes it quickly usable and deployable in practical contexts. Hence, the core of the KobrA method is a component framework whose development is driven by the development of a component hierarchy. In other words, architectural concerns are not the main drivers of the framework development.
Combined PuLSE-DSSA and the KobrA method provide a concrete method for applying and developing product line architectures, which is not as widely applicable as PuLSE-DSSA but the usage of the UML and the component focus make it usable in many industrial contexts.

3 The architecture creation process

In this section, KobrA's way of developing an architecture for a family of systems is presented as an instance of the abstract process framework described by PuLSE-

DSSA. The intention is to provide a simple and comprehensive overview on the different process steps.

3.1 Create Scenarios

Input	▪ Domain decision model ▪ Generic work products (e.g. functional, non-functional requirements, use-cases, feature models)
Output	▪ Generic scenarios in the form of activity models

The process begins with the creation of scenarios that refine the functional and non-functional requirements from the domain analysis phase. Other domain analysis work products (e.g. feature models) are considered as well.

3.1.1 Functional Requirements

Functional requirements are often derived from the business processes to be supported by a system. They typically relate user actions to desired system responses. In the scenario creation step we describe these business processes detailed and completely. For this purpose, activity models with swim lanes are used because they provide a standardized way for concretely describing the workflow of business processes. Swim lanes are used to assign particular organizational units or roles involved in a workflow to activities. Alternatively or additionally, interaction scripts can be used as a textual artifact [12]. The scripts are written in form of tables where the columns are the textual equivalents of the swim lanes in activity diagrams. The rows provide typically one-sentence descriptions of the actor and system activities.

In the first set of activity models created we typically have two main roles: a user and the system. All activities in the user's swim lane identify the interface that the system must support. The activities in the system's swim lane identify internal operations and components required by the system.

3.1.2 Non-functional Requirements

Non-functional requirements play an important role and may even be the dominant drivers of the architecture creation process [13]. In this case, the architects deal with non-functional properties of the system first by applying appropriate architectural approaches. In this way architectural fragments are defined that are then matched with the functional requirements.

For concretizing the non-functional requirements in form of scenarios, the stimulus/response scheme is used as described in [13][14]. This scheme suggests that each non-functional requirement is described in terms of an abstract scenario, a set of stimuli, and the desired responses. Thereby, the scenario describes the required system property (e.g. "The system must recover from a server failure"), the stimuli precisely describe the events that map to the scenario (e.g. "The server fails"), and the responses are the concrete measures (e.g. 99% availability). The definition of the responses can be deferred until step 3.3, which describes the evaluation criteria.

3.2 Select Scenarios and Plan Next Iteration

Inputs	▪ Generic Scenarios ▪ Scope Definition
Outputs	▪ Architecture Creation Plan

When the scenarios were created, they are prioritized and grouped to define the architecture creation plan, which defines the order of processing the groups with respect to their priority.

Prioritizing scenarios should follow a simple, basic rule: the bigger the impact of a scenario on the system architecture, the higher the scenario's priority. However simple in theory, evaluating the architectural impact of a scenario from its description is a difficult, non-trivial task that requires much experience. Based on implicit experience, scenarios may be weighted through expert voting. Voting can be supported or replaced by specific evaluation criteria.

Below, we list some criteria that can be used for prioritizing scenarios and constructing the architecture creation plan. These criteria can be used as an indication of the expected architectural impact of scenarios:

1. *Economic Value*: This is the value from an economic point of view that will be added to the product-line when a scenario will be realized. The product line scope can deliver this information. It is clear that a high economic value is a hint for high scenario priority.
2. *Typicality – Criticality*: A scenario is of high typicality when it reflects routine operations whereas critical scenarios occur in a rather sporadic manner and when the user does not expect them to occur. Typical scenarios should therefore implemented first.
3. *Future-proof*: A scenario is future-proof when it considers possible evolution points. Future-proof scenarios are likely to have a big impact on the architecture and therefore they should assigned high priorities.
4. *Effort*: This is an estimation of the effort required for realizing a scenario. Since the architecture creation plan also assigns responsibilities to personnel, the effort estimation is surely helpful.
5. *Non-functional requirements* suggest the use of well-known architectural approaches. The architectural impact of a scenario that has non-functional attributes can be therefore high.

The number of variation points and pre-conditions in a scenario can be additionally considered when the above criteria do not produce a sufficient ranking.

After the prioritization of the scenarios, the groups and the order in which they will be processed throughout the architecture creation is derived. Additionally, a useful set of architectural views (and work products) is defined according to the importance of relevant architectural concerns. At the end of this process step, it has been defined how many iterations the architecture creation process will have, in which iteration which group of scenarios is processed and which views are created.

3.3 Define Evaluation Criteria

Inputs	▪ Architecture creation plan ▪ Quality scenarios
Outputs	▪ Architecture evaluation plan consisting of: ▪ Consistency Rules ▪ Quality assurance documentation

In this step, the criteria to be met by the architecture are defined. The basic criterion requires the support of all scenarios in all variants. Additionally, the architect defines further necessary criteria. Here is a list of categories of additional criteria that may be considered:

1. *Responses*: Here we can add the desired responses to the stimuli of the quality requirements.
2. *Style-specific rules*: In case that styles or patterns are used we must make sure that the according assumptions and rules are met.
3. *Metrics*: Metrics like coupling, cohesion etc. may be defined here.

3.4 Apply Scenarios

Inputs	▪ Architecture creation plan
Outputs	▪ Product Line Architecture consisting of: ▪ Architecture description as structural, activity and interaction models ▪ Architecture decision model ▪ Prototype implementation (optional)

Applying the scenarios means executing all framework-engineering activities of the KobrA method scenario by scenario processed in the current iteration. The steps starts at the root of the component hierarchy (i.e., the context realization), goes down to the system, the first level subcomponents, and so on. The resulting abstract components can be implemented in the same iterative way or the implementation may be delayed to the last iteration of the architectural process.

3.4.1 Context Realization

Applying the group of scenarios considered during the current iteration means defining which architectural elements (e.g. components) are responsible for which activities in the scenarios. To this end a context realization for the system under development is first created. This is a class diagram showing the system as a single component along with the interface it exposes and the entities (i.e. data) that it needs in order to operate. It is important to note that the purpose of the context realization is not to identify subcomponents but only the context of the overall system.

3.4.2 Specification

Following the context realization we proceed with the specification of the system as a whole. The focus lies here in the detailed description of the interface exposed by the

system. Optionally we can model the possible states of the system. The principle of locality plays an important role at this point. It suggests that only the parts are modeled, that have an immediate bearing on the subject component. This principle is applied in the specification as well in the realization phase (see next section).

3.4.3 Realization

After having specified what the system does we must describe in more detail how the functionality is realized. For the operations specified previously we consider here what component services are needed. It is now that we identify subordinate or associated components.
The identification of necessary component services is facilitated through the creation of an interaction model consisting of collaboration or sequence diagrams. They refine the internal operations contained in the system swim lane of the activity diagram that represents the current scenario.

The additional component services identified here lead to a new class diagram that shows the associations between the new components and the system component. This class diagram realizes the context of the new components and therefore is the starting point for their specification and realization. The architect may now deal with each of the newly added components in the same way he did with the system component. This may result to several recursions until all components are fully described. For that reason the architect may decide to just identify a component and defer handling in detail with it.

3.4.4 Implementation

Implementation is an optional step and is applied when prototypes of the architecture are required. While the architectural descriptions produced in the previous steps are meant to be technology-independent, the goal of the implementation step is first to create an architecture description that is bound to an implementation technology. To that end we first decide how the logical KobrA components will be packaged physically in the target technology. Then we map the logical components onto their implementation counterparts. This produces implementation class diagrams that can be next translated into code.

3.5 Evaluate Architecture

Inputs	▪ Product Line Architecture ▪ Architecture Evaluation Plan
Outputs	▪ Architecture Evaluation Results

The evaluation of the product line architecture applies the evaluation plan (see section 3.3) to the models documenting the architecture. To minimize the effort involved in the evaluation, only the functionality newly added in the last iteration should be evaluated in detail. Nevertheless, it is essential to ensure that all scenarios from previous iterations are still supported.

In a product line context the need for this kind of regression evaluation also exists after two other activities in the product line life-cycle: first, during framework maintenance after the product line infrastructure has been extended due to a new variability, and second during application engineering when system-specific functionality has been integrated with product line infrastructure artifacts which have been already evaluated during framework engineering.

3.6 Analyze Problem

Precondition	At least one of the evaluation criteria was not fulfilled.
Inputs	Architecture Evaluation Results
Outputs	▪ Problem analysis ▪ Assignment of action items.

If any problems are identified during the evaluation of the architecture these are analyzed to find a solution. The analysis includes an in-depth reasoning about the chosen architectural approaches and decisions.

In the best case, changing just the last iteration may solve the problem. In the worst case, if a solution supporting all scenarios in all variants even exists it is necessary to track back to the first iteration.

4 A concrete example

In this section we will illustrate the process described above in terms of an example taken from messaging domain of mobile phones. The goal of the example is to show how an application is designed that enables the user to send and receive messages from mobile phones. Due to space constraints we will omit details and focus on the main issues.

4.1 Create Scenarios

The messaging domain encompasses several use cases; the example will take two of the main use-cases into account: s*end message* and *show message*. Both consist of a main success scenario and a number of extensions. The first main success scenario that we encounter is the scenario for sending a message. It is described by the activity diagram shown in Figure 5. It becomes clear that the send message activity can be split in three lower-level activities, namely to *composing*, *submitting*, and *saving* the message.

The extensions attached to this scenario deal with features such as object insertion, system reaction when there is an incoming call during message composition etc. These are also described by activity diagrams.

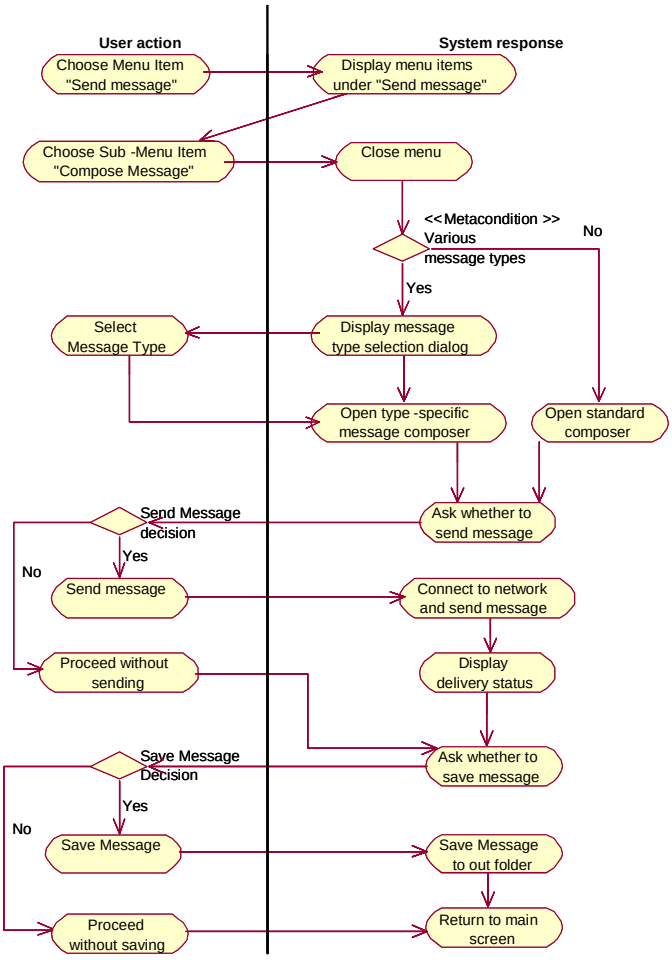


Figure 5 Send Message Scenario

4.2 Select scenarios and plan next iteration

The scenario selection process has to prioritize the following scenarios (listed in alphabetical order):

- Acknowledgment failure
- Handling of attachments
- Handling of special messages
- Incoming call during message editing
- Memory failure
- Message sending failure
- Network failure
- Object insertion
- Send Message Main Success Scenario
- Show Message Main Success Scenario
- T9 functionality (dictionary suggestions)
- Unrecognized symbol entered

The *send-message* and *show-message* main success scenarios have the highest economic value, represent the most typical use-cases, and are future-proof. The *send-message* scenario has been selected to be processed during the first iteration because

- More effort was required (i.e. message composition requires more effort than message display)
- It has a series of non-functional requirements that apply to it (high usability of composer, low failure-rates when submitting a message, message submission procedure should not exceed 2 seconds)
- It has no preconditions, and
- It has five variation points as opposed to four of the show message scenario

4.3 Define Evaluation Criteria

Next to the main criterion, which in this case relates to supporting all variants (mobile phones with and without different message types) we take the following additional criteria into account:

- The send message scenario suggests the use of a client/server approach for submitting messages. The specifics of this approach (e.g. low coupling between client and server) were therefore one evaluation criterion.
- A further criterion refers to separating the user interaction from the business logic.

4.4 Apply Scenarios

4.4.1 Context Realization

Figure 6 illustrates the context realization class diagram for the send message scenario. *GoPhone* represents the system component as a whole along with the operations it exposes, which were extracted from the activity diagram in Figure 5. The subject stereotype relates to the principle of locality and declares that *GoPhone* is the subject of the diagram. *MessageCenter* represents the external server component that

receives and forwards messages. The rest of the classes in the picture are entities needed by the system.

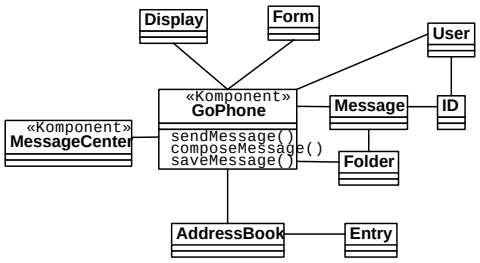


Figure 6 Context Realization of the System component

4.4.2 Specification

The following diagram is based on the context realization. It refines the entities by removing the *User* entity and by making the *ID* entity an attribute of *Message*.

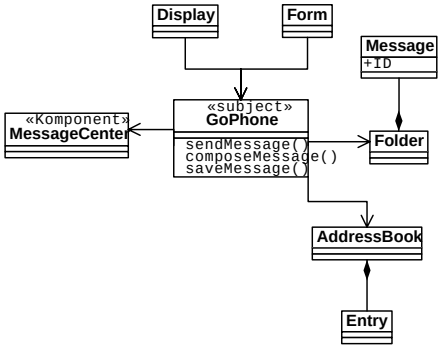


Figure 7 System component specification class diagram

After the specification class diagram we specify in a detailed way the operations exposed by the *GoPhone* component. The following table gives an example of such an operation specification.

Table 2 *composeMessage()* operation specification

Name	composeMessage
Description	Creates the form for composing messages and starts the composition.
Receives	User ID
Returns	Message Return Code

Changes	new Message
Result	A new message is composed.

4.4.3 Realization

Figure 8 shows now the realization of the system component. The Form entity from the specification has now been further analyzed. Additionally the *SMSClient* component was assigned the responsibility of connecting to the network and submitting messages while *MessageComposer* takes care of the details for the message composing.

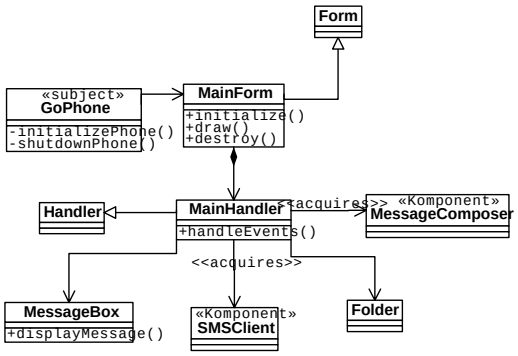


Figure 8 System component realization class diagram

The above diagram can be now seen as the context realization for the *SMSClient* and *MessageComposer* components. These components can now be specified and realized following the same process. Figure 9 and Figure 10 shows how this can be accomplished for the *MessageComposer*.

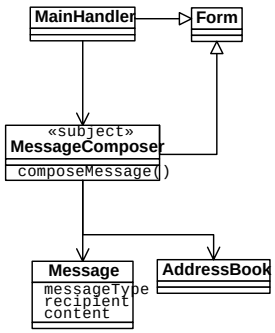


Figure 9 MessageComposer specification class diagram

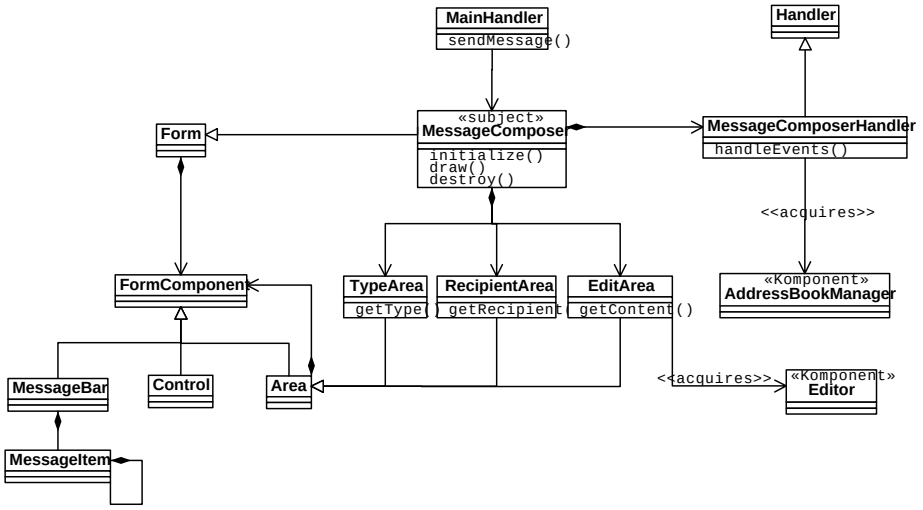


Figure 10 MessageComposer realization class diagram

4.4.4 Implementation

For creating a prototype of the architecture, which in the first iteration deals with sending a message, we selected the Java Micro Edition technology and especially the Mobile Information Device Profile [15]. The following table shows how logical components can be mapped onto MIDP classes.

Table 3 Mapping between KobrA and MIDP

KobrA	MIDP	Reasoning
GoPhone		This component should be implemented as a native device-dependent component since it deals with the initialization of the phone
MainForm	1. MIDlet 2. Screen	Entry point for the Messaging application. It will initialize the application and show the initial Screen.
Message Composer	General-purpose Java class	Message composition refers to the workflow: "Message Type Selection" → "Recipient Selection" → "Message Editing". Hence it makes sense to have different screens for each of these actions. That means one List for selecting the message type, one *TextBox* for entering the recipient and one *TextBox* for entering the message content. A general-purpose class is therefore assigned the management of this workflow.

5 Conclusions

In this paper we have presented how the general PuLSE-DSSA approach towards applying product line architectures can be combined with the systematic and concrete method, known as KobrA. The resulting method can be viewed as a concrete instantiation of the PuLSE-DSSA approach, serving to operationalize PuLSE-DSSA for use "out-of-the-box" without the need for further instantiation. However, the combination is more than an ordinary concrete instantiation of PuLSE-DSSA because it leverages component-based and model-driven development concepts. Thus, taken as a whole, the combined approach essentially integrates the product-line engineering (PLE), component-based development (CBD) and model-driven architecture (MDA) paradigms.

The key to this synergy is to reflect each "paradigm" as a different dimension of concern. Consideration of components and their nesting relates to the CBD dimension, consideration of models and their mapping into executable forms relates to the MDA dimension, and concerns for generic artifacts and their instantiation into specific forms relates to the PLE dimension.

References

[1] Weiss, D. M. and Lai, C. T. R., Software Product Line Engineering: A family Based Software Engineering Process, Addison-Wesley, 1999.

[2] K.Kang, S.Cohen, J.Hess, W.Nowak and S.Peterson, Feature-oriented Domain Analysis (FODA), Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990

[3] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid. K., Widen, T, and Debaud, J.-M., *PuLSE: A Methodology to develop Software Product Lines*, Proceedings of the Symposium on Software Reuse(SSR'99), May 1999

[4] Bosch, J. *Design and Use of Software Architectures.* Addison-Wesley, 2000.

[5] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel. Component-Based Product-Line Engineering with UML. Addison-Wesley, 2001

[6] M. Anastasopoulos, J. Bayer, O. Flege, and C. Gacek. *A Process for Product Line Architecture Creation and Evaluation: PuLSE-DSSA V2.0*, Technical Report, Fraunhofer IESE, 2000

[7] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley 1999.

[8] *Executive Overview: Model Driven Architecture*, Object Management Group, 2001 http://www.omg.org/mda

[9] *Unified Modeling Language Specification*, Version 1.4, Object Management Group, 2000.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, July 1996.

[11] M. Shaw, and D. Garlan, *Software Architecture: Perspective on an Emerging Discipline*, Prentice-Hall, 1996
[12] John Carroll, Making Use, Scenario-Based Design of Human-Computer Interactions, Cambridge: MIT Press, 2000
[13] Len Bass, Mark Klein, Felix Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method." 4th Conference on Product Family Engineering. Bilbao, Spain, October 2001.
[14] Rick Kazman, Mark Klein, Paul Clements, ATAM: Method for Architecture Evaluation, Technical Report, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA ,August 2000
[15] Java 2 Micro Edition, Mobile Information Device Profile, http://java.sun.com/products/midp/, Sun Microsystems 2002
[16] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, Fabio Peruzzi, The Architecture Based Design Method, Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January 2000