Component-Based Product Line Development: The KobrA Approach

Colin Atkinson, Joachim Bayer, and Dirk Muthig

Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6, D-67661 Kaiserslautern, Germany, +49 (0) 6301 707 211, {atkinson, bayer, muthig}@iese.fhg.de

Keywords: Product Lines, Components, Component Based Software Engineering, Frameworks

Abstract: The product line and component-based approaches to software engineering both hold the potential to significantly increase the level of reuse in industrial software development and maintenance. They also have complementary strengths, since they address the problem of reuse at opposite ends of the granularity spectrum - product line development essentially supports "reuse in the large" while component based development supports "reuse in the small". This paper describes a method, KobrA, which cleanly integrates the two paradigms into a systematic, unified approach to software development and maintenance. Key synergies resulting from this integration include support for the rapid and flexible instantiation of system variants, and the provision of methodological support for component-based framework development.

1. INTRODUCTION

The potential advantages of a product line approach to the development, maintenance and deployment of software have long been recognized. Instead of continually "reinventing the wheel", or incorporating parts of "old" systems in an ad hoc manner, organizations following a product line approach can consolidate their key software assets within a high-quality, reusable software core, and concentrate their resources on adapting this core to meet the changing needs of customers.

Despite its promise, however, systematic product line development still remains the exception rather than the rule, and its potential remains largely unfulfilled. A major reason is that traditional software implementation technologies do not really provide the mechanisms needed to support the rapid

and cost-effective adaptation of implemented code in a way required by a genuine product line approach. As a result, existing product line approaches have been forced to concentrate on the earlier activities in the software life cycle, and thus often appear to developers to be somewhat divorced from the "real" business of coding. There is also an erroneous perception that product line development is incompatible with "regular" development of single systems. Many developers therefore feel the adoption of a product line approach would force them to discard their current "single system" practices with which they feel comfortable.

The advent of component-based software engineering changes this situation by making available mechanisms that enable software elements, right down to the binary level, to be rapidly and efficiently assembled into new applications. This allows the basic tenet of product line development to be applied at all phases and levels of software development, and to work with software in all its different forms, including binary forms. The ability of component-based software engineering to support the interoperation of binary code modules is one of the key characteristics that distinguishes it from the object-oriented paradigm (upon which it based). Components, therefore, provide the perfect foundation for the practical application of product line development.

The benefits are not just one way, however. Component-based software engineering also stands to gain significantly from product line ideas. It is to be expected that component-based systems within a given domain, or created by a given organization, will share many similarities, and in particular, will use many of the same components. The variabilities between systems in a family will thus likely revolve around a relatively small number of critical components. Therefore, rather than assemble every system in the family from scratch, it makes sense to build so-called "frameworks", which "hard-wire" the common aspects of the family, and allow the variable components to be "plugged in" as and when needed. Although the value of such frameworks has been recognized for some time, however, their creation and maintenance is still something of a "black art", lacking concrete methodological support. The techniques and ideas of product line development are the ideal foundation for the provision of such methodological support.

In short, the product-line and component-based approaches to software development seem to have complementary strengths. They both represent powerful techniques for supporting reuse, but essentially at the opposite ends of the granularity spectrum. Components provide a technology for "reuse in the small" while product line development represents an approach for "reuse in the large". Therefore, significant benefits can be expected from their integration. This paper introduces the KobrA method, which attempts to create a natural synergy between the component-based and product line approaches to software development. In essence, component technologies provide the flexible and rapid configurability needed for genuine product line development,

while the disciplines of the product line approach provide the methodological foundation needed for the development and deployment of sound component-based frameworks.

The remainder of this paper is structured as follows. In the next section we describe the product-line approach that provides the foundation for the product line aspects of the KobrA method. The main part of the paper then describes how product line development is supported in KobrA, starting with an overview of the method's main features, and then continuing with a description of the central "framework engineering" and "application engineering" activities. The paper concludes with a description of related work, and a discussion of future plans.

2. PULSE

Product line software engineering aims at creating generic software assets that are reusable across a family of target products. PuLSE™ (Product Line Software Engineering)[1] is a method for enabling the conception and deployment of software product lines in a large variety of enterprise contexts [1].

The life cycle of a software product line in PuLSE is split into the following phases: initialization, product line infrastructure construction, usage, and evolution. PuLSE provides technical components for the different deployment phases that contain the technical know how needed to operationalize the product line development. The technical components are customizable to the respective context. Customization of PuLSE to the context where it will be applied ensures that the process and products are appropriate. In the initialization phase, the other phases and the technical components are tailored. Through this tailoring of the technical com-ponents, a customized version of the construction, usage, and evolution phases of PuLSE is created.

The principle dimensions of customization are the nature of the application domain, the organizational context, reuse aims and practices, as well as the project structure and available resources.

PuLSE has been applied successfully in various different contexts for different purposes. Among other things it has proved helpful for introducing sound documentation and development techniques into existing development practices. However, in circumstances where there were no pre-existing processes or well-defined products, the introduction of PuLSE turned out to be problematic. In such cases, the "customization" of PuLSE was actually more concerned with the introduction of basic software engineering processes than with the adaptation of the product line ideas to existing processes. Especially

1. PuLSE is a registered trademark of the Fraunhofer IESE.

in immature environments, the effort for this process definition can be considerable and even prohibitive.

From the perspective of the PuLSE method, therefore, there is much to be gained by the definition of a "ready-to-use" customization of the method that already contains the required software development processes that may be missing in immature organizations. KobrA can be viewed as such as "ready-to-use" customization of PuLSE.

3. THE KOBRA METHOD

The KobrA method[2] represents a synthesis of several advanced software engineering technologies, including product line development, component-based software development, frameworks, architecture-centric inspections, quality modeling, and process modeling. These have been integrated in KobrA with the basic goal of providing a systematic approach to the development of high-quality, component-based application frameworks. Numerous methods claim to support component-based software development, but these invariably tend to be rather vague and unprescriptive in nature. They define a lot of possibilities, but provide little, if any, help in resolving the resulting choices between them. KobrA, in contrast, aims at being as concrete and prescriptive as possible.

A fundamental tenet of KobrA is the strict distinction of products and processes. The products of a KobrA project (e.g., models, documents, code modules, test cases, etc.) are defined independently of, and prior to, the processes by which they are created, and effectively represent the goals of these processes. Furthermore, all products are organized around, and oriented towards, the description of individual components. This means that, as far as possible, there are no global or system-wide products - all products (and accompanying processes) are defined to carry information only related to their particular component. The advantage is that components (and the products that describe them) can then easily be separated from the environment in which they were developed and therefore can be reused independently.

From a product line perspective, KobrA represents an object-oriented customization of the PuLSE method. The infrastructure construction phase of PuLSE corresponds to KobrA's framework engineering activity, the infrastructure usage phase of PuLSE corresponds to KobrA's application engineering activity, and the product line evolution phase of PuLSE corresponds to the maintenance of the frameworks and applications. Figure1 illustrates the overall relationship of these activities in KobrA.

---

2. The KobrA project is funded by the German Government and is being undertaken by a consortium of four organizations: Softlab GmbH, Munich, Psipenta GmbH, Berlin, GMD-FIRST, Berlin and Fraunhofer IESE, Kaiserslautern.

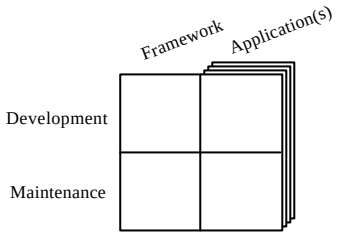


Figure 1. KobrA Product-Line Activities

The purpose of the framework engineering activity is to create, and later maintain, a generic framework that embodies all product variants that make up the family, including information about their common and disjoint features. The purpose of the application engineering activity is to instantiate this framework to create particular variants in the product family, each tailored to meet the specific needs of different customers, and later to maintain these concrete variants. A given framework can therefore be instantiated multiple times to yield multiple applications.

It is important to note that the distinction between the framework activities in KobrA is the level of generality/specificity, not the level of detail. In fact, the framework and application engineering activities both result in descriptions of components in terms of a mixture of textual and UML-based (graphical) models. The difference between the two is that the framework models potentially contain variabilities, while the application models do not. The advantage of using the UML is that frameworks and associated application are independent of any particular programming language or component technology (e.g., Java Beans, COM, CORBA).

The transformation of an application into an executable form is carried out in a distinct set of activities that are essentially orthogonal to the framework and application engineering activities. The implementation activity takes instantiated UML models and maps them, through a series of well-defined refinement and translation steps into an executable representation (e.g., high-level source code) [2]. Finally, the build activity actually creates binary load modules ready for deployment in the target environment.

The following sections describe the product-line aspects of KobrA in more detail. In particular Section3.1 describes the framework engineering activity, while Section3.2 concentrates on the application engineering activity. In both cases we use a running example to visualize the key elements of the method. The case study is a library system framework, based on a domain defined by several library systems located in Kaiserslautern: the local city library, the university library, and our institute library.

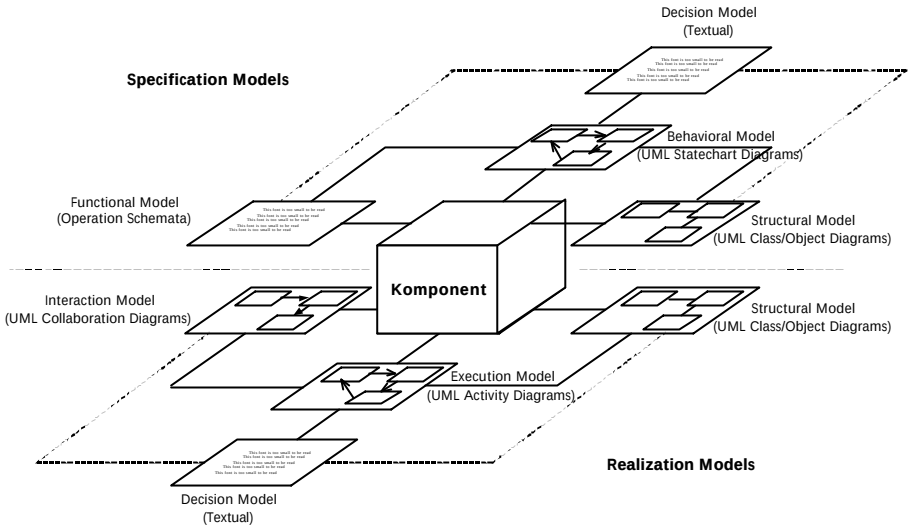


Figure 2. Komponent Specification and Realization

3.1 Framework Engineering

In KobrA, a framework is the static representation of a set of Komponents[3] organized in the form of a tree. Each Komponent is described at two levels of abstraction - a specification, which defines the Komponent's externally visible properties and behaviors, and thus serves to capture the contract that the Komponent fulfils, and a realization, which describes how the Komponent fulfils this contract in terms of contracts with lower level Komponents. A framework, therefore, is a tightly coupled arrangement of Komponent specifications and realizations. Figure2 shows the general set of UML models, which make up Komponent specifications and realizations.

To start the framework development process, the context of the Komponent at the root of the tree is modeled. Since this takes the form of a realization it is known as the context realization. Subkomponents are then identified, their specifications derived from the context realization models, and finally the subkomponents realizations are designed. This is performed recursively until no further subkomponents are required.

The framework is a reuse infrastructure for creating systems within the application domain. The family aspects are captured by decision models, which, as illustrated in Figure4(b), are a part of all specifications and realizations. The decisions relate to variabilities in the domain that are explicitly reflected in the models of the generic framework.

3. In KobrA we use the term “Komponent” as shorthand for “KobrA component”

3.1.1 Context Realization

Framework engineering starts with the elicitation of the environment properties for the planned system family, including the determination of the framework's scope. The underlying elicitation process and the used workproducts depend on the domain of interest and the project context, as described in [3]. However, the application of KobrA requires a particular set of models at the end of context realization, which is needed to begin the recursive KobrA development process. These models correspond to the models used for realizing Komponents.

The table on the left side in Figure4(a) gives an overview of the high-level business processes supported by the library framework case study. Some of the processes are optional, for example, not all libraries allow customers to reserve items. Activity diagrams are used to model each process in more detail. The activity diagram for the business process "item Check In" is shown on the right side of Figure4(a). The black conditions represent conditions that capture variabilities in the modeled process. Each variability is related to at least one decision in the library system decision model. Parts of the model are shown in Figure 4. Each decision provides a set of possible resolutions and lists for each resolution, which diagrams must be tailored in what way to represent the specified member of the system family.

Further activity diagrams can be used to refine non-trivial activities. For example, for the collection of loan costs it is necessary to specify whether customers must pay cash, whether they can use their credit card, or whether they can accept the costs but pay later, etc. When the refinement has been finished, subkomponents are identified that, together, provide the remaining activities as operations. Figure5 shows the class diagram of the library system context. A library system is composed of five white-box Komponents - Komponents whose interface is part of the external interface of their parents (here: the library system).

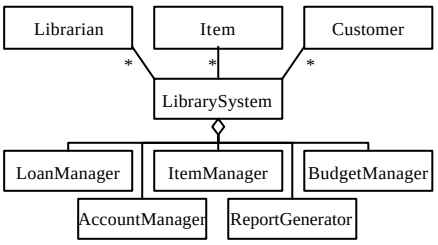
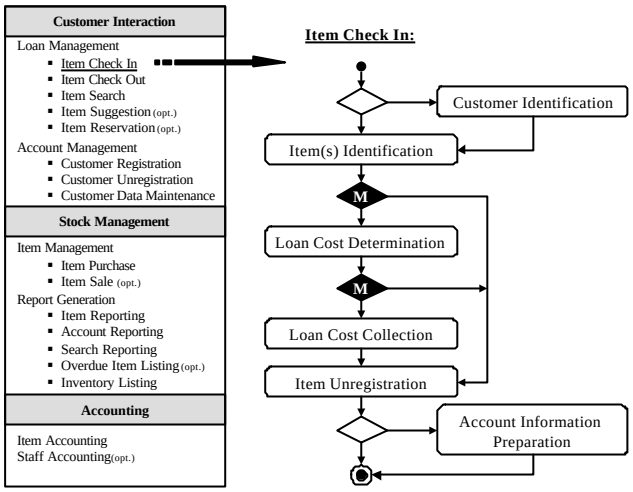


Figure 5. Context Class Diagram for Library Systems



(a) Library System Business Processess

Id	Customer Interaction	Resolution	Diagram	Effect
1	Does the loan of items cost anything?	y	Activity Diagram	Activity „Loan Cost Determination“ is present
			„Item Check In“	
		n	Activity Diagram	Activities „Loan Cost Determination“ and „Loan Cost
			„Item Check In“	Collection“ are not present.
2	What is the type of payment?	per item	Activity Diagram	Activity „Loan Cost Collection“ is present
			„Item Check In“	
		member-ship fee	Activity Diagram	Activity „Loan Cost Collection“ is not present.
			„Item Check In“	

(b) Library System Decision Model

Figure 4. Library System Example: Business Processes and Decision Model

3.1.2 Komponent Specification

The goal of Komponent Specification is to create a set of models that collectively describe the externally visible properties of a Komponent. As such, the specification can be viewed as defining the interface of a Komponent and describing the services a Komponent provides to its parent. The specification of a Komponent is comprised of four main models: the structural model, the behavioral model, the functional model, and the decision model. The structural, behavioral and functional models constitute the specification models for a Komponent as it is used in all applications covered by the framework. The

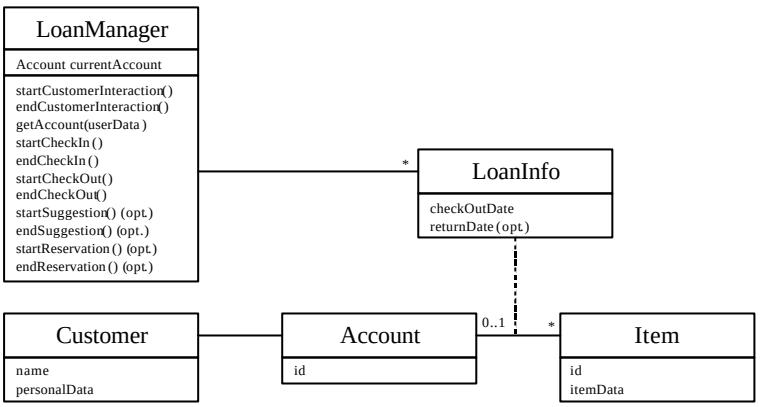


Figure 6. Specification Class Diagram for **LoanManager**

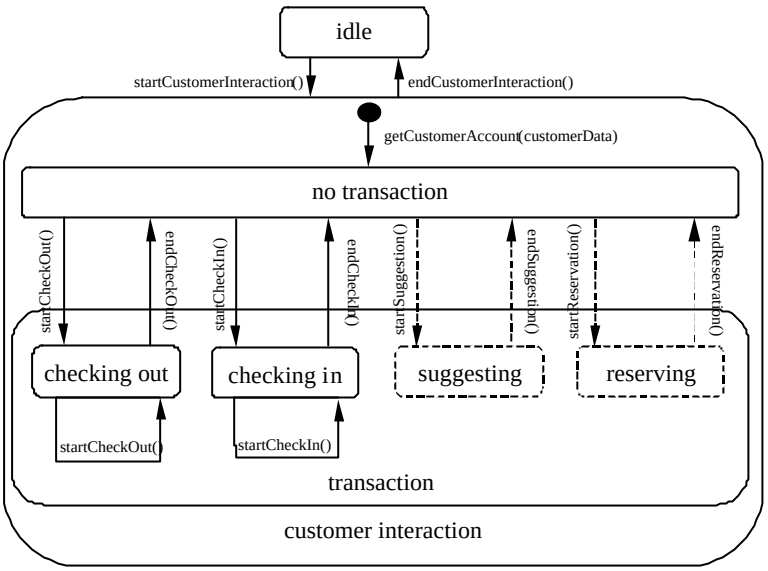
decision model contains information about how the models change for the different applications.

The structural model describes the classes and relationships by which a Komponent interacts with its environment, as well as any internal structure of the Komponent, which is visible at its interface. The structural model is composed of UML class diagrams and UML object diagrams. Class diagrams define the classes, attributes, and relationships that describe the externally visible types characterizing the Komponent's relationship to its environment. Object diagrams are only needed if the Komponent under specification contains white box components. If this is the case, the purpose of the object diagrams is to describe the parts of the internal structure that are externally visible.

Figure6 shows the specification class diagram for **LoanManager**, one of the white box Komponents of the library system identified in the context realization class diagram. The purpose of the specification class diagram is to show the externally visible properties and expectations of the Komponent under consideration. Since this **LoanManager** is focus of this class diagram, only its operations are shown. There are some optional operations (e.g., **startSuggestion** and **endSuggestion** that are taken out of the class diagram if the system does not provide user suggestions). The information about which operations are needed for which systems is captured in the decision model.

A Komponent's decision model describes the different variants of the Komponent. It is an extension of the decision model of the Komponent's parent. Variabilities that arise during the Komponent specification are investigated, and a determination made about whether variabilities can be captured by already existing decisions or if new decisions have to be added to the decision model.

The behavioral model describes how a Komponent reacts in response to external stimuli. It consists of an arbitrary number of UML statechart diagrams and an optional event map. A statechart in a KobrA specification



(a) Statechart Diagram

Operation:	startCheckIn
Description:	All returned items are identified, removed from the account, registered as present in the stock, loan costs are determined, and collected.
Reads:	supplied itemData of returned items, currentAccount
Changes:	currentAccount
Sends:	ItemManager.return(list of returned Items)
Assumes:	customer interaction is open but no transaction is active.
Result:	Returned items that were registered for currentAccount are back in stock and have been removed from currentAccount. Loan costs have been determined and collected.

(b) Operation Schema **startCheckIn**

Figure 7. Library System Example: Statechart Diagram and Operation Schema

describes user visible states of a Komponent and state changes that are reactions on user visible events. Events represent requests for the execution of an operation. The operations are exactly the operations given in the specification class diagram of the respective Komponent. Event maps capture the event-operation mapping.

Figure7(a) shows a statechart diagram for the **LoanManager** Komponent, which describes the state transitions that take place for the different customer interactions. The optional parts are depicted with dashed lines. For example, the possibilities to suggest or reserve items are not needed in all library systems.

The functional model of a KobrA Komponent describes the externally visible effects of the operations that are provided by that Komponent. It consists of a set of operation schemata (operation schemata are not part of the UML, but have their origin in Fusion [4]). Each operation listed in the class diagram must have a corresponding operation schema which defines its effects in terms of input parameters, changed variables, output values (reads, changes, and sends clauses), as well as pre- and post conditions (assumes and result clauses).

Figure7(b) shows the operation schema for the operation **startCheckIn**. The variable parts of operation schemata are placed in boxes. If a system is used in library that is free, loan costs do not have to be determined. On the other hand if a library customer has to pay, the costs may be a membership fee, in which case the loan costs are not collected during check in. If the costs are determined per item the costs are collected during check in.

3.1.3 Komponent Realization

The goal of Komponent Realization is to create a set of models that collectively describe the private design of a Komponent. As with all design, the basic requirement is that the realization must realize the Komponent's specification. A Komponent's realization is comprised of four main models: the interaction model, the structural model, the activity model, and the decision model.

Interaction models define how groups of objects interact at run-time to realize Komponent operations. A UML interaction diagram (either a UML collaboration diagram or a UML sequence diagram) describes each operation that is part of the specification. The operation schemata from the Komponent specification provide most of the information needed to develop the interaction diagrams. The basic requirement is that the corresponding interaction diagram must realize all the effects defined in the result clause of a schema. In particular, whenever an object is read or changed, a corresponding message is required in the interaction diagram.Figure8 shows the collaboration diagram for the operation **startCheckIn**. Variable parts of the collaboration diagram are represented with dashed lines.

Activity diagrams can be used as intermediate models to bridge the step from operation schemata to interaction models. Activity models provide a process-oriented view of the realization of the Komponent operations. For each operation described by an operation schema in the specification, a UML activity diagram is created. Using activity diagrams, the activities that are necessary to perform an operation are modeled and subsequently used to create the interaction models.

The realization structural model describes the classes and relationships from which the Komponent is realized, and the architecture of the Komponent. Like the specification structural model, the realization structural model

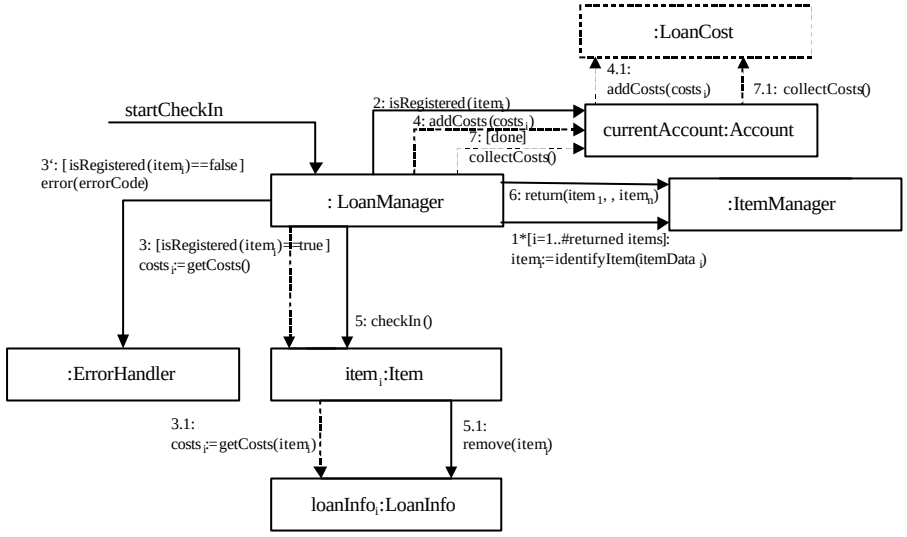


Figure 8. Collaboration Diagram **startCheckIn**

consists of a number of UML class and UML object diagrams. The realization class diagram is basically a superset and refinement of the corresponding specialization class diagram. Elements taken from the specification class diagram are described in more detail and additionally new elements (often subkomponents) uncovered during the creation of the interaction are included. In contrast to specification class diagrams, however, in realization class diagrams there are no restrictions on the inclusion of operations, or any other features, for any of the classes. Object diagrams at the realization level describe the actual instances of the elements depicted in the class diagram, and hence provide a snapshot of a typical configuration of the objects in a Komponent. They essentially capture the architecture of the Komponent, therefore.

Figure9 shows the realization class diagram for the **LoanManager** Komponent. It is a refinement of the specification class diagram shown in Figure6. The class **LoanCost**, uncovered during functional modeling (cf. Figure7(b)), is for example new. The decision model in a Komponent realization is an extension of the decision model given in the Komponent's specification.

A Komponent realization serves as the starting point for creating the next level in a Komponent framework. Based on the realization, subkomponents of the Komponent under investigation are identified. For each of the identified subkomponents, a Komponent specification is created as described in the previous section. Thus, the realization models are the primary information source for the creation of the specifications of subkomponents.

Another possible way of realizing a specification is to reuse pre-existing components such as COTS components or reengineered legacy components. To achieve this, parts of the specified interface are matched to the interface supplied by the pre-existing component. When the two interfaces are the same

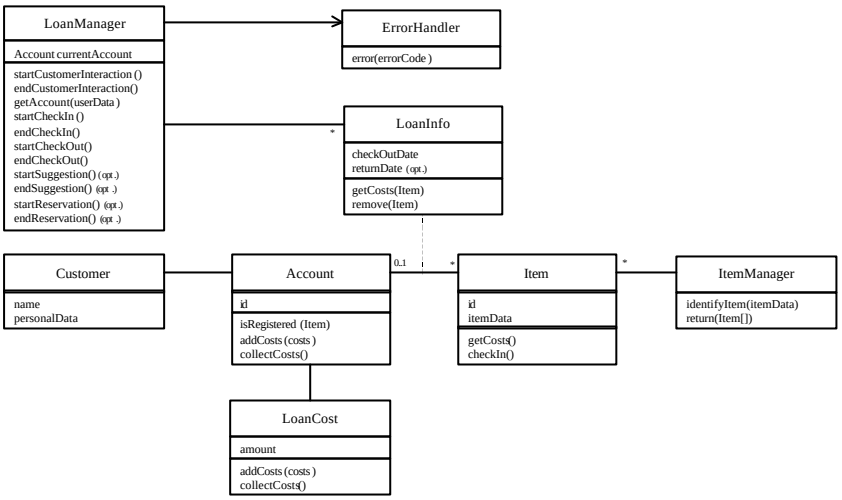


Figure 9. Realization Class Diagram for **LoanManager**

they are said to be in "mutual interface" agreement and the supplier component can be integrated in the Komponent framework. If the two interfaces are not initially the same, changes must be made to the reused component and/or the client Komponent in the framework.

3.2 Application Engineering

Application engineering uses the framework built during framework engineering to construct specific applications in the domain covered by the framework. Therefore, to be cost-effective, the benefits gained from reusing framework Komponents in the creation of several applications must be greater than the effort needed to develop the framework. In KobrA, this is achieved by assembling single products, or at least significant parts of them, from framework components. However, in order to benefit systematically from the framework, a defined method for application engineering must accompany the processes for developing the framework. This method by necessity tightly coupled with the models that are developed during framework engineering.

Figure10 gives an overview of the application engineering process. The process is centered on the given framework and driven by the framework's decision models. The framework is traversed in a top-down manner, recursively resolving decisions until all the generic framework models are transformed into specific models for the particular application.

The specific models are needed for two reasons. First, a maintainer of a particular application needs models that exactly correspond to the implementation of the maintained system. This is often not the case for the generic models of the framework. For example, in a digital library, it is possible that a

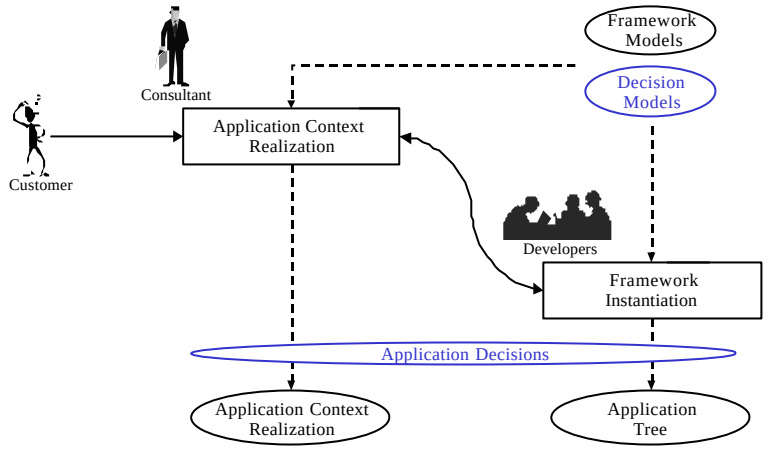


Figure 10. Overview of the Application Engineering Process

single item is registered in more than one user account. If a digital library is integrated into the framework, the relationship between the classes Account and Item will turn into a many-to-many relationship. However, such a model would cause confusion in the context of family members that deal with physical items, which are modeled in Figure 6. Second, user requirements are usually not all supported by the existing framework. If they cannot be integrated into the framework in a generic way, perhaps because there are too many conflicting constraints, they are added as customer-specific requirements to the instantiated models.

According to the common separation of requirements engineering and system design, the application engineering process is split into two primary steps: context realization instantiation and framework instantiation. The two steps are described in the following sections.

3.2.1 Application Context Realization

The instantiation of the framework's context realization is the first major activity of application engineering. It starts when the software development organization has established an initial contact to a potential customer who is interested in a software system in the domain of one of the organization's frameworks. The outputs of this process are the context decisions and a concrete realization of the application's context.

Ideally, a consultant handles interaction with the customer during this activity. The role of a consultant is played by a person who is an expert with respect to the application domain and to applications based on the existing framework. The consultant elicits the requirements for the application to be developed while working with the customer to identify problems

The elicitation process is driven by a decision sequence derived from the decision model of the framework's context realization. For example, the con-

sultant asks the customer whether future users of the library system must pay for loaning items. According to the customer's resolution, all models are changed with respect to the effect described in the decision model.

When a decision cannot be resolved directly by the customer, the resolution is supported by partially instantiated framework models, which represent the intermediate state of the application context (e.g., the activity diagram "Item Check In" without the cost collection activity).

This strategy for requirement elicitation is tightly coupled with the framework because exactly the alternatives supported by the existing framework are provided to the customer. The offering of a set of possible alternatives also simplifies the elicitation process because it corresponds to the selection of one of the provided choices.

Only when none of the supported alternatives meets the customer's needs must the required properties be explicitly modeled during requirement elicitation. The framework alternative that is the closest to the required one serves as the input for the modeling activity. Hence, the alternative not yet supported by the framework can be expressed by means of differences to requirements supported by the existing framework. This not only simplifies the later integration, either generally with the framework (i.e., a new framework revision) or specifically with a particular instance, but also guides reuse during its implementation.

When all decisions in the decision model of the framework's context realization have been resolved, the main phase of the elicitation process is finished. The result is a concrete instance containing a set of models that realize the context of the particular application to be developed. In addition to the instances of the generic framework models, customer-specific requirements that are not part of the framework can be added to extend the application context realization.

The instantiation of the generic framework context realization stops when the customer accepts the realization of the application context after checking it for completeness and correctness. The application context realization contains the requirements for the application to be developed, and the context decisions contain the choices made by the customer. They enable traceability between the realizations of the framework context and the application context. Both are passed to the developers and used during the further development of the application.

3.2.2 Framework Instantiation

The instantiation of the framework is the second major activity of application engineering. It starts when the application context realization is (partially) created and thus also the context decisions (partially) exist.

The context decisions are used to initially instantiate the generic Komponent hierarchy of the framework. This is achieved by identifying decisions at

lower levels in the Komponent hierarchy that are connected to decisions resolved during the instantiation of the framework context realization. These lower-level decisions are then resolved in accordance with the resolution of the connected context. For example, the marked sentences concerning payment in the operation schema for startCheckIn (cf. Figure 8) are removed according to the decision made by the customer.

The intermediate result is a partially instantiated Komponent hierarchy which is an application tree with unresolved points of variation, and decision models that contain the still unresolved decisions. These unresolved decisions relate either to design-related issues or user requirements that have not been handled during requirement elicitation. Both kinds of unresolved decisions are fed back to the consultant who is responsible for their resolution. The consultant resolves them either personally, together with the customer, or together with the developers. All resolutions are collected as decisions in the appropriate place in Komponent hierarchy.

In addition to the resolution of the decisions provided by the decision models of the Komponent hierarchy, customer-specific requirements must be realized and therefore integrated into either the framework or the instantiated models of the particular application. If it is expected that other customers in the future will have the same requirements, the generic integration of the realization of customer-specific requirements is the preferred alternative. The determination of whether the framework can support the new requirements must, in general, be performed by the organization.

If the new requirements are integrated into the framework, there will be a decision in the framework concerning the new requirements. The application engineering process then resolves the new decision and instantiates the new framework models so that the new requirements are part of the application tree. On the other hand, if the new requirements are not integrated into the framework, they must be modeled exclusively for the particular application in hand and integrated into the already instantiated framework models. The decision models support the integration process by indicating where in models points of variation already exist and where there are similar variants integrated or attached to the framework models.

Problems that occur during the processing of customer-specific requirements, while integrating them into the framework or into the instantiated models, may have two causes: the customer-specific requirements are either incompatible with some other requirements or with their realization in the framework. In the first case, the problem must be solved within the requirement elicitation process because this indicates an incompatibility among requirements themselves. In the other case, both the customer-specific and the incompatible requirements supported by the framework become more expensive because they have to be realized individually for the particular customer. Together with the customer the consultant must be decided whether the requirements are still to be developed as specified or whether they can be

changed with respect to framework-compatible alternatives so that they finally can be realized less expensively.

Throughout the whole instantiation of the Komponent hierarchy, consistency between adjacent layers, as well as the internal consistency of each specification and realization must be ensured. When no unresolved decision points are left, all customer-specific requirements are separately modeled and integrated and the application has successfully passed all quality assurance activities, the application engineering process is finished. The final results are the application decisions consisting of the context decisions and the Komponent hierarchy decisions, together with the application realization and the application tree.

4. RELATED WORK

As a method for industrial software development, KobrA builds upon established methods and best practice techniques. The product line aspects of KobrA are related to various other approaches. In Gomaa's work [5], object-oriented models and feature models are also used to describe product lines. The feature diagrams serve the same purpose as decision models in KobrA in that they are used to capture the variabilities within applications in the product line. However, variabilities are only captured at a rather high, problem-oriented level, and are not considered throughout the complete life cycle like decision models in KobrA.

Decision models for supporting the instantiation of domain models also appear in Lucent's product line approach [6]. KobrA's decision models have been influenced by this work, but extend it with support for higher level decisions, the capturing of relationships between decisions, and the description of variability in object-oriented models.

The most significant difference between KobrA's decision models and those of other product line approaches is their complete integration into the composition hierarchy of a framework. Every Komponent in a KobrA framework has a pair of associated decisions models (although these may be empty) - one at the specification level and one at the relaization level. These decision models therefore play a role in the definition of a Komponent's relationship (i.e. its contract) with its parent and child Komponents within the composition hierarchy. Thus, the resolution of decisions at one level almost invariably has an impact on the resolutions of decisions lower down in the hierarchy.

The component-engineering aspects of KobrA are most strongly influenced by the Fusion method [4], as adapted for use with the UML by Catalysis [7] and FuML [8]. Not only are the models used to describe components loosely based on the Fusion analysis and design models, but KobrA also adopts Fusion underlying philosophy of having models, which are well-defined, minimal and strongly related. In effect, KobrA treats a component as

Fusion would treat a system. KobrA's incremental development strategy within framework engineering is also strongly influenced by Evolutionary Fusion [9], the incremental version of Fusion.

The Cleanroom method [10] has also had a significant influence on KobrA's framework engineering approach. Although it is not an object-oriented method, Cleanroom has pioneered many of the principle and techniques required to systematically engineer high quality software systems in the context of a tree-based architecture. These principles include comprehensive built-in inspections, ubiquitous quality modeling, and a clean separation of product and process.

Finally, KobrA also has similarities with the current generation of UML-oriented development methods and process frameworks. Several of the higher-level managerial activities involved in systematic software engineering have been adapted from the Unified Process [11] where they are most highly developed. KobrA consequently shares several of the same workflows as the Unified Process, but casts them within the context of a product line rather than a cyclic waterfall derivative. Since it is described in terms of contract driven activities, the KobrA process is also compatible with the OPEN process framework [12].

5. CONCLUSION

By integrating the concepts of product-line engineering with the mechanisms of component-based development, in the way described above, the KobrA method offers an approach to software engineering that is more powerful than either paradigm individually. Product-line development benefits from the flexibility and scalability offered by components, while component based development (particularly component-based frameworks) benefits from the methodological infrastructure offered by the product line philosophy.

To provide tool support for the method, the project consortium is constructing a dedicated workbench which is designed to enable users to use the KobrA method with their own preferred suite of tools, and is thus built on the XML-based UML interchange format, XMI [13]. All XMI compliant case tools can thus be used with KobrA. In addition, the method itself is currently being validated in an industrial setting on a case study in the domain of Enterprise Resource Planning, the primary focus of the project. It is also being used in the development of the KobrA workbench. Once lessons learned from this experience are fed back into the method, and initial version of the workbench has been completed, the method will be available for use in full-scale industrial projects.

ACKNOWLEDGEMENTS

The authors are grateful to their colleagues on the KobrA method development team for their contribution to the ideas in this article: Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Barbara Paech, Tanya Widen, Peter Rösch, Jürgen Wüst, and Jörg Zettel, and to the other members of the KobrA project.

REFERENCES

[1] 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 Reusability (SSR'99), May 1999.

[2] Bunse, C., Atkinson, C. "The Normal Object Form: Bridging the Gap from Models to Code", Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, USA, October, 1999

[3] Bayer, J., Muthig, D., and Widen, T. "Customizable Domain Analysis", Proceedings of the International Symposium on Generative and Component-Based Software Engineering, September 1999.

[4] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P. Object Oriented Development: The Fusion Method. Prentice Hall International, 1994.

[5] Gomaa, H., Kerschberg, L., Sugumaran, V., Bosch, C., Tavakoli, I., and O'Hara. L. "A knowledge-based software engineering environment for reusable soft-ware requirements and architectures", Automated Software Engineering, 3(3,4), pp. 285-307, August 1996.

[6] Weiss, D. M., Lai, C. T. R. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999.

[7] D'Souza, D. and Wills, A. C., Catalysis: Objects, Frameworks, and Components in UML, Addison-Wesley, Object Technology Series, 1998.

[8] Atkinson, C., "Adapting the Fusion Process to Support the UML", Object Magazine, 1997.

[9] Cotton, T., "Evolutionary Fusion: A Customer-oriented Incremental Life-Cycle for Fusion", in Object-Oriented Development at Work: Fusion in the Real World, eds. R. Malan, R. Letsinger and D. Coleman, Prentice Hall, 1995.

[10] Mills, H., Dyer, M., Linger, R.C., "Cleanroom Software Engineering", IEEE Software, vol. 4, 1987.

[11] Jacobson, I., Rumbaugh, J. and Booch, G., The Unified Software Development Process, Addison-Wesley, Object Technology Series, 1999.

[12] Graham, I., Henderson-Sellers, B., Younessi, H., The OPEN Process Specification, Addison-Wesley, 1997.

[13] XML Metadata Interchange (XMI), version 1.1, OMG Document ad/99-10-02, October 1999.