Copyright © [2001] IEEE.
Reprinted from Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2001, pp. 212-223

This material is posted here with permission of the IEEE. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org
By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Developing and Applying Component-Based Model-Driven Architectures in KobrA

Colin Atkinson[1], Barbara Paech[1], Jens Reinhold[2], Torsten Sander[2]
[1]*Fraunhofer IESE, Kaiserslautern, D-67661 Germany*
[2]*PSIPENTA Software Systems GmbH, D-10178, Berlin*

Abstract

Component-based software engineering is widely expected to revolutionize the way in which software systems are developed and maintained. However, companies who wish to adopt the component paradigm for serious enterprise software development face serious migration obstacles due to the perceived incompatibility of components with traditional, commonly used development approaches. This perception is reinforced by contemporary methods and component technologies, which typically view components as merely "binary-level" modules with little relevance beyond the implementation and deployment phases of development. In this paper we present a method, known as KobrA, that embraces the component concept at all phases of the software life-cycle, and allows high-level components (described in the UML) to be implemented using conventional software development approaches as well as the latest component technologies (e.g. JavaBeans, CORBA, COM). The approach therefore provides a practical vehicle for applying he component paradigm within the context of a model driven architecture. After explaining the noteworthy features of the method, the paper briefly presents an example of its use in the development of an Enterprise Resource Planning System.

1. Introduction

It has long been recognized that the solution to the quality and productivity problems currently facing the software industry lies in increased software reuse. Furthermore, of the various reuse approaches currently available, component based development probably holds the potential to facilitate one of the most significant improvements in reuse levels. However, the adoption of the component-paradigm in real-world software products is being hampered by the narrow view of components currently offered by the dominant component-oriented implem- entation technologies (e.g. JavaBeans, COM and CORBA), and the leading development notations/methods (e.g. UML[1]/RUP[2]). These all essentially take the view that components are binary executables (or something close to them) and are thus of importance only in the latter stages of a software development project. In short, they take the view that components are the *result* of a development project rather than an integral part of it.

This view of components may make sense in "green field" development projects (where there is no existing software) or for small scale (sub) systems where everything can be developed from scratch (e.g. GUI subsystems), but it is not practicable for industrial scale software development. Developers in this environment invariably have to struggle with the demands of the market, legacy software systems and less than up-to-date technology (e.g. tools, languages etc.). Companies concerned with the day-to-day business of enterprise software development are therefore rarely in a position to throw away all their existing software assets and reimplement them in the latest technology of the day (e.g. component technologies). On the contrary, as with the transition to any new development paradigm, the move to component-based development will be a slow, incremental and often painful process for most companies.

The "binary module" view of components that currently dominates contemporary development approaches and component technologies is therefore doing little to help software organizations actually move to a more component-oriented way of doing things in practical, enterprise development situations. What is required, in contrast, are development approaches that support the essence of the component paradigm at a level of abstraction that is independent of specific implementation technol- ogies, and thus can be used with any of them. In short, methods are needed that make the component concept an integral part of the complete software life-cycle (including analysis and design as well as implementation) and are based on development activities that are strongly oriented around

0-7695-1345-X/01/$10.00 © 2001 IEEE

the components making up a system's architecture. Moreover, such methods must be supported by flexible development environments capable of managing the abstract descriptions of components, as well as their implementation in various technologies. This should include traditional technologies as well as "component technologies."

Modern approaches to component-based development also typically employ some kind of supporting framework to provide the context in which components are instantiated and deployed. The rationale is that many of the components within a given domain tend to be deployed in a similar way in the majority of applications, and the variations between applications are typically embodied within a critical few components. It therefore makes sense to consolidate these common components and their deployment patterns within a preconfigured, but tailorable framework. However, companies facing real world development pressures are no more capable of throwing away their entire software base to move to a framework approach than they are to a component-based approach. An incremental way of migrating to frameworks is also therefore needed to make these technologies more viable in practical development contexts.

In this paper we describe a method that aims to meet these needs. Known as KobrA, the most important single feature of the approach is the strict separation of the abstract description of components from their implementation. In other words, composition and implementation are two totally separate dimension of concern. Individual components, as well as their nesting within one another, are described at a level of abstraction akin to analysis/design in the form of UML diagrams. This abstract model driven architecture can then be mapped to various different implementation approaches depending on the prevailing forces on the development projects. By separating the description of components and their relationships from specific implementation idiosyncrasies, the approach allows an incremental and controlled migration to the component paradigm, first at the analysis/design level and later at the implementation level. The method also supports an incremental introduction to framework engineering.

The remainder of this paper is structured as follows. In section 2 we describes how UML models are used in KobrA to describe components. Section 3 then sketches the development process recommended by KobrA. In section 4 we explain the difference between framework engineering and application engineering in KobrA, while section 5 discusses how the principle of separation of concern is realized. Section 6 then discusses how KobrA was used for the development of a component-based framework for Enterprise Resource Planning (ERP) applications.

2. UML-Based Component Modeling

In view of its widespread acceptance, the obvious candidate for supporting a high-level (technology independent[1]) representation of components is the Unified Modeling language (UML). The language certainly provides a rich set of modeling concepts and diagram types intended to support the description of components and their related characteristics. However, the UML's support for components is very much based on the "binary module" concept, and the related idea that they are of significance only in the latter phases of development. The UML itself does not explicitly support a process, but this view of components is clearly implicit in the intended role of component and deployment diagrams (the so called implementation diagrams). Moreover, Rational's recommended process for applying the UML, the Rational Unified Process [2], explicitly delegates components to the final "implementation" stages of development.

As mentioned above, this limited "implementation-level" view of components complicates the task of using component-oriented ideas with other implementation technologies, and misses the opportunity to exploit the advantages of component in the earlier phases of the software life cycle. Instead of merely providing a graphical picture of the final component-based implementation of a system, what is required is a way of organizing and structuring UML development artifacts (including class diagrams and behavioral diagrams) in a component-oriented fashion. This will allow the analysis and design of a system to proceed in a more component-oriented way, and the benefits of modular organization to be exploited when attempting to reuse parts of the system for new applications.

This observation is not new. Probably the best-known approach advocating a component-oriented approach to the use of the UML is the Catalysis method [3]. However, Catalysis only follows this approach to a limited extent, since it also uses other concepts, like

[1] Here we mean implementation-level technology, such as programming language or a particular component technology

frameworks, to organize UML diagrams and defines many kinds of models (or views) that are not intended to be fundamentally component-oriented. In contrast, the KobrA approach is based on the principle that all UML artifacts used in system development should be structured and organized around the essential components in a system. This is embodied by KobrA's principle of locality which holds that, to the greatest extent possible, all development artifacts (e.g. requirements, architectures, designs etc.), should be created from the perspective of (and thus belong to) one and only one component in the system.

At some point or other every method purporting to use or support components must define what it means by this concept. The literature clearly indicates that this is no easy task [5]. The KobrA method approach to this issue is based on the simple and practical metaphor of -

components = (sub)systems.

Essentially any behavior-rich software abstraction capable of remembering some state information and/or of responding intelligently to external requests can be a component in KobrA. This includes entities ranging from entire systems to small. To make this principle work in practice it is necessary to have a uniform view of components in which they are all modeled in the same way regardless of their granularity or location. Moreover, it is also important to be able to explicitly capture the nesting of components within one another. In short, the method must adopt a fractal like view of a component-oriented system in which, to the greatest extent possible, all components at all level are treated in the same way[2].

KobrA components are also very much like subsystems in a technical sense as well. Like UML subsystems, components in KobrA exhibit the dual properties of classes and of packages. This means that they cannot only offer features and services (like a class), but they can act as containers for other entities, such as lower-level (i.e. sub) components. The difference between UML subsystems and KobrA components is that whereas the former cannot have any behavior of their own, the latter certainly can and usually do.

Viewing a component as a system also has the advantage that it is possible to leverage the substantial body of knowledge on the object-oriented modeling of software systems. As illustrated in Figure 1, KobrA's way of describing components is essentially based on the Fusion approach [4] for system description. This in turn is based on the original OMT approach which heavily influenced object-oriented development in the early 90's.

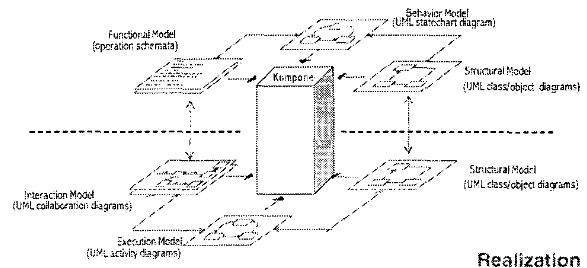

Figure 1 UML-based representation of Components

Figure 1 shows that the description of a component in KobrA is composed of two parts - a specification and a realization. These roughly correspond to the analysis and design views of a traditional non-component based system. The specification describes the externally visible properties of a component in terms of three main models - one or more static structure diagram giving the structural view, a set of operation specifications[3] giving the functional view, and a statechart diagram giving the behavioral views. These three views reflect the original three dimensions of object-oriented analysis popularized by OMT [6], and later strengthened by Fusion [4][4].

The realization, on the other hand, describes how a component realizes its specified properties in terms of interactions with other components (possibly internal subcomponents). This is achieved by means of three main models - static structure diagrams presenting the design level structural view, a set of interaction diagrams (collaboration or sequence diagrams) giving the interaction-oriented view, and a set of activity diagrams giving the algorithmic view. The realization

[2] Catalysis is another well-known method that also recognizes the importance of the fractal-like model of a component-based system.

[3] These were called operation schemata in the Fusion method

[4] In fact, the KobrA specification models represent a cross between OMT and Fusion, since the functional view essentially adopts the Fusion "operation schemata" approach while the behavioral view adopts OMT's state chart approach.

views are therefore essentially based on the Fusion concept of object-oriented design artifacts, with the addition of activity diagrams and the use of a design class diagram in place of the Fusion class descriptions.

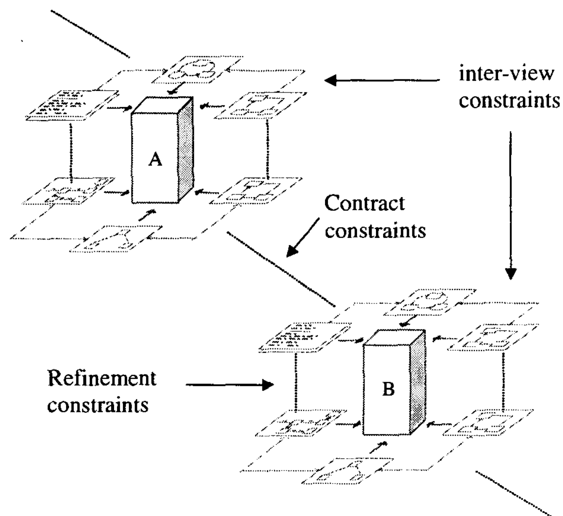

Figure 2 Inter-model constraints

One of the main motivations for basing the KobrA models on those of the Fusion method is to leverage Fusion's degree of rigor and quality control. The Fusion method is widely recognized as providing one of the most systematic approaches to object-oriented development thanks to its rigorous inter-model consistency constraints and its systematic development process. As illustrated in Figure 2, the KobrA method has adopted and extended these to provide a comprehensive set of inter-model consistency constraints. These not only serve as a means of checking the correctness of models with respect to one another, but also provide a concrete way of defining the completeness of a model. In particular, information in one model that is not used by any others is redundant and can be removed.

The main difference between KobrA and "traditional" object-oriented methods such as Fusion and OMT is that the modeling approach is applied recursively to yield a nested hierarchy of (sub)components, all described in the same way. The result is a tree of components organized according to their composition hierarchy. Naturally the inter-view and refinement constraints found in the description of a single component are now augmented by contract constraints between components as shown in Figure 2. These ensure that the properties defined within a specification of a subcomponent match the expectations of the supercomponent defined in its realization. Three main kinds of consistency rules control the form and inter-relationships of the models describing a hierarchy of components, as shown in Figure 2.

It is important to note that not all the models defined above have to be created for every component. Sometimes a model contains no useful information, and thus is not needed. This is the case, for example with very small components, or purely passive components. The position taken by KobrA is that if a model is deemed necessary it must take the form explained above, but it can be omitted. Additional artifacts not mentioned above can also be used in the documentation of components, including test cases, data dictionaries etc. One special form of auxiliary model supporting framework engineering is described below.

3. Component-Based Development

A software development method has two basic parts, a *product* and a *process*. The component-oriented model driven architecture described in the previous section represents the product of a KobrA project: in this section we describe the process.

Strictly speaking, since the nature of the KobrA product is perfectly well defined independently of a process, any process can in principle be used in its creation. As long as a set of artifacts is produced that conforms to the rules discussed above, it is not of particular importance how they come into existence. In practice, however, companies wishing to apply the KobrA approach will need the support of a process. Moreover, the simpler and more systematic the process steps can be, the easier and less error prone the application of the method will be.

3.1 Specification and Realization Activities

The goal of simplicity is primarily achieved in KobrA by the provision of a recursive development process. Since the product of a KobrA project is composed of a nested hierarchy of components, the process can be recursive. In other words, the product can be created by the recursive application of the same basic set of development activates. This not only leads to a significant simplification of the methods, since the process can be defined in a relatively small number of

activities, but it also means that the process is tightly coupled to the product. This is because activities are performed as and when required according to the shape of the product. Another powerful feature of a recursive process is therefore that it is naturally scaleable to products of different sizes.

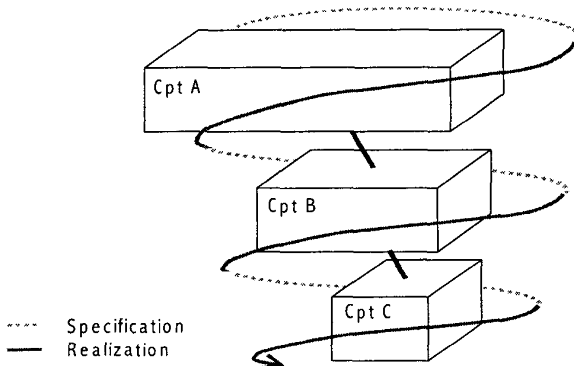

Figure 3 Recursive Development Process

As illustrated in Figure 3, the basic development process in KobrA is characterized by the recursive application of specification and realization activities, which roughly correspond to analysis and design in conventional methods. As might be expected, the purpose of the specification activity is to create a component specification, and the purpose of the realization activity is to create a component realization.

Since the nature of a component's specification is determined by the needs of its parent, the component specification activity takes places in the context of the parent component's realization models, and is driven by the corresponding "contract" between the two. Similarly, since the purpose of a component's realization is to realize its specification, the component realization activity takes place in the context of the component's specification models, and is driven by corresponding refinement rules. The overall development process is thus characterized by a repeated cycle of specification and realization activities, primarily in a top down process. The bottom up aspects of the process are discussed in the Component Reuse section below.

3.2 Context Realization

Obviously there is one fundamental question that arises with the recursive development approach just described above - where does the processes start? Clearly it must be possible to create a component specification or realization by some other means in order to start off the recursive process. In KobrA the answer is provided by the context realization activity. This is a special variant of the regular realization activity whose job is to provide the needed starting point. It outputs the same set of realization models as the regular realization activity, but does so without the benefit of a prior specification.

Since it has to start from scratch, so to speak, the context realization activity is based on "front end" requirements elicitation and analysis approaches such as uses case analysis and business process modeling. In view of this, the name of the activity "context realization" might at first seem surprising, but further thought reveals that it is in fact appropriate. This is because the models created within this activity describe the "context" of the system in the sense that it is usually understood. Included in the context realization models are a description of the components (some of which may be human) in the environment of the system, and the way in which it interacts with them. The term realization is used because the introduction of a new computer system into a business process almost always involves some changes to the process. Describing the context of the system therefore also amounts to a description of the new way in which the particular business process is to be realized once the system under development becomes available.

In a sense the context can be viewed as a pseudo component at the root of the development tree. The system can be treated as a regular component, just like any other in the KobrA development hierarchy. The context realization process starts with enterprise modeling as a substitute for the missing specification. The enterprise models are input for structural, usage and interaction modeling. Because of the complexity of the context, there are several feedback loops between these activities.

3.3 Implementation and Building

The specification and realization activities described in the previous subsections give rise to an abstract description of a tree of components. But since this mainly takes the form of a set of UML models, it is not executable. Other activities are thus needed to translate these abstract descriptions into executable forms. In KobrA this is achieved by means of the implementation and building activities.

An important principle in KobrA is that the implementation and building activities should be orthogonal to the decomposition activities captured in the UML models. In other words, as shown in Figure 4 the implementation and building activities represent a different dimension of concern to composition. This separation of concerns is the basis of KobrA's flexibility with respect to traditional implementation technologies and thus is support the model-driven architecture paradigm. The executable incarnations of components depicted on the right hand side of the figure could be implemented in almost any kind of development technology, ranging from programs in standard programming languages, to components in a modern component technology such as COM, EJB or CORBA. Of course, in the former situation more work has to be done to realize inter-component communication (e.g. using sockets) whereas in the latter case this is supported automatically.

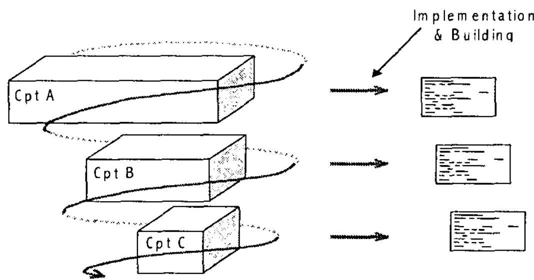

Figure 4 Implementation and Building Activities

There is also no reason why every logical component should be mapped to a separate executable. This represents one extreme approach[5]. As shown in Figure 5, at the other extreme it is also possible to combine all the abstract components into a single executable (e.g. a program). The choice depends on the prevailing needs of the customer when the executable form is generated.

Another important point to note about the KobrA approach so far as the creation of executables is concerned is that it is not necessary for every component in a hierarchy to be included in every executable image. This is because of the fact that since KobrA describes every component uniformly, whether it be the top-level component at the root of the tree, or smaller components lower down, every component that provides useful services to a potential user can be treated as a system, and mapped into an executable image. Thus, when an abstract component is incorporated into an executable image its subcomponents must also be contained in the image since they are in a sense part of it.

[5] It is assumed that abstract components represent the smallest level granularity (i.e. they will rarely be split up into more than one executable).

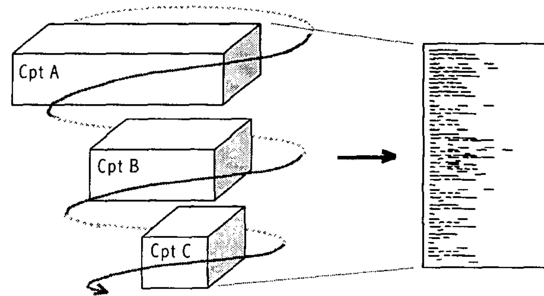

Figure 5 Single Executable

However, a component's supercomponent need not necessarily be included. Figure 6 below illustrates a scenario in which a component other than the root of the tree is selected to represent a system and is translated into an executable form.

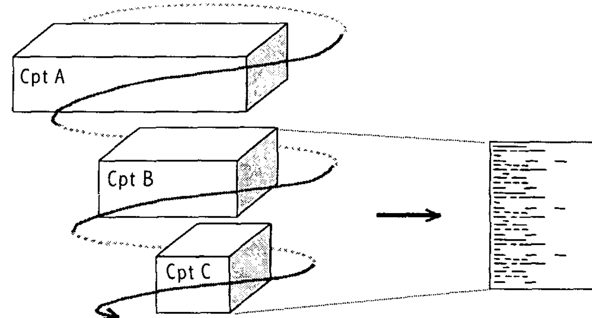

Figure 6 (sub)component as a system

3.4 Component Reuse

Another important aspect of the KobrA process is the so-called component reuse activity. The KobrA development activities described to this point are all concerned with the development of new software components, and thus all lean towards a top-down style of development. However, this is not the way in which

the component-paradigm is supposed to work. Instead, application development should support the assembly of existing components as well as the creation of new ones where needed. This is the role of the component reuse activity in KobrA.

Whenever a new component specification has been defined, the KobrA developer has the option of realizing it from scratch using the regular realization activity, or realizing it using an existing component if one is available. This is essentially the infamous "make" or "buy" problem. The make option corresponds to the regular realization activity while the buy option corresponds to component reuse.

The first task when pursuing the buy option is to find a potentially suitable component whose services and properties provide a reasonably close match to those required. Of course, if a perfect match is found, and a component is available that provides a perfect realization of the specification, this is simply used without modification. This is rarely, if ever, the case however. Usually candidate components have at least a few properties that do not exactly match the requirements defined in the specification.

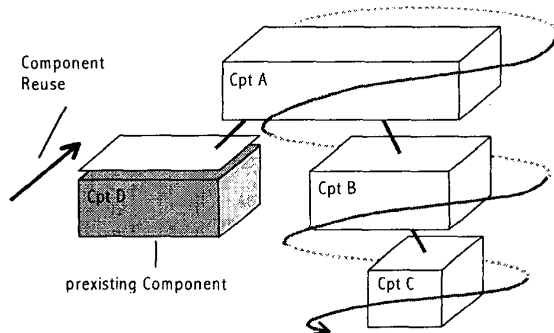

Figure 7 Component Reuse Activity

The next step is therefore a process of negotiation between the candidate component and the potentially using component to see if an agreed-upon interface can be determined. If the candidate for reuse can be modified in some way, or is highly parameterized, both parties have the flexibility to adjust to a slightly modified interface. However, in many cases the implementation of the candidate component is completely fixed, or may not even be known, as with COTS components for example, so all adjustments have to be made within the realization of the using components. If the candidate component and the potential user can agree on a mutually acceptable interface (i.e. specification in KobrA terminology) - that is an interface which the former can use and the latter can fulfill, then the candidate component can be used and a fresh realization can be avoided.

As shown in Figure 7, since it enables existing components to be inserted into a component tree wherever a match is found, the component reuse activity introduces a more bottom up way of developing a hierarchy of components. The overall KobrA process is therefore balanced between the top-down and bottom-up ways of proceeding.

3.5 Quality Assurance

As mentioned previously, one of the benefits expected from the component paradigm is increased quality (or the attainment of given quality levels more easily). In the long term this will arise as the proportion of tried-and-tested, off-the-shelf components to newly developed components increases. The less brand-new software that has to be developed for a new application the lower the chance of errors being introduced.

This notwithstanding, there will always be the need for some new software to be developed, even if it is only "glue" software in the root component's realization to configure a new assembly of preexisting components. Furthermore, this need is likely to be the greatest in the earliest phases of a company's transition to the component paradigm, when they are most vulnerable to project cancellation due to perceived failures. To help ensure that the quality expectations of users are satisfied, the KobrA approach includes several fully integrated quality assurance activities. Moreover, since the costs of errors rapidly increase the longer they remain undetected, the emphasis on KobrA is on early defect detection and quality evaluation. In addition to testing, this is achieved by means of two main techniques.

The first is the comprehensive and fully integrated use of inspections within all activities of the KobrA process. Every major development activity, such as specification, realization etc. culminates in a series of inspection activities designed to check that the extensive inter-model consistency rules mentioned above are satisfied. KobrA employs an form of perspective-based reading [7] that requires components to be inspected from the perspectives of various different stakeholders, thereby optimizing the defect detection effectiveness of the overall inspection process.

Another important aspect of KobrA's quality assurance approach is the early application of quantitative quality modeling techniques to try to evaluate the quality of development artifacts soon after they are created. Although it is usually the most critical quality issue, correctness is not the only concern. It is also important to be able to evaluate "how well" a specification or a realization, once correct, satisfies other quality issues, such as maintainability, reliability etc. In other words, it is important to ascertain to what extent a components adherence to non-functional requirements[6]. To help address this need, KobrA incorporates quality assessment techniques based on a quantitative analysis of the UML models used to describe components. Using statistical analyses of relationships between concrete internal measures (such as component coupling and cohesion etc) and externally visible properties, such as maintainability etc., it is possible to gain early indicators of the quality of particular realizations and specifications, and thereby to help direct limited resources to where they can be most effective.

Quality assurance activities of the kind just described are in fact of more importance in KobrA than in other component-oriented development approaches because of its reliance on a tree-based structure. When a development process involves the top down elaboration of a tree, errors made early in the process take on much more significance. For an example a mistake in the specification of the root component could later require the redevelopment of large parts of the component hierarchy, whereas errors made in lower level components will not have such a significant impact.

In its tree-based organization of the development process, and its emphasis on early quality control and defect detection, KobrA bears some similarities to the Cleanroom approach [9], one of the most systematic methods used in practical software engineering projects. In fact, many of the tree organization issue faced in KobrA are similar to those faced in the Cleanroom approach.

3.6 Incremental development

Another important feature of the KobrA method is its natural support for incremental development. Since the implementation and building activities are orthogonal to the component modeling and decomposition activities, as soon as a component realization has been completed it is possible to translate it into an executable form (i.e. an increment). Of course, to actually execute such an increment it is necessary to generate preliminary substitutes (i.e. stubs) for yet-to-be-built parts of the system (i.e. the subcomponents), but this is no different to any other incremental process. In KobrA, developers are free to choose when to translate a component realization into an executable form. Each component can be implemented and tested as soon as it is realized, to give a highly incremental approach to development, or the implementation, building and deployment activities can be postponed until the entire component tree is complete.

Looking at the development process in Figure 3 "from the top", so to speak, it is clear that the KobrA approach can be viewed as an incarnation of the well-known spiral process model. The main difference between KobrA's approach and other leading UML-oriented methods such as RUP [2], is that in KobrA the development increments are more component-oriented. Relying on use cases to drive the development process, as in the RUP, leads to development increments that are more function-oriented. This in turns creates a tension between the function (i.e. use-case) oriented increments and the component-oriented architecture. In KobrA, the development increments and software architecture are both component-oriented.

4. Framework and Application Engineering

The aspects of the KobrA approach described above all relate to the development and deployment of a one-off single system. While this is a perfectly feasible and effective way of applying the KobrA method, it is well known that the vast majority of software organizations actually have to support a family of related products, rather one single product. The KobrA approach therefore also accommodates a more product line approach to software development in which the differences between product variants can be explicitly modeled.

KobrA supports product lines by allowing component specifications and realizations to be augmented by so called decision models [8]. The purpose of a decision model is to relate properties of the component, as described by the UML models, to externally visible features that represent points of variation. Components that possess a decision model are referred to as *generic components*, and model the total set of features

[6] Commonly captured as the well-known "ilities".

possessed by all their variants in the product line. The decision model captures these variabilities in terms of the UML models and defines the rules by which they may be resolved in specific instances to create specific components.

Since a system is simply regarded as a component in KobrA, it is a natural extension to regard a product line as a generic component. Specific instances of the product can therefore be viewed as specific instances of the generic component, created by the resolution of the accompanying decision model. The general form of the KobrA product is therefore a hierarchy of generic components, or a framework, containing interdependent decision models. As one might expect, the resolution of decision models at one level in a framework will place constraints on how decision models lower in the component hierarchy are resolved. In extreme cases, choices made for a component may lead to some of its subcomponents not even being included in that specific variant.

In their most general form, all the development activities described in the previous sections can be used to develop a framework of generic components. They therefore form part of what is called the *framework engineering* activity of KobrA. The activities can also of course be used to develop a hierarchy of component without any variabilities (i.e. decision models) but this is simply viewed as a special case of the more general situation. Moreover, to remain consistent with the view that any component can be a system (or an application), KobrA takes the view that any generic component can be a framework.

Before an executing system can be derived from a KobrA framework containing decision models, it is first necessary to resolve the corresponding decisions to create a concrete instance. This is the role of the application engineering activity. The result is a hierarchy of concrete specific components, known as an *application*, which can be translated into an executable form in precisely the way described above. Each application instantiated from a framework in this way is said to be a *variant* of the framework.

An important characteristics of this approach is that its supports an incremental way of introducing product line ideas. A company can work with a single system version of the component hierarchy (i.e. a framework without variabilities) until they feel comfortable enough to start to introduce variant features and decision models. Moreover, the first decision models can be very simple.

5. Separation of Concerns

The strength of KobrA's approach to component-based and product line engineering is its strict separation of concerns. Not only is the product of a KobrA project strictly separated from the process for creating it, but as illustrated in Figure 8, the various development activities are carefully separated into three main dimensions. The abstraction dimension relates to the level of detail at which a hierarchy of components is described (i.e. UML or code level), the specificity dimension relates to the presence or absence of variable features from the perspective of a product line, and the composition dimension relates to the nesting of components within one another. By making solutions to these concerns as orthogonal as possible, different strategies can be "mixed and matched" in a manner that best suits a particular development organization.

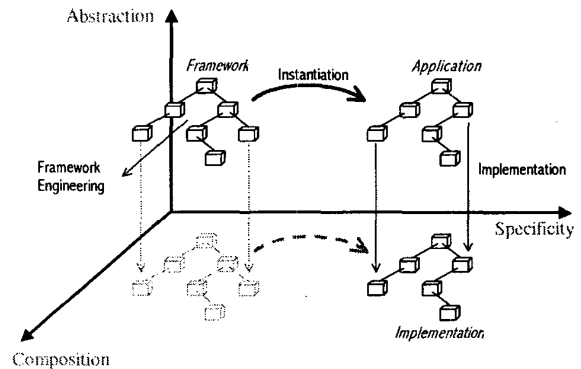

Figure 8 Separation of Concerns

A central dilemma for any development method is achieving the appropriate balance between prescriptivity (that is, the enforcement of concrete rules) and flexibility (that is, the provision of freedom for the developer to work as he or she sees fit). The KobrA method's approach to this dilemma is illustrated in Figure 9, which plots the degree of prescriptiveness for each of the main artifacts created in the general KobrA development chain.

As can be seen from this figure, the method is most prescriptive (i.e. defines the most concrete rules) in the earlier framework engineering phase, but the level of prescriptiveness decreases towards the later development stages where the method is much more liberal. We believe this provides the optimal support for practical migration to the component paradigm since it

gives companies prescriptive support where it is most need (in the earlier front end phases) but allows them to employ their own in house development technologies for the final implementation-oriented phases.

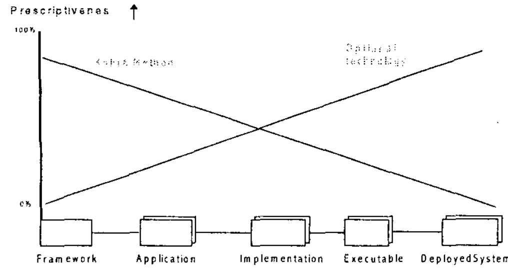

Figure 9 Degree of Prescriptiveness

6. Example

In this section we describe some experiences gained in applying aspects of the KobrA method in the development of an Enterprise Resource Planning (ERP) framework at PSIPENTA Software Systems GmbH, one of the leading ERP system producers in Europe. PSIPENTA took the opportunity to apply KobrA as part of the introduction of a new product line known as PSIPENTA.COM.

PSIPENTA's software products have traditionally been based on the typical client/server architecture. In the new product generation, however, this classical structure was enhanced by the introduction of a third abstraction layer, which contains a new user interface embedded within the Microsoft Internet Explorer.

The underlying functionality of PSIPENTA is provided by around 600 so called PSIPENTA business objects. These are not physical components in the sense of a DLL, a COM server or a Java Bean, but logical (i.e. functional and descriptive) components in the sense of KobrA. As illustrated in Figure 10, the underlying architecture of the new system is distributed. PSIPENTA business objects are thus typically deployed on different computers.

Although each business object possesses its own data and behavior, they all interact with the user via the same basic user interface model. In particular, the interface to each PSIPENTA business object is composed of three master window types: *filter*, *overview* and *detail view*. A business object is first opened in the filter view. Following the specification of the appropriate filter parameters, the system then shows the filtered data sets in the overview view. The user may subsequently select specific data sets for closer analysis in a detail view or may manipulate a subset of the data in a processing box. Processing boxes are additional windows associated with business objects to provide additional optional inputs.

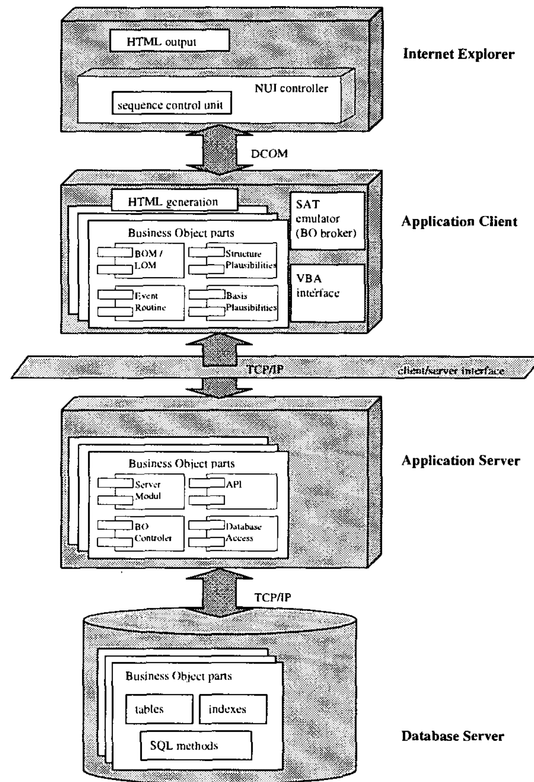

Figure 10 Architecture of PSIPENTA.COM

This PSIPENTA-specific window handling approach is referred to as the standard sequence (in German: “Standardablauf”). A central component in the system known as the standard sequence driver (in German: “Standardablauftreiber”, or “SAT”) is responsible for managing the execution of this standard sequence for individual business objects. It achieves this by means of so called SAT events driven by user inputs. Figure 11 shows the possible window transitions for a PSIPENTA business object and the associated SAT events which initiate these transitions. In the new product version of the system, PSIPENTA.COM, the SAT takes the form of an emulation module on top of the central *Business*

Object Broker (BOB). This makes it possible to use all the business objects in the manner familiar to PSIPENTA users, but in addition any business object is callable via the BOB from outside the PSIPENTA system using COM.

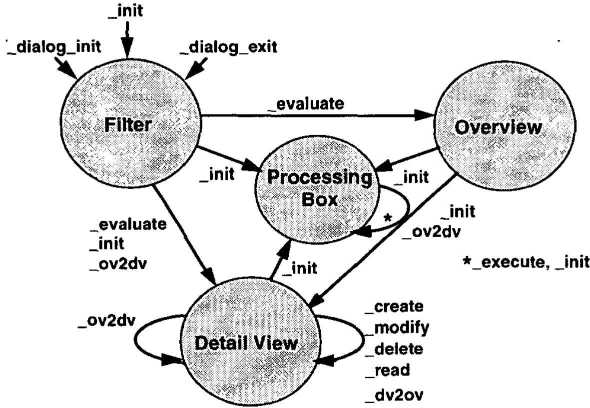

Figure 11 BO Window Crossovers and SAT Events

In developing the new product generation, the central goal was to establish a framework which embodied all important information about business objects, including the standard user interface sequence, window alternation and the generic handling of SAT events. The aim was to make it easier and faster to build specific PSIPENTA business objects, without the need for any code duplication to access PSIPENTA components for business object management and control.

The application of the KobrA approach helped us to view business objects as logical systems, each constructed from a possibly large set of subsystems. Figure 12 illustrates the overall KobrA component tree for the new framework. The first level components below *PSIPENT BO - Interface*, *Business Logic*, *Data Model* and *Persistence*, are abstract components which describe the common properties inherited by the concrete components in the levels below.

These components were partly decomposed into lower-level subcomponents during subsequent steps. For example the GUI component was decomposed into subcomponents related to the three main window types and a generic processing window component. During the decomposition processes the strict rules of the KobrA method proved to be particularly helpful in managing the overall execution of the project. KobrA was also very helpful during the specification phase for identifying important subcomponents, deciding what kinds of diagrams to develop and what they should contain.

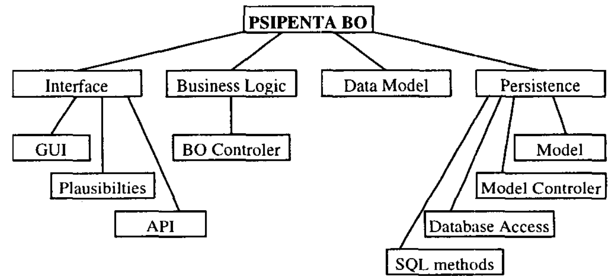

Figure 12 PSIPENTA BO component tree

The first use of the new framework was in the generation of a new system for user authorization. Figure 13 illustrates the individual subcomponents making up this authorization system. Since our framework was evolved for a single PSIPENTA BO, we effectively developed an application in the sense of KobrA for each of the new PSIPENTA BOs in this figure.

The connections between the individual components show the relationships between the underlying data structures in the new authorization system. After instantiation of the application, the specific functionality and the real data structures were added for each business object, and the specific forms and displays for the user interface were defined.

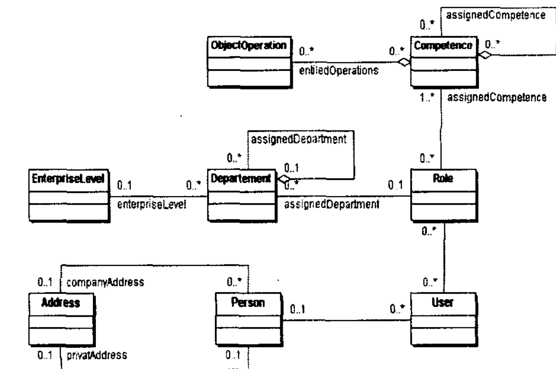

Figure 13 Authorization System Business Objects

The use of the framework to develop specific applications and to build new PSIPENTA BOs has appreciably improved the reusability, adaptability and

extensibility of the resulting components. The framework has not been in operation long enough to judge the long term return on the investment put into its development, but the effort involved in creating new PSIPENTA BOs has been significantly reduced.

In the future, the sale and deployment of ERP systems will increasingly depend on the quick and cost effective adaptation and extension of generic software to customer specific needs. By application of the KobrA method, PSIPENTA was able to develop a general-purpose framework for the new component based version of its ERP software PSIPENTA.COM. This makes it possible to respond quickly and flexibly to new user requests by adding new components. In particular, it simplifies the interoperation of PSIPENTA with third party software such as workflow systems, e-business applications and so on.

7. Conclusion

The binary-module view of components that currently prevails in the UML and contemporary component technologies (e.g. COM, CORBA and EJB) is limiting their impact in domains where their benefits were widely heralded, such as e-business and web-based development. In this paper we have described a method, known as KobrA, which tries to address this problem by supporting a more model driven approach to component-based development that enables components to be exploited in all phases of development. As well as defining a systematic way of using the UML to model components, the method also employs a product line strategy for making them generic. This involves the creation of generic frameworks that can be rapidly instantiated and adapted as needed to meet the needs of specific customers.

Most of the development strategies embodied by KobrA are not new. What is new about the method is the way they are integrated into a coherent whole. By cleanly and strictly separating concerns for different development dimensions (e.g. containment, genericity and abstraction), and keeping the number of distinct development concepts as small as possible, KobrA enables the principles of component-based development, product line engineering and model driven architectures to be used in a simple and systematic way.

To date, the main example of KobrA's application on an enterprise level is in the development of the Enterprise Resource Planning (ERP) system reported in the final part of the paper. This satisfactorily demonstrated the applicability of the key aspects of the method, but insufficient time has passed to judge whether the investment in reusable artifacts has paid off in a significant way. However, preliminary indications are that this is the case. Further information will be provided by additional industrial applications of the method currently in the pipeline.

Acknowledgements

The authors are grateful to their colleagues on the KobrA team for their contribution to the ideas in this paper.

References

1. OMG Unified Modeling Language Specification, Version 1.3. OMG document ad/99-06-08, 1999.
2. I. Jacobson, G. Booch and J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley, 1998.
3. D. D'Souza and A. C. Wills, Catalysis: Objects, Frameworks, and Components in UML, Addison-Wesley, 1998.
4. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H. Hayes, F., and Jeremaes, P., Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.
5. C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1998.
6. J. Rumbaugh et. al, "Object-Oriented Modeling and Design",., Prentice Hall, 1991.
7. O. Laitenberger, C. Atkinson, Generalizing Perspective-based Inspection to handle Object-Oriented Development Artefacts, ICSE'99, 1999.
8. C. Atkinson, J. Bayer and D. Muthig, "Component-Based product Line Development: The KobrA Approach", First International Software Product Line Conference, Pittsburgh, August 2000.
9. Deck, M., "Cleanroom and object-oriented software engineering: A unique synergy". In Proceedings of the Eighth Annual Software Technology Conference, Salt Lake City, USA, April 1996.