Fraunhofer Institut Experimentelles Software Engineering

Product Line Implementation Technologies

Component Technology View

Authors:
Stefan Kettemann,
Dirk Muthig
and Michalis Anastasopoulos

Funded by Stiftung Rheinland-Pfalz für Innovation, Project #15202-386261/51: “Entwicklung und Erprobung eines Methodenleitfadens für Software-Produktlinien-Implementierungstechnologien“

IESE-Report No. 015.03/E
Version 1.0
March 20, 2003

A publication by Fraunhofer IESE

Executive Summary

Nearly all software organizations today develop and maintain more than a single product. This holds for organizations that develop tailored systems individually for single customers, as well as for organizations that develop products for a mass market. The products developed by an organization typically are similar applications in the same application domain. Hence, these products share some common characteristics and thus can be viewed as a software product line.

To implement a product line approach in practice, special technologies are required that effectively support the identification of reusable artifacts, as well as explicit means for capturing and controlling commonalities and variabilities. The focus of the PoLITe project are product line technologies at the implementation level. PoLITe defines three categories of implementation technologies [MAL+02], namely configuration management, component technologies, and generative features of programming languages (including generators). This report summarizes the product line implementation aspects in the component technology dimension.

The report first presents the term “component technologies” and its role in the PoLITe context, surveys the actual distribution of component technologies in practice. It then establishes a component model that facilitates the seamless integration of component technologies in a model-driven implementation process. This implementation approach is subsequently embedded in a product line implementation process. Finally, the application of the principles is illustrated by a case study using J2EE/Enterprise Java Beans as component technology.

Keywords: software product lines, product line implementation technologies, variability mechanisms, component-based development

TABLE OF CONTENTS

1 Introduction

Nearly all software organizations today develop and maintain more than a single product. This holds for organizations that develop tailored systems individually for single customers, as well as for organizations that develop products for a mass market. Even for organizations that believe to develop a single product only, surveys have uncovered that also these organizations spend most of their resources on tailoring their systems to the needs of individual customers or enhancing systems by features that are newly required by customers [KHS98], and thus also these organizations must maintain and evolve a set of customer-specific variants.

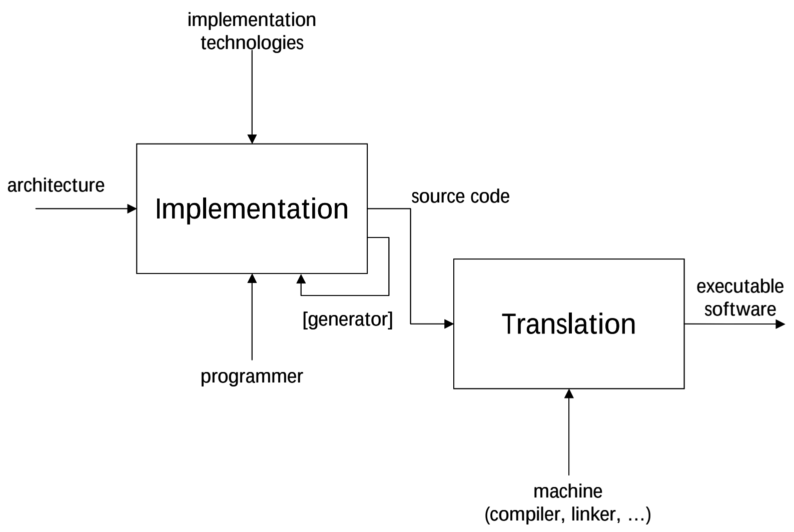
The products developed by an organization typically are similar applications in the same application domain. Hence, these products share some common characteristics and thus can be viewed as a software product line. Typically, the complexity and size of these software products, today, rapidly increases and customers are requesting more and more quality products tailored to their individual needs. Due to these increasing challenges, also the requirements on the development skills of an organizations increase. Hence, there is a need for organizations to learn how to manage a product line.

Software product lines are a reuse approach; the reuse approach that promises to master all of the described challenges [ABB+02]. To implement a product line approach in practice, special technologies are required that effectively support the identification of reusable artifacts, as well as explicit means for capturing and controlling commonalities and variabilities. The focus of the PoLITe project are product line technologies at the implementation level.

1.1 Product Line Implementation

In traditional software development approaches implementation typically means producing source code [Cop99]. The implementation activities refine a system's architecture down to a level that can be interpreted by a machine. Hence, the information captured by implementation artifacts is at the lowest level of abstraction compared to the content of all other development artifacts. The implementation artifacts (i.e., the source code) can be transformed automatically into an executable form. The implementation process is, at a general level,

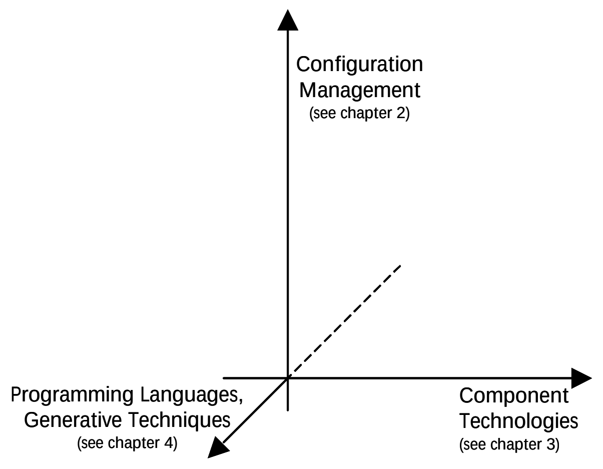
Figure 1: Software implementation



depicted in Figure 1 (using the IDEF0 notation). It covers all implementation-related activities in the general product line life-cycle and integrates them into a single implementation process. Hence, this process only implicitly distinguishes between implementation-for-reuse and implementation-with-reuse. In the PoLITe project, we discuss implementation technologies in the context of this process because it allows both aspects of an implementation technology to be discussed together independent of life-cycle issues.

The PoLITe project defines three categories of implementation technologies [MAL+02], namely configuration management, component technologies, and generative features of programming languages (including generators). The three

Figure 2:Technology dimensions enabling the automation of implementation approaches



dimensions are depicted in Figure 2. Initially, each dimension is investigated separately.

This report summarizes the product line implementation aspects in the component technology dimension.

1.2 Component Technology Dimension

This report purely presents an approach to manage currently available implementation technologies from a component technology point-of-view. Note that the other two dimensions, configuration management and programming language technologies, are investigated and described in reports of their own [LM03], [PM02].

The underlying theme is to find an efficient bridge between product line architectures and its generic implementation. To understand the situation we first look at the way single systems are implemented. In single-system cases, software is created on a one-by-one basis without any pro-active reuse activities. That is, the models describing a required system are realized straight-forward.

In a product line context, the series of products delivered by an organization is seen as a series of products that are variations of the same infrastructure. From this viewpoint, models describe not only a single system but a set of similar systems in the same application domain. Hence, the models capture generic as well as variable features of the product family.

This is where the component paradigm comes in - by encapsulating genericity and variability in separate components it facilitates to establish the required product line infrastructure. Building a specific member of the product line is then simply done by plugging in or plugging out the variable components that are required (or not required) for a specific product.

Product line approaches that apply these principles have primarily been concentrated on the activities producing models covering the right scope of genericity, that is, on activities early in the software engineering lifecycle [ABB+02]. The component technology dimension that is being surveyed in this report focuses on the phases later in the lifecycle where models are transferred into implementations using concrete component technologies. In this context, the report's contribution is a model-driven implementation approach that is designed for a high automation rate. Finally, the presented optimization in the automation of development activities brings the report in line with the PoLITe focus on improvements in process-automation [MAL+02].

1.3 Outline

This report summarizes product line implementation technologies from a component technology point-of-view. The remainder of the report is organized in the following way:

Chapter 2 defines the term “component”, surveys the actual distribution of component technologies in practice, and selects a component technology that is used in the case study: J2EE / Enterprise Java Beans (EJB).

Chapter 3 introduces a component model on the architectural level. This facilitates smooth transitions when proceeding from the architecture to the concrete component technology level during implementation. The conceptual correspondence to Model Driven Architecture (MDA) and its automation principles is also presented.

Chapter 4 presents the details of a semi-automated implementation process, i.e. involved models, artifacts and workflows. The presented work differs slightly from the approach suggested by the Object Management Group (OMG) [OMG01-A]. For example, so called refinements are introduced in order to explicitly comprehend and record engineering activities on the technological level.

Chapter 5 integrates the suggested approach into the overall realization of product lines.

Chapter 6 illustrates and validates the presented concepts in a case study.

Chapter 7 summarizes the report and gives an outlook on future research activities to further improve the current state-of-the-practice.

2 Component Technologies

In this chapter we survey the following questions:

- What are the core concepts and -problems of components in industrial practice ?
- State-of-the-practice: What are the characteristics of conrete component-technologies, especially in conjunction with the core concepts ?

2.1 Core Concepts

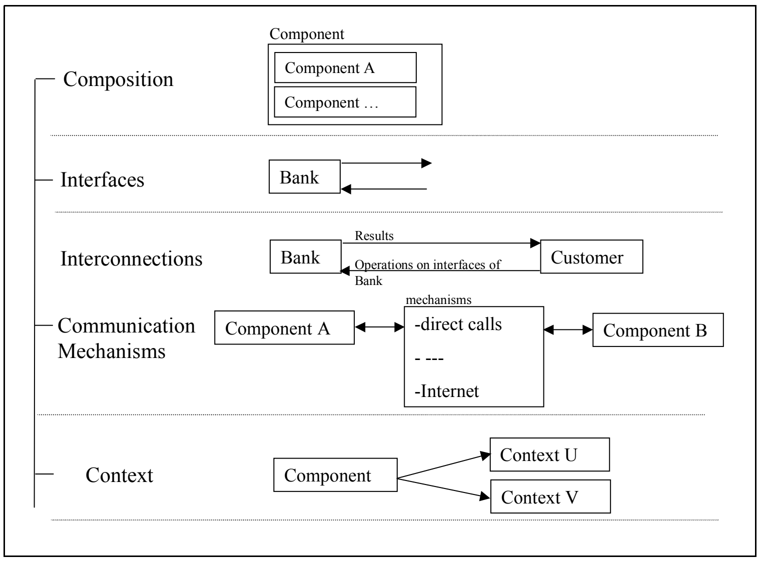
A commonly accepted definition for components is:

“A software component is a unit of composition with contractually specified interfaces and explicit context only. A software component can be deployed independently and is subject to composition by third parties“ ([Coi96]).

Collecting and refining the features of components we can summarize them as follows (See [Cle01], p. 350-404; , [Szy98], pp. 29) which is also depicted in Figure 3:

- Composition
- Interfaces
- Interconnection
- Communication Mechanisms
- Context Dependencies
- Visually Configuring & Composing
- Deployment

Figure 3:
Component Core Concepts



Composition

Component Composition is dealing with the inner structure of a component. Typically a component can consist of other components. This principle, called containment, can be recursively applied top-down which leads to containment trees.

Interfaces

Interfaces describe the interaction between a component and other software elements (e.g. other components). Syntactically component interfaces are described by an Interface Description Language (IDL). Normally there is a distinction between Export and Import interfaces. Export interfaces operate as callees providing services to callers (other software elements), whereas Import interfaces operate as callers themselves, i.e. they call services of other software elements.

Interconnections

Interconnections define which components communicate with each other. A so-called MIL (Module Interconnection Language) describes such interconnections (elements: components, interfaces, connectors). Special cases in this area are:

- Interface Adapters (when interfaces do not match exactly)
- Asynchronous Connections

- Push and Pull Connections

Communication Mechanisms

Communication mechanisms constitute the technical backbone for components to communicate with each other. Java for example offers a mechanism called RMI (Remote Method Invocation) which enables the communication between remote software elements. Communication mechanisms play a crucial role for component technologies because they also define the restrictions for application integration. This becomes even more obvious if we understand that the internet has initiated a huge demand for application integration. Hence we can observe how it pushes new communication technologies like .Net or SOAP (Simple Object Access Protocol), all of them having the ultimate goal to integrate different applications in a highly heterogenous environment.

Context dependencies

A given component always makes assumptions about the context it will be used in. These assumptions are dependencies which decrease the range of systems that can use the component. Therefore dependencies should be minimized because fewer dependencies mean a wider range of systems that can use the component. Typical examples for such assumptions are:

- usage of global variables (e.g. logfiles)
- interface types which restrict the interface to work only for one type
- assumption about communication mechnism (e.g. RMI)

In order to define the context in which it can be used, a component should contain a self-contained context-description.

Visual Composition & Configuration

Components and their hitherto described core-concepts represent a new high-level view on software artifacts. In order to enable a component-oriented development it is required that component technology vendors support the visual composition & configuration (connections) of components.

2.2 Automation

So far, we have collected the constituting elements of component-technologies. However, we have not shown yet what the benefits of component-technologies are. Like in any production process also in software development a new technology should improve productivity and quality. Both goals are usually accomplished by an increase in automation. Surveying this question in the context of component technologies we realize that there are several mechanisms of automation:

Each component technology brings along its own **communication mechanism** (see also next chapter).This relieves the developer from implementing mechanisms of his own. Tools for enabling the **visual composition & configuration** of software components provide an effective human-oriented handling and also the **generation of code** from the visual representation. All the surveyed component technologies provide **additional services** which increase productivity and quality of the programmers work. Examples of such services are: naming services, persistency, transaction management, security, concurrency.

2.3 State-Of-The-Practice

Referring to the component core-concepts that we have described above, we will now survey the following component technologies:

- COM, DCOM,
- .NET
- Enterprise Java Beans - EJBs
- CORBA Component Model - CCM

In addition to the component core-concepts we provide a section “General Purpose Features” which gives an overview over those concepts which do not directly relate to components (see [GT00], pp. 10-13).

2.3.1 Component Object Model (COM)

COM/DCOM is Microsofts standard for component-models. It is an evolved standard which is reflected in different names for the different versions. The initial version **COM (Component Object Model)** adresses the early version which can only be applied in a non-distributed environment. **DCOM (Distributed Component Object Model)** is a later enhanced version which supports distributed environments. Although initially implemented only for Microsoft platforms (Win95/98, WindowsNT, Win2000) it is now also available on other platform like Solaris, OS/390 and Mac OS, thus enabling to use it in heterogenous environments. As described in [EE1998], p. 19-26, COM’s philosophy is that components are parts of software in binary form which shall facilitate integration and reuse of software. Hence COM defines a binary standard for component-interoperability.

Composition

COM provides two alternatives for Composition - Containment and Aggregation.

Interfaces

A central concept of COM-components is that their behaviour is defined in an interface which resembles a typed contract between the component and a potential client. There can be multiple interfaces for one COM-component. The implementation of an interface is not done on component level but rather on class level, i.e. a so called COM class which resides in the COM component implements the interface. COM allows to extend a component with new interfaces (existing interfaces have to be kept untouched) which provides a kind of versioning. Interface descriptions are written in MIDL (Microsoft Interface Definition Language)

Interconnection

Eventhandling in COM is done via so-called connection points: the sender defines outgoing interfaces, the receiver defines an eventsink.

Communication Mechanisms

COM provides a communication mechanism of its own using a COM-server in the background which processes the communication between two components which reside on the same machine. This means that COM-communication is restricted to a non-distributed environment. DCOM is showing a clear extension here, allowing the communication between COM components in a distributed environment (i.e. separate machines connected via a network).

Context & Dependencies

However, the most relevant context-dependency is hidden here : a COM-component assumes that it communicates with another COM-component. This means that communication to non-COM components is not supported. (see [Wes02], p.6).

Visual Configuration & Composition

Visual Programming is only provided for ActiveX Controls (which are based on COM), but not for COM-components.

Deployment

Installation is done by registering DLL- or EXE-files into the windows registry. There are no further preconditions for their activation.

General Purpose Features

COM is defining a binary standard which is furthermore independent of the programming language. For example it doesn't make a difference if components are developed with Visual C++ or Visual Basic. Usually, COM-components are realized via classes (Component structure). Persistency has to be implemented by the developer, it is eased by using services from a mechanism called Structured Storage. Transactions are possible either using Structured Storage or Microsoft Transaction Server (MTS). Mechanisms for security, concurrency, memory management and exception handling are available. The Microsoft Foundations Classes (MFC) and the Active X Template Library (ATL) are the two most popular frameworks for developing COM-components. Dynamic Loading is done with Dynamic Link Libraries (DLLs).

2.3.2 .NET

In .NET Microsoft has removed many of the weak points in the COM/DCOM architecture. Therefore it can be seen as the consequential advancement of COM. Interoperability between different technologies is maybe the most important innovation:

The Internet and new hardware-platforms (e.g. PDA's) generate a huge demand for interoperability between different platforms and applications. COM doesn't address this communication problem as it only supports communication between components of the same technology - COM. .NET is filling this gap, adressing explicitly internet-interoperability and platform independency (see [Wes02], p.13):

- .NET has to provide a platform for the use and offer of component-services in the Internet.
- . NET has to support standards in order to provide interoperability with Internet software services based on other platforms.
- . Net has to be leightweight enough to be deployed on a variety of platforms.

The key concepts for providing application-interoperability over the Internet are (XML-) **Webservices & SOAP**. A Webservice is a website offering an XML-based API. The API defines all functions the service consists of. However, it does not specify the implementation, i.e. how the functions are realized. A caller sends requests to the service via XML-packages (specifying methodname & parameters), which are responded in the same way, i.e. with XML-packages (see Figure). Thus platform-independency is established - it doesn't matter for caller or service, which platform is actually used internally. In addition, an XML-based standard describing Webservice-calls is required. The quasi-standard here is SOAP - Simple Object Access Protocol.

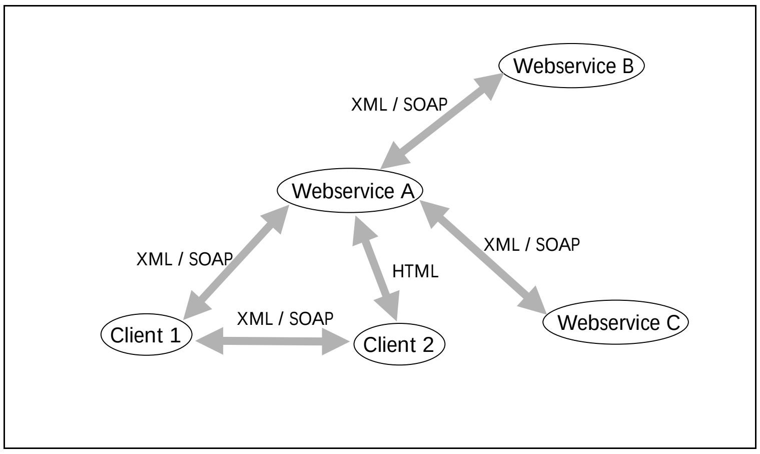


Figure 4: Interoperability of Internet components cooperating as Webservices

We can summarize that Webservices are an internet-compatible concept for components. The implementation of a given Webservice-API can be done with any .NET-language (e.g. VB, C#) by implementing respective classes and methods. In addition to the webservice concept which allows remote interoperability, .NET also provides a binary component-concept which defines how components are physically deployed on a given hardware. These physical runtime entities are called assemblies (see - general purpose features).

Composition

Interfaces

In .NET the each Webservice is specified by XML-based APIs. They represent interfaces (see also communication mechanisms).

Interconnection

Communication Mechanisms

With SOAP communication is possible between components built from different component technologies (see the introduction to .NET above).

Context & Dependencies

While DCOM restricted caller & service to use the same technology (DCOM), with Webservices & SOAP, .NET has removed this dependency.

Visual Configuration & Composition

Deployment

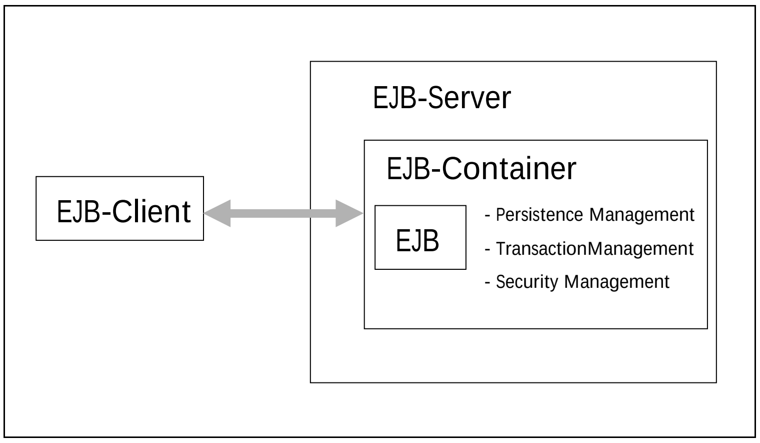
The deployment-entities in .NET are called assemblies. They are obtained by compiling the source code to executable binary files. For deployment these binary components are held in one-project-specific folder. Additionally each assembly contains a so-called Manifest which holds the whole metainformation (types, interfaces) of the assembly. This means that in . NET installed code and the required metainformation constitute one entity. This deployment principle leads to more robust installations compared to COM where the metainformation was held in one place, the registry. The latter one lead to a lot of problems as it happened quite often that different installations mutually overwrote their respective metainformation.

2.3.3 Enterprise Java Beans (EJB)

The EJB-technology is developed by SUN Microsystems. In contrast to COM which is a proprietary technology (Microsoft), a lot of industrial partners like IBM, Oracle, Bea were involved in the specification process of EJBs. Therefore, the EJB specification has the character of a quasi standard.

An Enterprise Java Bean is a component which can be deployed wherever an EJB Server is installed. A bean does not communicate directly with other components, but is always running wrapped in a special environment, called EJB-container. The container provides additional services like the bean's invocation, persistency and security (see Figure 5).

Figure 5:
EJBs, EJB-Container & EJB-Server



Furthermore the specification EJB 2.0 distinguishes three kinds of beans:

- Entity Beans are persistent in the database (e.g. customer information),
- Session Beans are not persistent - their maximum lifetime is the lifetime of a session (e.g. shopping basket),
- Message Driven Beans provide in their interfaces asynchronous messages instead of synchronous methods - they are used for the implementation of message-oriented services.

Composition

EJB does not provide mechanisms for composition, yet it is possible for an instance to reference other instances via Remote Interface.

Interfaces

EJBs require two interfaces. The home interface defines lifecycle-methods (e.g. create, destroy) whereas the remote interface specifies the businesslogic behaviour.

Interconnection

Communication Mechanisms

For a remote EJB-client the EJB system consists only of the bean's remote and home interface, everything else is invisible. The EJB-server has to support this client view. This means that he can use any protocol for distributed object services like CORBA IIOP or Java RMI, but this must map to the Java RMI-IIOP programming model [Mon01].

Context & Dependencies

Context-descriptions are done in the so-called Component Contract of the EJB-Container.

Visual Configuration & Composition

As component attributes and events are not explicitly handled, they cannot be handled via Visual Programming.

Deployment

For each Bean a so-called deployment descriptor contains relevant metainformation which specifies the Beans behaviour concerning persistency, transaction support and security.

General Purpose Features

EJBs (as well as java) are translated into a portable bytecode that represents the binary standard. The component model is not (!) independent from the programming language, as at the moment the implementation can be done only in java. Encapsulation is handled via interfaces so that there is no direct access to inner component-data. The mechanisms for identity are naming conventions on component level, whereas on instance level there is no explicit identity (like in COM there is only implicit identity in the sense that it can be checked if two references refer to the same object). There is no mechanism for versioning EJB's. Scalability is explicitly taken care of in the EJB-architecture so that availability can be maintained even with an increase in components and clients. Although EJB does not provide an explicit mechanism for data transfer, it may be implemented via RMI (Remote Method Invocation). EJB-components are implemented via classes. Persistency & transaction support is integral part of the EJB architecture. EJB provides mechanisms for security, concurrency, memory management and exception handling. Frameworks: For the development of EJBs the package “javax.ejb“ is required. In addition the used IDE may require proprietary extensions. Dynamic Loading: Currently it is not possible to extend a running EJB application server with EJB-components. EJBs are distributed as EJB.jar-files, their installation is mostly done via startscripts. The activation process requires separate preparation steps.

Despite the fact that EJB is still a young and evolving technology, it has already gained a good resonance in the market and is estimated to play a kex role in the future component market.

2.3.4 CORBA Component Model (CCM)

CORBA can be characterized as an open standard that is defined by the OMG (Object Management Group), a consortium of numerous members from industry. Like in EJB, CORBA's component model is using containers where components are embedded. Each component and its container has to conform to one out of 4 predefined component-types: service, session, process and entity. The containers provide their components with services like persistence or transaction support (container managed persistence / transactions), see[SP01].

Composition

Composition is done with the relationship service.

Interfaces

IDL (Interface Definition Language) is used for the Interface descriptions.

Interconnection

Communication Mechanisms

CORBA was designed to enable open interconnection of a wide variety of languages, implementations and platforms. Thus CORBA-components do not interoperate on binary level, but use a high-level protocol, IIOP (Internet interORB protocol).

Context & Dependencies

Visual Configuration & Composition

So far, CORBA doesn't support visual programming.

General Purpose Features

Platform independency is one of the key-features in CORBA's architecture. There are numerous languages in which CORBA-components can be implemented. Encapsulation is provided via interfacing like in COM & EJB. Mechanisms for identity are also integrated. Services for administration of component-versions are under specification. With the publisher-subscribe model and the event service CORBA provides powerful concepts for eventhandling. Mechanisms for datatransfer are available since version 3.0. The mappings from CORBA-IDLs to a specific language is handled differently by the different vendors. CORBA defines mechanisms for persistency but they are rarely supported by vendors. Flat and nested transactions are supported. Mechanisms for security, concurrency, memory management and exception handling are provided.

Frameworks are available from different vendors. CORBA allows dynamic loading. The physical distribution form depends on the programming language, for example java bytecode-files are used. Components are installed by registration in the implementation repository. Activation requires starting of a background process (daemon).

2.4 Technology Selection

In an industrial project, a specific technology has to be chosen for the implementation. The selection has to take into account requirements that are crucial for the to-be-implemented system. For example, a banking system requires that the component technology provides a thorough support for transaction management, security and authentication. Our purpose in this survey - illustration and validation of the presented implementation approach - does not predefine certain technical requirements. It is sufficient that the technology follows a component-oriented paradigm. Randomly we chose EJB for our case-studies, see Chapter 6.

3 Component Model

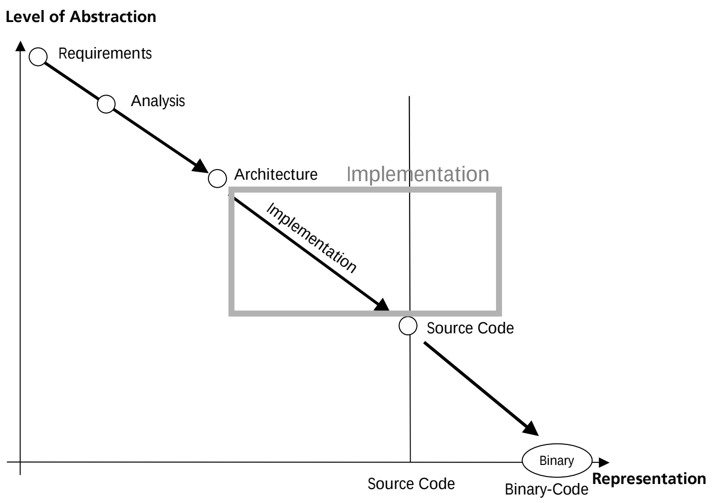
In the last chapter we had a thorough look at the constituting elements of component technologies. In this chapter we present a component model that forms the integrating basis in a component-based software development process. The integration of principles derived from Model Driven Architecture (MDA) allows to automate considerable parts of the implementation process.

Starting point of our survey is the analysis of the traditional implementation process and its shortcomings.

3.1 Traditional Implementation Process

In a traditional implementation process (see Figure 6) a programmer would directly implement the source code from a technology-independent architecture.

Figure 6: Traditional Implementation Process



Unfortunately this approach bears several disadvantages:

- **Paradigm Gap :** Current architectures being used do not capture components (this is because approaches for modeling components on an abstract level are relatively new, e.g. [ABB+02]). This means that suddenly - on the implementation level - a component-concept has to be introduced. This para-

digm shift from e.g. an object oriented model into a component-oriented one is likely to produce problems such as inconsistencies and errors.
- **Poor Reuse & Flexibility :** If the architecture does not cover components (see above) the only available component-artifacts will be the technology-dependent component implementation. This will make it quite hard to adopt the component implementation to another component technology. As a component implementation contains the specifica of technology A it cannot be transformed directly to technology B, rather the (full) implementation process has to executed again.
- **Missing Automation (poor efficiency and quality) :** The implementation process itself is 100% individual intellectual work where the quality of the implementation totally depends on the individual developers capabilities and intuition. We will show that there is quite a big amount of activities that could be automated resulting in a higher productivity and quality.

These disadvantages can be overcome with the help of a component architecture and a systematic way of doing the implementation step. (discussed in the next two chapters).

3.2 Benefits using a Component Architecture

A component architecture that explicitly adresses components helps overcoming two of the mentioned problems - namely the paradigm gap and poor reuse (see Figure 7). Using a component architecture a paradigm gap just doesn't exist because component architecture and component implementation speak the same language - both are talking in "components". This facilitates the implementation with a concrete component technology. And since the component architecture is a high level artifact which abstracts from technology it can be reused for the implementation with any component technology.

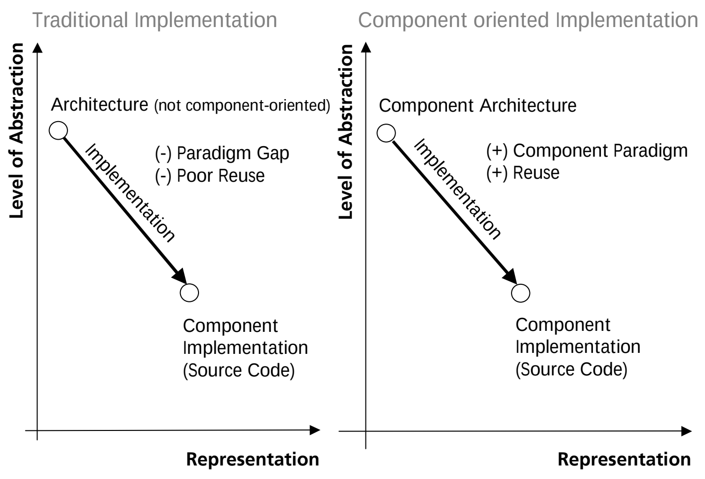


Figure 7: Implementation using a component architecture

But what is a component architecture and how does it look like ?

Cheesman and Daniels provide a definition which complies with most of the elements we have elaborated so far (see [CD00], p.13):

*".. a **Component Architecture** is a set of applicationlevel software components, their structural relationships, and their behavioural dependencies."*

In our context a component architecture is a component-oriented model of a business application. The model aspect requires a meta model that reflects the characteristics of the above definition. Such meta models are just being developed. In this report we are applying the KobrA-method as one possible meta model ([ABB+02]). The advantage of this method is that it is built up on top of UML introducing only a few extensions (for example the notion of a component as modeling entity).

The description of the KobrA method is out of scope in this report, for details please refer to [ABB+02]. The model examples used in the case study should be suited to make someone familiar with the method's principles.

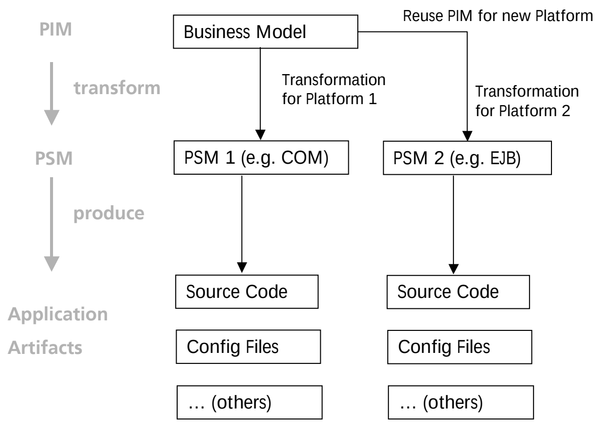
3.3 Model Driven Architecture

The Model Driven Architecture - MDA - is an initiative of the OMG. Its goal is to create a higher return of investment for the development of IT applications by reusing Business Models over changing platforms and automating a considerable amount of current implementation efforts. This is reasonable because business models tend to change far less often than technology/platform does. Therefore MDA focuses on models placing them at the center of development and treat them as the key IT investments.

Separating Business Logic from Technology

In order to facilitate the migration to any upcoming technology the Business Model needs to be totally independent from any technology. In the terms of the OMG these models are called **Platform Independent Models - PIMs**. Using technology-specific transformations the PIMs are transformed into **Platform Specific Models - PSMs**. Figure 8 shows how different platform specific models are derived from one platform indepenedent model. Finally the artifacts on code level are produced from the specific PSM (see also [OMG01-B]). The clear separation from each other allows that the PIM be reused and adapted to any technology. Furthermore the adaption to each technology can be highly automated by Transformations. Reuse and automation can improve drastically the productivty & quality of software development & maintenance. Thus the investment in establishing PIMs promises high RoIs.

Figure 8:
PIMs and PSMs in the MDA approach



MDA recommends the UML as meta model to be used for the business modeling. Additionally the OMG is specifying standards for industry-specific UML profiles. The goal here is to provide standardized UML extensions that cover the specific characteristics of a business area so that the MDA approach can be realized for each industry and its specific needs. For example UML profiles are being defined for the following areas: Manufacturing, Finance, Space, E-Commerce, Transportation, Telecom, Healthcare and more.

3.4 Model Driven Component Implementation

Our approach follows the MDA principles by focusing on a technology independent business model and transforming it as automized as possible into a technology dependent model that is the basis for the implementation artifacts on code level. In the context of component technologies however, we favorize a component-oriented business model. The **component architecture** and **component implementation architecture** in our approach (Figure 9) correspond to the **PIM** and **PSM** of the MDA. Substantially we integrate component-orientation into the MDA approach as this provides a more seamless implementation process in the context of component technologies.

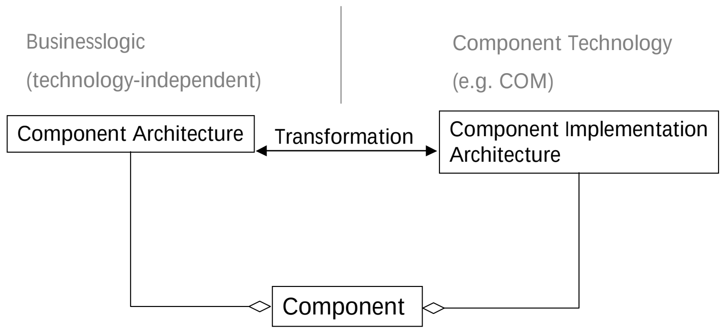


Figure 9: Separating business logic from technology

The component architecture captures the business logic of a specific domain, e.g. a banking system. This engineering activity is technology-independent and focuses on specifying and modeling the businesslogic. It is not the focus of this survey so we will not describe further details here, for a detailed description please refer to [ABB+02], and [CD00].

The component implementation architecture adds technology-specific details to the component architecture. This step is a kind of refinement / specialization towards the concrete technology-infrastructure and it is probable that we require additional modeling elements in order to express this refinement.

Automating the implementation process

At the moment MDA remains vague in some aspects especially when it comes to the details of concrete transformations. Actually this is one of the key issues of this survey - finding out how exactly these transformations look like and how they can be automated in a component-oriented implementation process.

In this survey we build upon the work of Bunse who analyzed the automation of object oriented implementation processes (see [Bun01]). He found that the traditional implementation step should be replaced by two subsequent steps refinement & Translation (Figure 10).

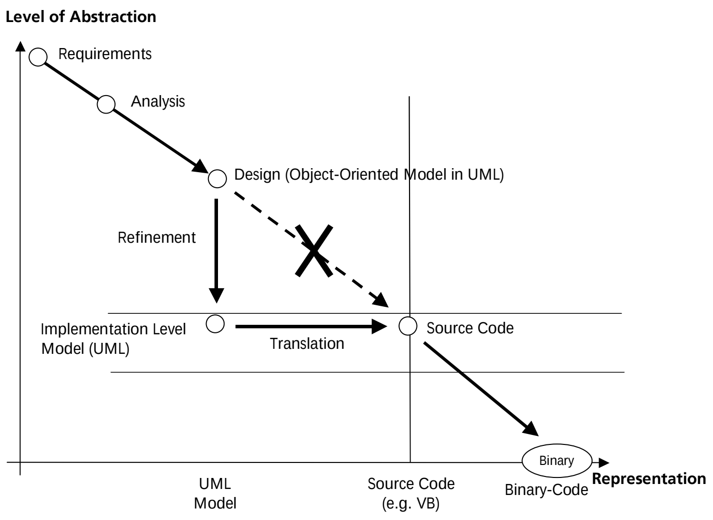


Figure 10: Refinement and Translation Patterns

Refinement transforms a high-level design model (abstraction) into a low-level implementation model (realization) which is closer to the implementation. In general it can be viewed as the relationship between two descriptions at a different level of detail, i.e. the abstraction contains less information than the realization. **Translation** transforms the implementation level model into source code. In contrast to refinement translation describes a relation between two descriptions at the same level of detail, but using different notations.

Both steps can be (semi-)automated using transformation rules called refinement and translation patterns.

Refinement & Translation in a Component Context

We postulated before that the implementation of software components should be based on a component architecture (Chapter 3.2), i.e. a business model that is instantiated from a component meta model. Executing refinements & translations on such a component architecture leads us to the implementation process depicted in Figure 11.

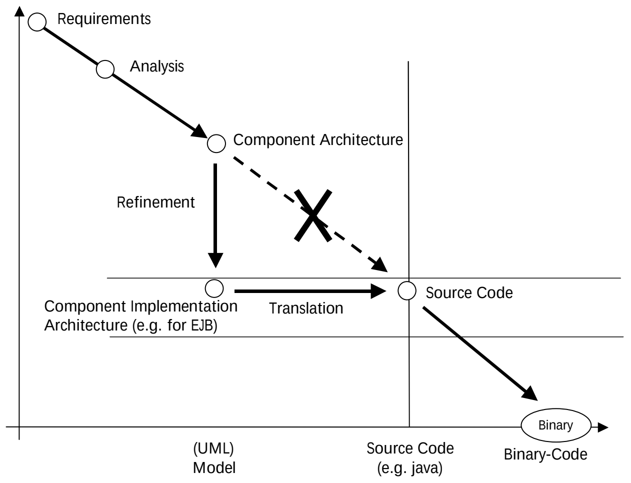


Figure 11: Component-oriented implementation process with refinement & translation patterns

This simplified view of the process and its artifacts will be detailed in Chapter 4 and the case study (Chapter 6). At this point we only want to summarize important results from these chapters: compared to the traditional one (Chapter 3.1) a component-oriented implementation process using refinement & translation patterns has the following benefits :

- **Consistency** : the component paradigm is applied throughout all levels of abstraction.
- **Reuse Components in different systems** : developing software components is quite different from software systems as the focus is on the development of independent components that can be reused in other systems / contexts. This represents already a first step into the direction of system families / product lines that are built up from a pool of reusable components.
- **Productivity & Quality**: increases through the automation of the refinement- & translation steps.

Component refinement as it is described in this survey differs from the refinement discussed by Bunse. The two main differences are that refinement here

- (1) is technology-specific (which is not the case in Bunse's work)
- (2) is not a pure transformation step but contains technical decisions that have to be made, e.g. decisions concerning persistency and transaction support and
- (3) tends to consist of sequences rather than single independent refinements.

3.5 Foundation for Model Transformations

The (semi-) automatic transformation between models due to the MDA approach requires a sound theoretical basis. Therefore we present the key concepts that the OMG developed and recommends for the implementation of the MDA approach.

3.5.1 Summary of the OMG's framework for MDA

The framework that the OMG developed for the realization of the MDA is quite huge. Therefore we can give only a rough description of the involved concepts. The presentation specializes the OMG's work in so far as it puts a specific focus on components.

Meta Models

As we mentioned before, when modeling an application we are using the formal modeling elements of a meta-model. For example class-boxes are a typical modeling element provided by an object meta model like UML.

Component Meta Model

The role of a component meta model is to provide the formal notation and its elements which are required for modeling a component architecture. That is it provides formal elements to describe

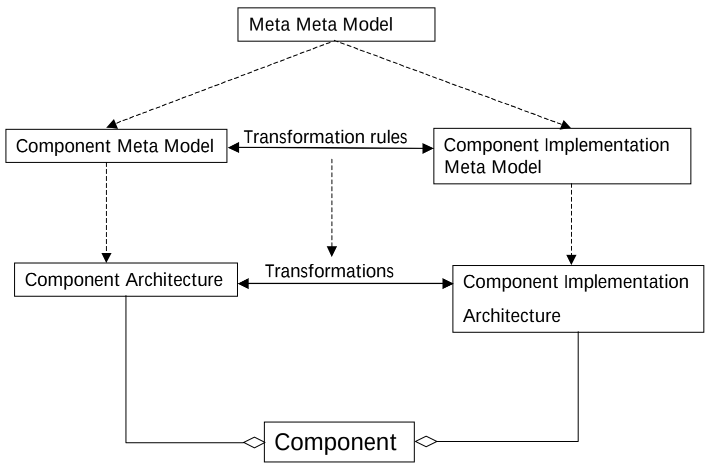
- the parts a component may be constituted from,
- the different types it can be of,
- the structural and behavioral relationships between components.

Transformations

For achieving a systematic and efficient transformation there are some preliminary considerations, see also Figure 12:

The transformation will rely on transformation rules based on the formal elements provided by the meta model. This means we need meta models for both models: an **component meta model** defining the formal elements an abstract component architecture is built from plus an **component implementation meta model** defining the formal elements an component implementation architecture is constituted from. The dotted lines indicate that the meta model is not an input to the model instantiation but rather a constraint which defines which modeling elements are to be used. Reverse transformations from implementation architecture towards abstract architecture may be required, for example because of maintenance activities.

Figure 12:
Meta Models and Transformation Rules



The transformation rules can be made consistent if both meta models are derived from a general concept for meta models. This may seem a bit exaggerated at first glance but we will show in the next section that such a meta meta model is not only existing - the OMG's Meta Object Facility (MOF) - but that it was introduced for exactly this purpose: to facilitate the transformation of models based on different metamodels into each other. The benefit of such a metamodel is that tools can be developed that can automatically transform models into each other.

MOF - Meta Object Facility

The Meta Object Facility defines the core constructs that constitute a metamodel, i.e. a metamodel is modeled by using the MOF core constructs.

The Meta Object Facility can be applied in a variaty of domains and is repository-oriented, i.e. it facilitates tool support. These and other advantages make the MOF and the MDA initiative face a great support within the scientific community. It is expected that future technologies and tool vendors will integrate with the related OMG's specifications. As a characteristic example we can mention the Open Information Model extensively used by the Microsoft Repository and developed by the Metadata Coalition which according to OMG sources [OMG00-B] will merge with the OMG.

UML Profiles

It is quite easy to create MOF-conform meta-models by using the UML profile mechanism. The latter is intended for the purpose of defining a tailored set of modeling concepts together with rules for their use. An UML profile is created as an extension of the standard UML using predefined extension mechanisms like stereotypes, constraints, tag definitions and tagged values (see [OMG01-A], 2-74 to 2-85). As UML is based on MOF a created UML profile is automatically conformant to MOF. If the predefined extension mechanisms are not sufficient, it is possible to extend the UML metamodel by adding new constructs which must be derived from the MOF meta-meta model. This is an optional step to be considered during the creation of a UML profile. Both abstract and implementation metamodels can be described as UML profiles.

Moreover, using the UML enables the easier description of implementation metamodels since the according model elements can be found in the UML metamodel. Consider an interface as an example. Describing an interface with UML is no problem, since the UML specification covers interfaces while the MOF, being more abstract, does not. In Chapter 4 we define a process that describes the creation of a UML profile.

XMI - XML Metadata Interchange

The OMG also specifies how MOF-conformant metamodels can be formalized using a XML-dialect for Metamodel Interchange (XMI). Based on XMI it is possible to develop tools that store, exchange, manage and transform meta models. A subset of XMI, the so-called XMI diffferences are capable of formally describing the differences between models and thus facilitate the transformation between different models. Using XMI-differences, transformations are performed based on three elementary operations: add, delete, replace. In general a transformation can be described as follows:

New Model = Old Model + Differences

In our context the XMI differences will be used to transform abstract models into implementation models:

Implementation Model = Abstract Model + Differences

XSLT

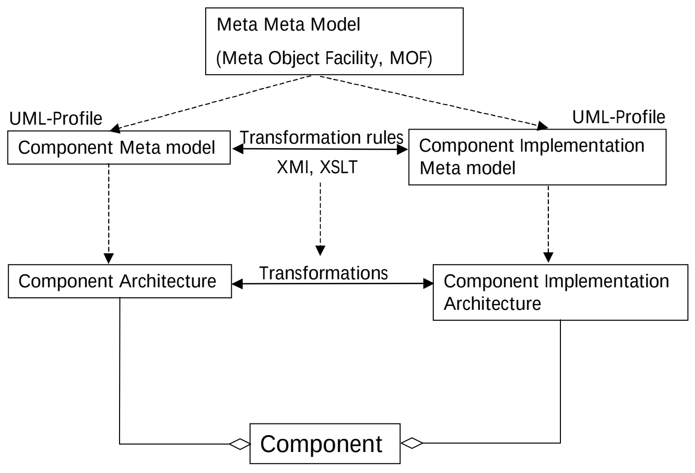
All parts of the above equation are XMI documents. However, instead of XMI differences XSLT could be used. XMI avoids the use of XSLT transformations which are a standard way of transforming XML documents in arbitrary formats including XML itself. The disandvantage of XSLT is its relative complexity which can make writing the code difficult and error-prone. XSLT is necessary in case that the XMI descriptions are to be transformed to formats different than XML.

The main difference between the XMI difference and the XSLT approach is that the former must be used with a decision model. That means that it must be firstly decided what is to be replaced, deleted or added and subsequently these decisions must be described as XMI differences. On the contrary with XSLT decisions or rules can be described directly. XSLT is rich enough to allow the description of the transformation logic as a collection of XSLT scripts. Another point to consider is the tool support which in case of XSLT appears better. We are not aware of any tool that can interpret XMI differences.

Conclusion

Specializing the OMG's MDA approach for component-oriented implementation results in the transformation process depicted in Figure 13.

Figure 13: Transforming models using XMI / XSLT



3.6 Defining a UML profile

In this section we propose a process for the UML profile as depicted in Figure 14.

A UML profile extends the basic UML metamodel using standard extension mechanisms defined in the UML specification. According to [Gre01] it can include one or more of the following:

1 **Standard extensions** beyond those specified by the identified subset of the UML metamodel. A standard extension is an instance of the UML Stereotype, Tagged Value or Constraint metaclasses.

2 **Semantics**, beyond those supplied by the specified subset of the UML metamodel, defined using natural language.

3 **Well-formedness rules**, beyond those supplied by the specified subset of the UML metamodel, expressed as Constraints written in the Object Constraint Language (OCL).

4 **Common model elements**, which are pre-defined instances of UML metamodel elements. The definitions of common model elements may use the standard extensions defined by the profile, and are constrained by the formal and informal semantics defined by the profile.

5 **UML metamodel extensions** created by defining new metaclasses using the Meta Object Facility. UML metamodel extensions should be introduced only when the standard extension mechanisms can not be used to accomplish the desired result, since their use may prevent some tools from reading and writing the resulting models.

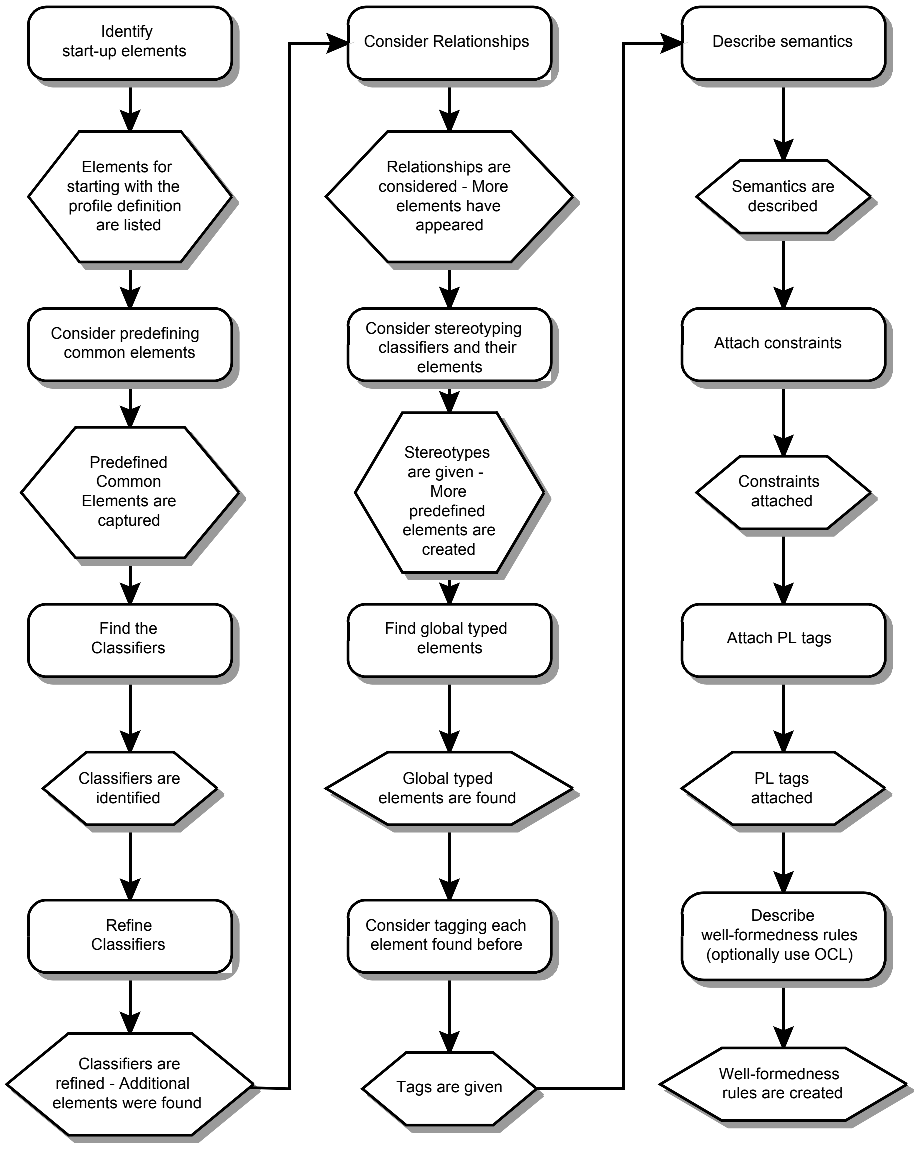

Identify start-up elements
Elements for starting with the profile definition are listed
Consider predefining common elements
Predefined Common Elements are captured
Find the Classifiers
Classifiers are identified
Refine Classifiers
Classifiers are refined - Additional elements were found
Consider Relationships
Relationships are considered - More elements have appeared
Consider stereotyping classifiers and their elements
Stereotypes are given - More predefined elements are created
Find global typed elements
Global typed elements are found
Consider tagging each element found before
Tags are given
Describe semantics
Semantics are described
Attach constraints
Constraints attached
Attach PL tags
PL tags attached
Describe well-formedness rules (optionally use OCL)
Well-formedness rules are created

Figure 14:Process of UML profile creation

For the creation of architectural profiles we propose the use of the **UML Core package** in combination with the **MOF Model package** (in case that meta-model extensions become necessary). For the implementation profiles the **NOF Core package** is more appropriate since its primary goal is to be as close as possible to the implementation and to that end it provides many useful model elements. In any case the process will mainly deal with Name spaces which will be stepwise refined through the use of the packages just mentioned.

The process starts with the identification of the artifacts that will be modelled in the UML profile. These artifacts are model elements of the profile and therefore can be treated as instances of the MOF construct ModelElement. A ModelElement can be one of the following: An Import, a Namespace, a Constraint, a Tag or a TypedElement.

After the model elements have been collected we continue the process with the collection of the name spaces (Namespace construct). which through hierarchical refinement leads to the identification of Packages, Associations, Data Types and Classes as well as Operations and Exceptions. Data Types can be further refined using the hierarchy of the DataType Package of the UML specification.

We deal with the Namespaces first since they can facilitate the identification of tags and constraints. The Import construct have not been considered here but it should be taken into account when the Profile is to import name spaces from another Profile.

As soon as the NameSpaces are collected we define UML Stereotypes for each one of them. We can then proceed with the identification of all other ModelElements. Constraints and Tags should be directly linked to the Name Spaces. TypedElements should also be marked with Stereotypes and connected to a Classifier Namespace, which represents the type of each TypedElement.

In a product line context instances of model elements may be subject to variability. In order to manage the variability information we must firstly specify whether an element can be optional and whether it is allowed to contain optional parts. To this end we can use two predefined common tags “Optional” and “HasVariability”and attach them to each element. If necessary the tagged values can be set to fixed values. This may be the case when an element is never optional or never contains variability. When the profile is applied the modeller will be able to set the according tagged values for all instances of the profile model elements.

Since in a product line setting all variability information can be managed with a decision model it makes sense to connect each namespace with a decision in the decision model. To this end we use two additional predefined common tags “Decision” and “Resolution”. The former is ment to connect a namespace to a decision and the latter to a resolution of this decision that activates the variation point.

The decision model is responsible for providing information about decision dependencies, constraints and possible resolutions. It could be also specified in terms of a UML profile, for example as a Namespace with decisions as Classifiers. Nevertheless dealing with this would exceed the limitations of this paper and is subject of future research.

Custom tags can be also applied as placeholders for configuration management information, like version number or special feature tags. More information on this can be found in the PoLITe report on configuration management [LM03]. Once these tags have been applied in the architectural models they must be taken over in the implementation models as well and this is something to consider therefore during the refinement process.

The process ends with the definition of well-formedness rules that usually are described with the Object Constraint Language.

4 Component Implementation Process

In the last chapter we introduced the concept of a model-driven component implementation. Now we take a detailed look at the process and its artifacts.

4.1 Model Driven Implementation Process and Infrastructure

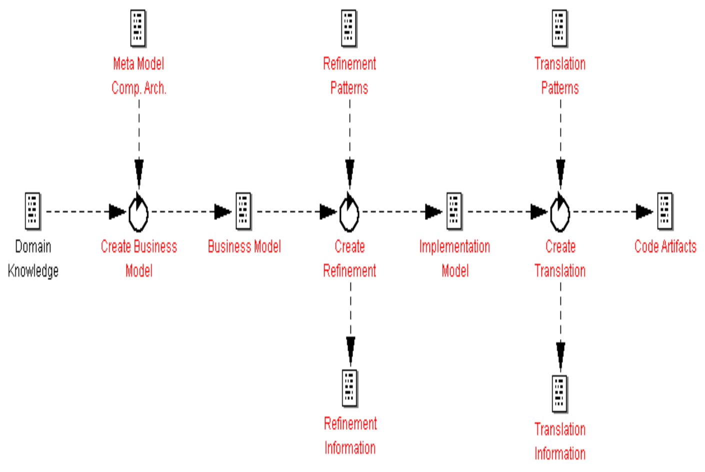
Any process, especially an implementation process requires a well-defined infrastructure that constitues a kind of framework supporting the core activities of the process. Therefore the implementation process can also be characterized as “infrastructure usage”. It is important that the infrastructure can be adapted and changed over time. This is necessary because the implementation process may require changes on the infrastructure , e.g. the used technology may change.

For didactical reasons we begin with the infrastructure usage - demonstrating which elements are required in the infrastructure that follows.

4.1.1 Model Driven Implementation - Infrastructure Usage

An implementation process that follows this concept consists of the process steps & artifacts that are depicted in Figure 15.

Figure 15: Model Driven Component Implementation



1 **Creating The Business Model**: Following a component paradigm the business model is created from domain knowledge using a component-oriented meta model, e.g. the KobrA method. Being instantiated from a component-oriented meta model, the business model constitutes a component architecture. The meta model has to be set up in the infrastructure creation (next chapter). The creation of the business model is not part of the implementation step, but it is depicted here as it provides the crucial input to it.

2 **Refinement**: Implementing a business model in a specific technology implies the execution of a series of **technology-specific decisions**. E.g. in EJB a component can be implemented as session-, message- or as entity-bean - this requires first a decision which alternative is chosen. In our approach refinements are instantiated from patterns that are part of the infrastructure. A refinement pattern is a template for a set of related technological decisions. For example a business component is refined by assigning the pattern “EJB SessionBean“ to it and specifying related decision values like whether the bean’s sessiontype should be “stateful“ or “stateless“. The result is the **Refinement Information** that is stored in relation to the component so that it can be modified and tracked separately. It consists mainly of three informations: (a) the pattern that is used for refinement, (b) a set of related decision-values (e.g. sessiontype = “stateless“) and (c) inherent elements of the technology’s meta model (e.g. Remote Interface). In this survey the latter are also refferred to as **”technology-units”**, see below and Chapter 6.5.1. There is no way to fully automate refinement - the decision making is pure intellectual work specific for each individual component. The **implementation model** is derived from the business model and the refinement information. Chapter 6.4 shows an example, namely an EJB-implementation model in UML.

3 **Translation**: maps technology units (e.g. a Bean-class) into source code artifacts (e.g. a java file). Analogous to refinement, translations are modeled via patterns. In contrast to the component-wise refinement decisions the translation informations that we have found in the case study are not component specific and can be applied generally. However, it may be required in some cases that translation decisions have to be defined component-wise. Based on the implementation model and the translation patterns traditional software artifacts (e.g. java source code) can be generated automatically.

This illustration of the implementation pocess still needs to be refined. This is done in Chapter 4.2, putting a special focus on separation of concerns.

Refinement & translation patterns are described in more detail in Chapter 6.4 and Chapter 6.5 (see also Chapter 3.4).

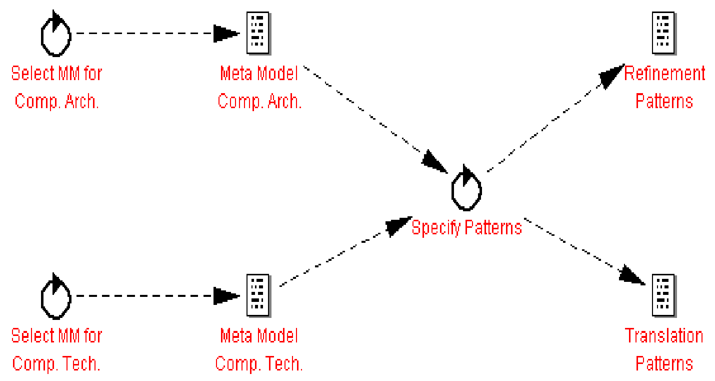
Algorithm Implementation

In a pure model-driven approach even the algorithms used in the methods are modeled as part of the Business Model. In UML this is possible with activity diagrams that are translated into the target programming language via Translation Patterns. From a pragmatical point of view however, this may be not efficient. Keeping in mind that describing and coding algorithms is a highly intellectual work, it may turn out that coding directly in the target language is more efficient than creating a UML description and defining the respective Translation Patterns. Therefore our approach does not oblige to provide models for the algorithms. In contrast, it allows that there is no model for the algorithms and that they are coded directly in the target programming language. These algorithm artifacts should be kept separate from (but related to) the other artifacts (e.g. the implementation model). Source code generation just merges these different artifacts together.

4.1.2 Creating the Infrastructure

In the illustration of the implementation process we already derived and mentioned the constituting elements that an infrastructure has to provide in order to support a model-driven component implementation. Figure 16 depicts the single activities & artifacts for the creation of such an infrastructure.

Figure 16: Create Infrastructure for Model Driven Component Implementation



1 **Select Meta Model for Component Architecture**: The meta model is the basis for the instantiation of the business model. In our component-oriented

approach it has to provide constituting elements that capture components and their relations to each other (see also Chapter 3.4). For the case study we selected the KobrA-method ([ABB+02]). Following a pure MDA-approach the meta-model would be specified as an UML-profile.

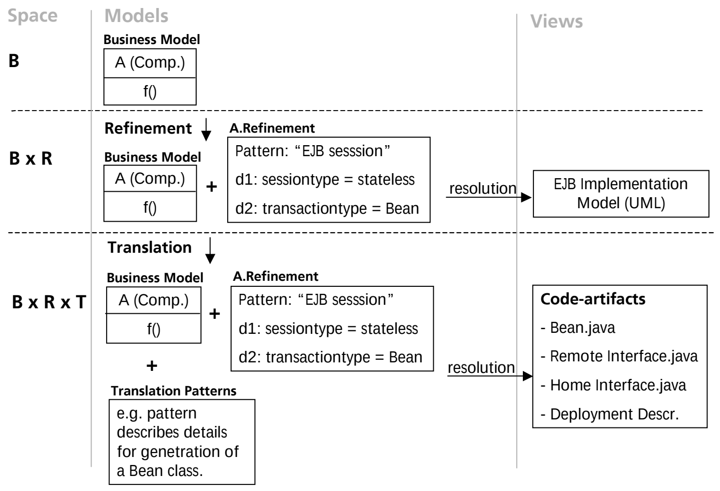
2 **Select Meta Model for Component Technology**: Similarly the meta model for the component technology has to be selected, and again following a pure MDA approach it would be specified as UML profile. In the case study we selected the EJB-technolgy which includes the decision for its meta model.

3 **Specifiy Patterns**: Based on the meta models (steps 1 + 2) refinement and translation patterns can be defined. **Refinement Patterns** describe the transition from business model entities (e.g. components) to implementation model entities (e.g. Session Beans). For example a pattern “EJBSessionBean“ defines that a business component is refined into an EJB SessionBean. In the implementation process this pattern is used and instantiated for a specific component refinement. **Translation Patterns** define transtitions from technology units into source-code artifacts.

4.2 Separation Of Concerns

It is well recognized that separation of concerns in the engineering of software leads to major improvements. Figure 17 depicts the separation principles that we introduce into our approach. The advantages are explained in the following. Please note that we are not referring to separation of concerns as it is discussed in the context of Aspect Oriented Programming (although there may be relations).

Figure 17:
Separating Business Model, Refinements & Translations



Normalization

Keeping the three core-artifacts of our implementation approach separate from each other has the consequence that all change-activities during implementation can be applied, tracked and modified on distinguished artifacts:

- **Business Model** : affected when e.g. new methods are introduced.
- **Refinement** : affected when e.g. technological decisions change.
- **Translation** : affected when e.g. programming guidelines change.

Figure 17 emphasizes the mathematical rationale for this separation : business model, refinements and translations belong to different model-spaces that are orthogonal to each other.

The benefits of this separation correspond to those that are achieved by normalizing a database model, one of the most important being the reduction of consistency problems between the elements.

Model View Controller

Yet another important conclusion is that sourcecode artifacts and implementation models are nothing but **views on the orthogonalized model**. This idea follows the Model View Controller concept. It points out the **main problem of traditional source code**: source code mixes all three model spaces: business model, refinements & translations. And it is exactly this mixing that makes it so hard to tackle problems like consistency. Using a consistent model with normalized model elements can overcome these problems.

4.3 Evaluating the Improvements

A process alone isn’t a guarantor for efficiency or quality improvement. Too many process steps could even worsen efficiency. Therefore we want to line out the most important improvements that are visible in the current state of analysis.

4.3.1 Automation stands for Efficiency & Quality

One outstanding result from the case study is that the amount of information increases dramatically as we move from realization to refinement and from refinement to translation (see case study). Even more important is the fact that the essential refinement or translation information constitutes a relatively small part, most of this information increase is therefore **overhead that can be generated automatically**. This is definitely a rich source for efficiency & quality improvement. Actually we experienced during our case study that creating redundant overhead information manually (using the traditional implementation approach, e.g. editing deployment descriptors) is very error-prone.

In order to tap this potential for efficiency & quality improvement, tools are required that automatically generate the overhead information. Although immature, there are already products on the market that support the MDA approach with automated steps from model to implementation. Assuming that the tools in this area will finally reach a level that they substitute the traditonal handcrafted implementation process, another major improvement can be achieved concerning consistency & traceability.

4.3.2 Consistency & Traceability

The problem with the software artifacts in a traditional process is that they need to be continously synchronized, otherwise they lose consistency over time - which is nearly always the case. For example a UML model was once the basis for implementation and later the implementation was changed in a business aspect. To achieve consistency the UML model needs to be updated with this change information.

An implementation approach like the one we presented in Chapter 4.2 actually needs no synchronization at all because it operates on a normalized model where the constituting elements like the Business Model are directly updated with each change. This drastically reduces the consistency and traceability problems that are present in the traditional implementation approach.

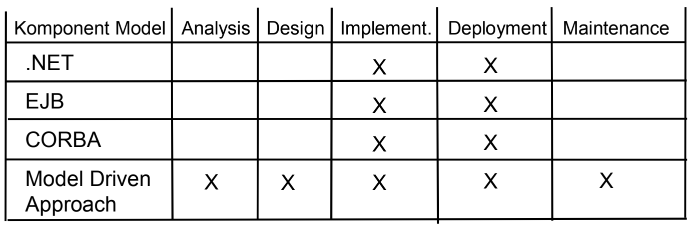
4.3.3 Reducing Reengineering Activities

Traditionally reengineering has to deal with legacy systems where the existence of architectural documentation is the exception. One important reengineering task is therefore the recovery of such architectural documentation. Although there are sophisticated methods supporting this activity, this task is still pretty tedious and intellectual understanding is still heaviliy required. A model-driven approach focuses on the creation of architectural documentation in the form of models. This leverages reengineering because existing models reduce the effort for recovery activities.

4.3.4 Integration & Software-Lifecycle Coverage

The case study indicates that our component-oriented approach can facilitate the integration of component technologies like EJB, .NET and CORBA. This is especially important because these state-of-the-art technologies do not provide seamless integration in the lifecycle by themselves. The presented model-driven approach facilitates the integration of component technologies and consequently manages to cover a big part of the software lifecycle (Figure 18).

Figure 18:
Lifecycle coverage of component-oriented methods & technologies



Komponent Model	Analysis	Design	Implement.	Deployment	Maintenance
.NET			X	X	
EJB			X	X	
CORBA			X	X	
Model Driven Approach	X	X	X	X	X

4.4 Realization of Refinement & Translation Patterns

Refinement and translation patterns form the crucial part of the presented approach. Therefore we developed designs for the realization of refinement & translation patterns that were derived directly from the experiences with the presented case study. The purpose is to demonstrate the convenience with which the theory can be implemented and applied in practice. For further details please refer to Chapter 6.4 and Chapter 6.5.

5 Realization of Software Product Lines

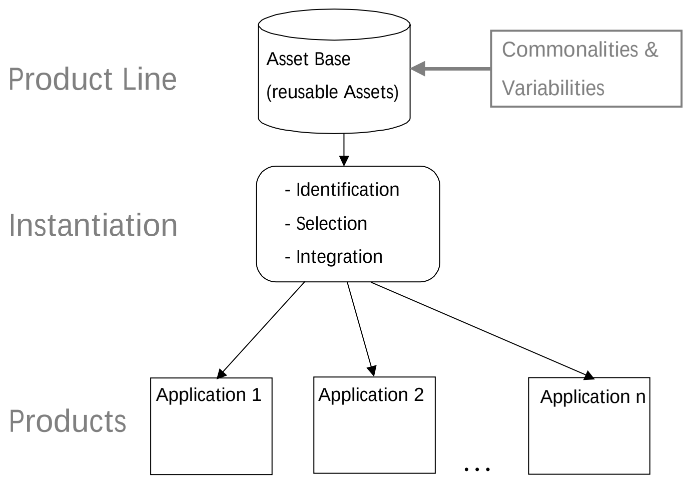
5.1 Principles of Software Product Lines

The software market - just like other markets - has a great demand for variety in products. The entirety of product-variants of one software is also referred to as software system family or software product line.

Manufacturing was the first discipline that provided an answer how to efficiently build varying products. Instead of building single system family members, interchangeable parts were assembled to products ([CE00], p. 3).

Actually the same principle can be transferred to software products, which is depicted in Figure 19. Reusable parts are identified & selected from a pool of reusable assets - the asset base - and integrated into single products, see also [ABB+02], p. 242-243. This step (instatiation) is thoroughly discussed in the PoLITe Report on Configuration Management [LM03].

Figure 19: Software Product-line & Products



Please note that the productline approach is not restricted to implementation artifacts, but also includes analysis and design artifacts ([ABB+02], pp. 242). However, in this document we focus on implementation artifacts. Furthermore it should be clear that the assets under focus in this document are mainly components.

The main motivation for product lines is the reuse of software-assets as it has a significant positive impact on the development costs, effort and quality ([AM01], p. 1).

Commonalities and Variabilities

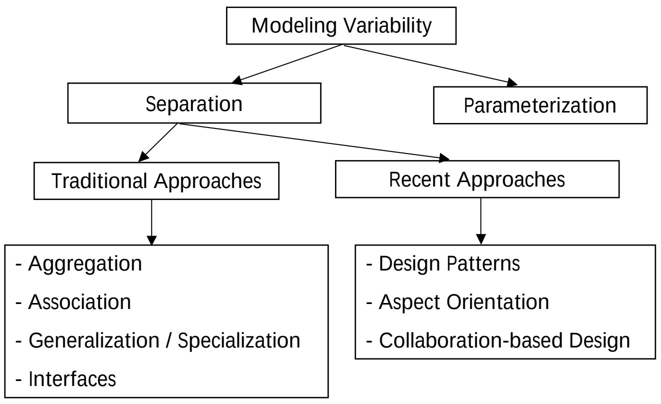
Reuse implies that the asset base contains artifacts which can be reused in more than only one single product. This means that the asset base contains common parts which do not change between product line members and variable parts that feature different functionalities from member to member. Variability can be realized on run-time or development time. Product line engineering is concerned with development time variabilities ([ABB+02], p. 244).

Our focus in the subsequent part is on design techniques that can be applied for modelling product line variabilities (commonality does not require special design techniques). All presented techniques can be applied to components and work on an architectural level which means that they are independent from the later to be used implementation technology or programming language. However, there are differences in how good a given implementation technology or language may support these design techniques.

5.2 Design Mechanisms for Variability

Figure 20 depicts important techniques for modeling variability.

Figure 20: Design Mechanisms for Variability



Parameterization

A first approach to supporting commonalities and variabilities at the design levels refers to parameterization. The underlying idea is to make a component flexible so that it can be tailored according to the context it is put in. To this end the component is equiped with parameters that specify its behaviour. The common parts of the component are not parameterized while the varying parts are.

However, parameterization has its limits when the elements of the component that vary, increase. This is because of the complexity the component must incorporate in order to manage the set of parameters and the value ranges. In such cases it makes sense to split things out. Separation is hence the second category we observe.

Separation

The underlying principle behind separation is divide & conquer which is a time-honoured strategy for handling complexity in science and engineering. It divides a complex problem into parts that can be "conquered" separately. Thus projects can be tackled which otherwise would be beyond the capacity of the human intellect. This becomes even more apparent in a software product line where high complexity and many different concerns are the result of the variabilities among product line members. Separation of concerns at the design level is primarly achieved by decomposition. Separate parts are designed that provide the solutions to parts of the complex problem. Thinking in product line terms one would decompose a complex component so that the common parts are kept together while the varying or variant parts are addressed by different components.

In the next chapter we cover traditional mechanisms that are based on the concept of separating concerns while the last chapter makes a short trip to recently developed, rather evolutionary approaches like aspect-orientation and collaboration based design.

The following approaches have been addressed extensively by the scientific community but mostly in combination with a concrete implementation technology. The PoLITe Report on Programming Languages covers them in more detail [PM02].

5.2.1 Traditional Approaches

Aggregation

Aggregation specifies that a component comprises other components. It is used when we wish to address rather a whole than its parts. This concept supports variability by delegation which means that the whole delegates special requests to the parts which provide the requested service. For example a browser may be requested to print the current frame. The browser will not implement this by itself. Rather it will comprise some kind of printerfacility object will delegate the printrequest to this facility object.

Aggregation can be physical or logical. In the case of **physical aggregation** (also called composite aggregation) the container is solely responsible for the parts. The lifetime of the container is connected to the lifetime of the parts. That means that the container is responsible for creating runtime instances of the parts and eventually for providing the parts with services they need. Moreover at runtime the container is the only composite component that is allowed to contain instances of the specific parts.

A graphical component may for example contain several graphical sub-components. In this case the containment is physical. Another well known example at a higher lever of abstraction represents application servers which provide a runtime environment for components. The core of this environment is a container that hosts runtime instances of components and provides them with technical services like transactions. In this case the common non-functional support implemented by the container is separated from the various functional interfaces implemented by the components.

With **logical aggregation** the aggregate component contains references to the parts. In this case the lifetime of the aggregate is connected to the lifetime of the reference and not of the part itself. For example a text editor component may be said to contain a spell checker while actually it contains only the reference to it since the spell checker code lies in a separate library which may be physically remote to the container and independent from it.

References: [AG01], pp. 111, 114.

Association

As mentioned before aggregation is always combined with delegation when a container forwards requests to its parts. Nevertheless delegation can be also used without aggregation. In this case there is no notion of containment but the notion of an association between components.

Associations come also into play when there is no need for delegating requests but when two components simply depend on each other (i.e clientship, creation relationship). In these cases the user of a service typically holds a reference to the component providing the service. It becomes clear that the concepts of logical aggregation and association are similar and therefore they are often treated as equal.

Generalization / Specialization

Generalization expresses that a type (class or interface) is more encompassing than another type. Specialization is the opposite of generalization. For example "car" and "bycicle" are specializations of the more general concept "vehicle". Generalization / Specialization handle variability in the following way: The generalized object type (supertype) contains the common features, i.e. which are common to all specialized object types (subtypes). A specialization (subtype) then contains the common features plus additional features which represent the variability and which are specific for the subtype. The generalization "vehicle" would have features like max. speed, break, weight. Additional to the common "vehicle"-features, "car" would contain variabilities like tank, gas pedal, wiper.

On implementation level there are different methods how generalization & specialization can be realized:

Inheritance

Inheritance is a concrete way how specialization can be implemented in some object-oriented languages. It should not be mixed with the concept generalization, as the latter could be implemented in many other ways. When using inheritance it is important to distinguish between interface-inheritance and implementation-inheritance (see section Interfaces). For example most component technologies offer **interface inheritance**, where only the behaviour (specified by the interface) is inherited (see [CD00], p. 154), not the implementation. In contrast many object-oriented languages (e.g. C++) offer an **implementation-inheritance** where not only the interface but also the concrete implementation is inherited. This kind of inheritance wires interface and implementation together which makes it difficult to replace the implementation.

Polymorphism

Polymorphism means that there is something in common to several related forms. The different techniques how polymorphism can be supported are rather implementation-specific so we will not discuss them here (e.g. inheritance, overloading, class templates).

References: [Martin95], p.76-97, [Coplien98], p.133-134.

Interfaces

Interfaces are an essential concept of components which can be used for separating concerns. They separate a component into functional facets.

Interfaces have several benefits, one of which is that they explicitly separate the external behaviour provided by the interface from the internal implementation.

Thus, it is possible to replace/vary the internal implementation according to the given settings. Another consequence of this separation is that users can concentrate on the behaviour and the integration of the component. They need not be concerned about the implementation details which are completely hidden.

5.2.2 Recent Approaches

Design Patterns

Design Patterns as described by Gamma [GAL+95] also provide means of parameterization and separation and have proven to be beneficial for solving recurring design problems. Some of them can certainly be applied for the design of variabilities. For example the Bridge Pattern [GAL+95] can be used to model variability by separating an interface from its implementation. Or the Broker Pattern [Bus96] can be used to achieve location transparency enabling a component to change its physical location without affecting any clients.

Aspect-oriented Design

While the previously described techniques are means for achieving separation of concerns, Aspect Orientation directly reflects the ideas behind that. Aspect Orientation is about engineering (cross-cutting) concerns. Aspect-oriented design deals with the identification of concerns and with the decision of which concerns to separate or to encapsulate at the design level. The different cross-cutting concerns are mapped to aspects or aspectual components of the system under development. They are ment to integrate their functionality into other system components when it is required and without the need to change the other components. As a result of this general description of concerns at the design level, the approach enables several kinds of aspect materializations through different frameworks. A typical object-oriented framework for example would suggest objects for the implementation of the aspectual components.

Collaboration-based Design

The main idea behind collaboration-based design is that a system functionality is realized by a set of components that collaborate. Each component plays a role in this collaboration. The roles are like contracts a component must fulfill if it wants to participate to the collaboration. This means that a component can replace another component of the collaboration as long as it plays the required role correctly.

5.3 Component Implementation in a Product Line Context

The preceding chapters (3 & 4) show how the implementation process can be improved by the consistent application of the component and MDA paradigma throughout the whole product lifecycle. So far this approach focuses on single systems.

For product lines the focus shifts now from single systems to system families. Therefore the presented approach has to be extended so that it integrates the implementation of system families. Emphasis here is on "Extension" which means that all principles that we have developed so far (e.g. refinements & translations) are still valid for the implementation of system families, it is just that product lines bring in an additional dimension that we treat in the following as kind of orthogonal to the already existing ones.

There are **2 possibilities** how to extend the implementation process:

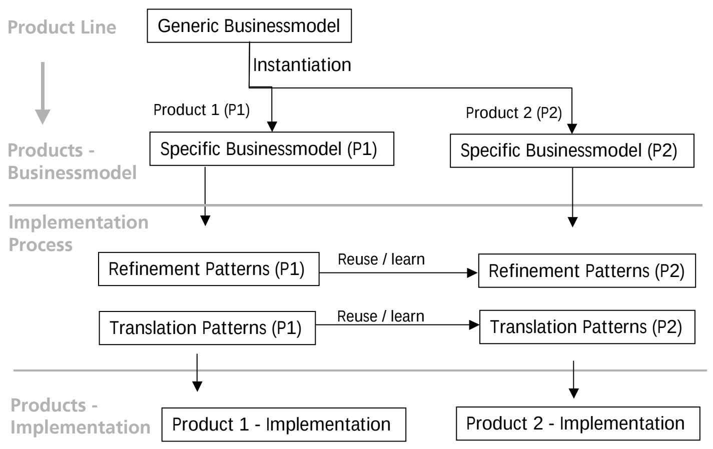
- Preliminary Model Instantiation
- Parallel Model Instantiation

Preliminary Model Instantation

This approach favors a total separation of product line related activities from the presented implementation acitivities (i.e., refinements & translations). This is achieved by instantiating the Business Model for the single system from a generic Business Model **before** any implementation step is executed (see Figure 21). The resulting single system Business Model constitutes the input for the implementation process following the presented component & MDA-paradigm. With this clear separation the implementation process need not be changed in its process steps and artifacts. For each product refinements and translations are defined. The final product implementation is obtained via generation.

Concerning the product-line, a **"learning-curve" effect** can be applied if other products of the product line are implemented after the first product. In this case already existing refinement & translation patterns for product 1 can be reused to derive analogous patterns for product 2 (see Figure 21).

Figure 21:
Preliminary Model Instantiation



The Business Model has to be based on a Meta Model that not only covers components but also **genericity**. The latter comes in because the Business Model is describing not a single system but a system family . The method presented in [ABB+02] inlcudes a Meta Model that integrates component-orientation as well as genericity.

Parallel Model Instantiation

Parallel Model Instantiation defers the model instantiation to “later“, i.e. variabilities are not resolved but kept during the implementation process. Typically this is done by writing **generic code-artifacts** that are resolved at the very end of the implementation process into product specific code. For further details on this topic please refer to the PoLITe Report on Programming Languages [PM02]. It describes the techniques for writing & instantiating generic code artifacts.

6 Case Study

In this chapter we examine in detail the component-oriented implementation process and its related artifacts by executing a simple case study. The business logic is modeled using the KobrA-method - a component meta model suggested in [ABB+02]. A special focus is on the refinement & translation activities that accompany the implementation process.

6.1 Comments on Business Modeling with KobrA

The KobrA-method is not the topic of this survey. But as it provides the business model which is the crucial input for the implementation phase, we consider it necessary to mention at least the following steps:

Specification: The specification focuses on the “outside view“ of components and their services. This is achieved by describing each component X in the system and the services it provides to the world outside (i.e. other components). Internal details are not part of the specification. For example algorithms or additional datastructures that are used to implement services are not described here.

Realization: the realization adds all internal details to the specification. E.g. Classes that are used internally for the realization of the services have to be defined here. The component-realization is the last artifact that is technology independent. All following steps constitute “implementation steps“ as they are executed in the context of a specific technology.

Specification & Realization are subsequent activities of the Business Modelling process, i.e. what we depicted as “Create Business Modell” in Figure 15 is actually subdivided in KobrA in two subsequent steps “Create Specification“ & “Create Realization”.

Flattening: Some technologies (e.g. COM) do not support runtime loading of fine-grained components efficiently. This means that loading a huge amount of fine-grained components has a bad performance. In these cases flattening is required, i.e. fine-grained components are combined to more coarse-grained components. This reduces the amount of physical components and leads to a better performance. Flattening guidelines help to answer the question: “How to organize the architectural components in physical entities?”. Physical entities can be libraries, executable files, assembly files or the implementation components themselves. In this survey flattening was not relevant, as the used EJB technology doesn’t require it. For more information on flattening please refer to [ABB+02].

6.2 Introduction to the Library System Case Study

The example system used in the case study is a Library System that allows a user to loan and return Items. In order to keep our focus on the implementation process and the related activities we modeled the system as simple and minimal as possible. The Library System allows users to loan and return Items.

6.2.1 Library System Specification

The specification (Figure 22) tells us WHAT the component Library System is expected to do. Items (e.g. books) can be loaned, an account is used to authorize a user to loan Items at all. Loans capture the Items that are loaned by a certain account plus the resepctive creation- and returnDates.

The stereotype <<Komponent>> specifies that the Library System component models behaviour. I.e. the Library Systems interface offers methods that support the typical processes needed in a Library System. On the other hand we have (KobrA) entities like Items, Accounts and Loans. The difference is that entities capture persistent components, i.e. their focus is more on the storage of the component’s attributes rather than behavioural aspects.

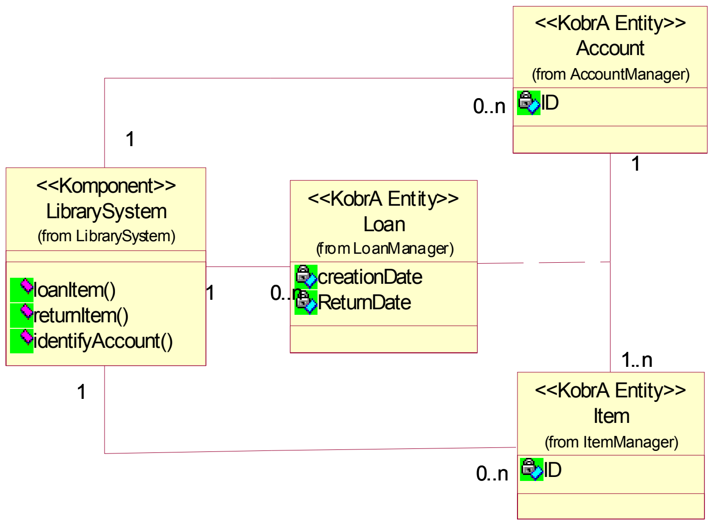
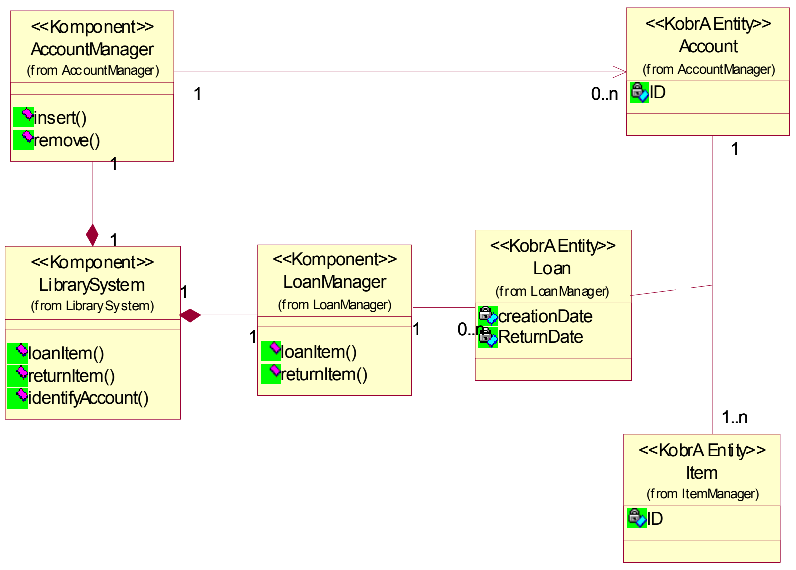


Figure 22: LibrarySystem.Specification

6.2.2 Library System Realization

Whereas the specification focuses on the outer structure of the Library System, the Realization focuses on the inner structure, i.e. details concerning the realization are added (see Figure 23). At a first glance we can see that inside of the Library System additional objects are required. A LoanManager is required for handling loaning-activities like the creation (loaning) and removal (returnItem) of Loans. An AccountManager is required for handling account-activities like the identification of a UserAccount. An ItemManager handles Item-related acitivties like checking if an Item is at all in the Library System (getItem).

Figure 23: LibrarySystem.Realization

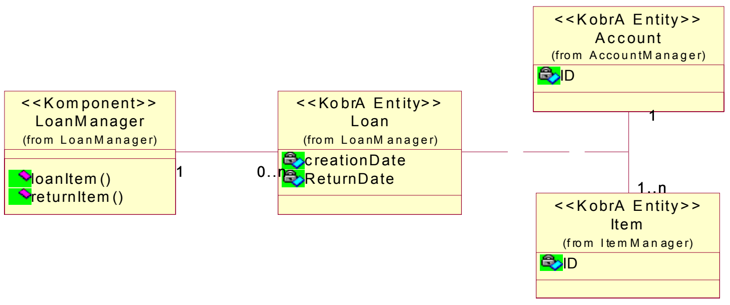


6.3 Business Model of the LoanManager Component

The KobrA approach is a fractal approach in the sense that all activities which have been executed for the Library System as the top-level component of the system are executed just the same for the lower level components of the system. In the following we analyze the whole implementation process with Refinements & Translations focusing on one the LoanManager component.

6.3.1 LoanManager Specification

Figure 24: LoanManager.Specification



The specification shows the LoanManager's services and the Businessobjects he is dealing with that are relevant to the outside world, i.e. for collaborating components like the library system. Relevant Businessobjects are Items, Accounts and Loans. The LoanManager provides the following services (interface):

- **setAccount (AccountId)**: registers a specific account on the LoanManager. Following actions like loaning Items are executed for this account.
- **loanItem (ItemId)**: the specified Item is loaned for the Account that has been set before
- **returnItem (ItemId)**: the specified Item is returned.

6.3.2 LoanManager Realization

The realization (Figure 25) enriches the model with a new object. The LoanStore is introduced for the creation (insert) and removal (remove) of Loans plus the information if a requested Item is already loaned (isLoaned). The LoanStore is not required for the communication with other components and therefore it is not visible in the specification.

Containment

The containment relationship defines the nesting of components within one another (see [ABB+02],pp. 119). Thereby it defines which components are packaged into one component. In our case, the LoanManager contains three : **LoanManager, LoanStore and Loan**. The decision that LoanManager contains Loan results from the fact that the LoanManager needs direct access to the Loan whereas it is not required directly by other components.

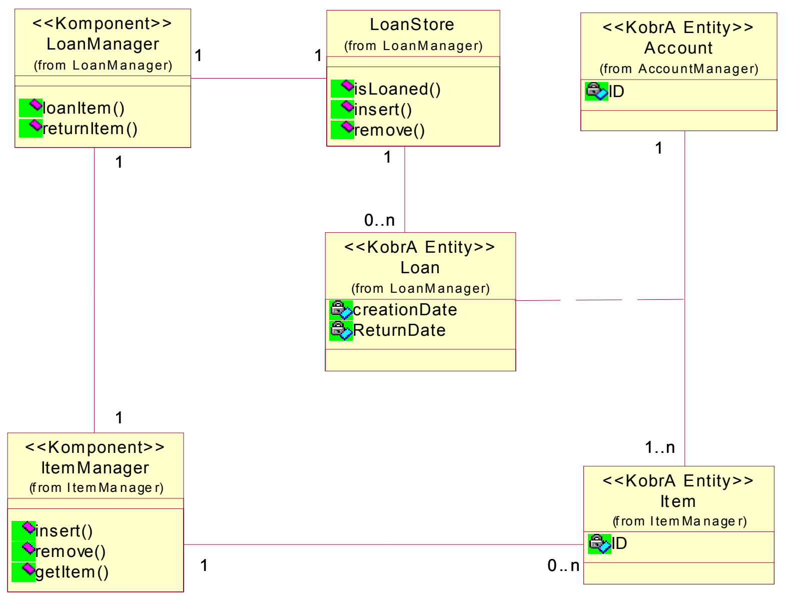


Figure 25: LoanManager.Realization

6.4 Refining the LoanManager Component

6.4.1 Analyzing the Refinements

In the following we derive the refinement information by comparing the LoanManager's implementation model with the corresponding business model. The purpose is to extract only refinement information that is intellectual work. Any other information that can be generated automatically is out of scope. On this base we can analyze and comprehend the underlying refinement patterns.

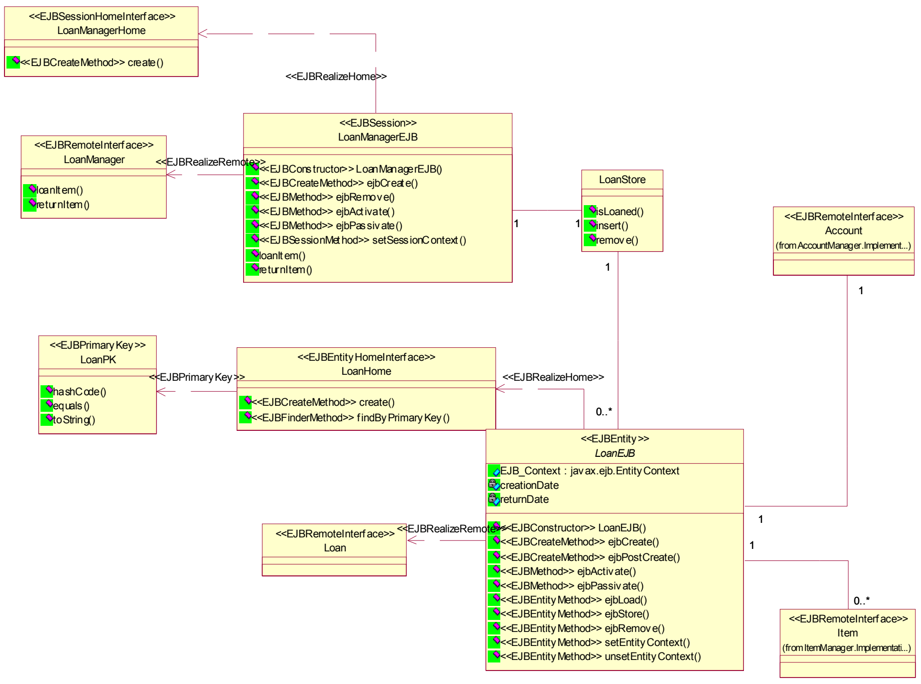
LoanManager EJB Implementation Model

Figure 26 shows the LoanManager's implementation model for EJB in UML. It is the graphical representation of the implementation (code) which is the result of a developer's intellectual work during the implementation process.

Information Overhead

First thing that strikes our attention is that the information has increased dramatically compared to the size of the business model. Another observation that cannot be drawn directly from the pictures is that the amount of actual refinements, i.e. the related intellectual technology decisions that led to the implementation model constitute only a minor part. This means that the bigger part of the information increase is overhead that could be generated automatically.

Figure 26: LoanManager.Implementation


<<EJBSessionHomeInterface>>
LoanManagerHome
<<EJBCreateMethod>> create()
<<EJBRealizeHome>>
<<EJBRemoteInterface>>
LoanManager
loanItem()
returnItem()
<<EJBRealizeRemote>>
<<EJBSession>>
LoanManagerEJB
<<EJBConstructor>> LoanManagerEJB()
<<EJBCreateMethod>> ejbCreate()
<<EJBMethod>> ejbRemove()
<<EJBMethod>> ejbActivate()
<<EJBMethod>> ejbPassivate()
<<EJBSessionMethod>> setSessionContext()
loanItem()
returnItem()
1
1
LoanStore
isLoaned()
insert()
remove()
1
<<EJBRemoteInterface>>
Account
(from AccountManager.Implement...)
1
<<EJBPrimaryKey>>
LoanPK
hashCode()
equals()
toString()
<<EJBPrimaryKey>>
<<EJBEntityHomeInterface>>
LoanHome
<<EJBCreateMethod>> create()
<<EJBFinderMethod>> findByPrimaryKey()
<<EJBRealizeHome>>
0..*
<<EJBEntity>>
LoanEJB
EJB_Context : javax.ejb.EntityContext
creationDate
returnDate
<<EJBConstructor>> LoanEJB()
<<EJBCreateMethod>> ejbCreate()
<<EJBCreateMethod>> ejbPostCreate()
<<EJBMethod>> ejbActivate()
<<EJBMethod>> ejbPassivate()
<<EJBEntityMethod>> ejbLoad()
<<EJBEntityMethod>> ejbStore()
<<EJBEntityMethod>> ejbRemove()
<<EJBEntityMethod>> setEntityContext()
<<EJBEntityMethod>> unsetEntityContext()
<<EJBRemoteInterface>>
Loan
<<EJBRealizeRemote>>
1
1
0..*
<<EJBRemoteInterface>>
Item
(from ItemManager.Implementati...)

.

Refining LoanManager into an EJB SessionBean

The component LoanManager is implemented as a SessionBean. Sessionbeans are used for components with a “session“-restricted lifetime. The LoanManager is such an object: it is only required if a client wants to loan an Item. As soon as the client disconnetcs, the EJB Container may destroy the LoanManager. The decision for a sessionbean always leads to three classes (i.e. this part of the pattern can be automated):

- **RemoteInterface**“LoanManager”, stereotype <<EJBRemoteInterface>>, specifies the interface to the outside world.
- **SessionBean**“LoanManagerEJB“, stereotype <<EJBSession>>, implements the operations provided by the interface.
- **HomeInterface**”LoanManagerHome”, stereotype <<LoanManager-Home>>, is an object factory, i.e.it is responsible for the creation and desctruction of the sessionbean.

There are two decisions that we have to make when implementing a Session-Bean.

1 **Sessiontype**: two values are possible : stateful or stateless. The sessiontype **stateful** is used when the Bean is designed to service multiple requests / transactions so that it has to track a state. As it is not required in our case to do a sophisticated tracking, we model it as **stateless**.

2 **Transactiontype**: two values are possible: Bean or Container, chosen was "Bean".

Refining Loan into an EJB EntityBean

KobrA Entities, i.e. objects that need to be persistent in the system, are implemented as Entitybeans. In our case the Loanobject is modeled as an Entity Bean. An Entity Bean consists of 4 elements(Primary Key depends on decision):

- **RemoteInterface** "Loan", stereotype <<EJBRemoteInterface>>
- **EntityBean** "LoanEJB", stereotype <<EJBEntity>>, implementing the Beans business methods (in the case of an entity bean there would'nt be much),
- **HomeInterface** "LoanHome", stereotype <<EJBRemoteInterface>>
- **PrimaryKey** "LoanPK", stereoptype <<EJBPrimaryKey>>

The following decisions have to be resolved when implementing an Entity Bean:

1 **Type of Persistence**: EJB distinguishes two types of persistence, **Container Managed Persistence (CMP)** and **Bean Managed Persistence (BMP)**. We chose CMP as it is the easiest way to make a Bean persistent.

2 **Primary Key**: it is possible to model the primary key as an object of its own. This is only required if the key is a composition of different properties. In our case we used a unique property and therefore did not need this extra class.

3 **Reentrant/Mulithreading**: An Entity that is not supposed to be shared between multiple threads is modeled as reentrant = false. A Bean is reentrant if one of its instances can be shared between several threads. For our simple example we chose false.

Remark: Besides from using EJB-persistence mechanisms we could also decide here to use another persistence framework, i.e. a different framework that is not provided by EJB.

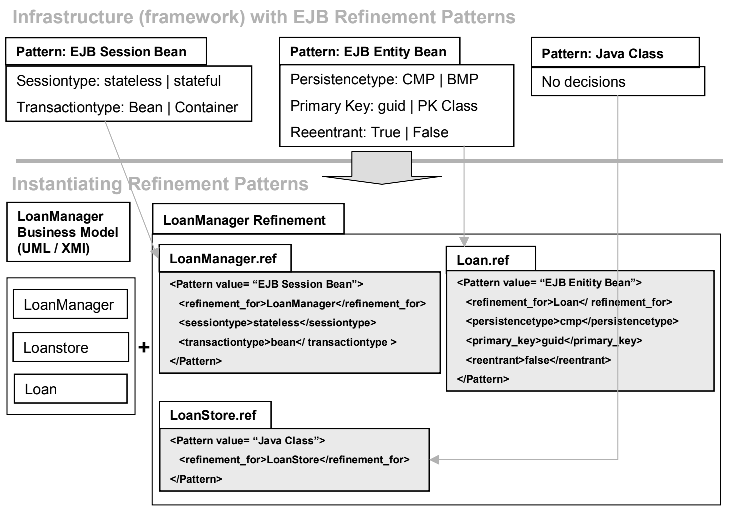
Refining LoanStore into a simple java class

The functionality of Loanstore is only required inside the LoanManager. It is neither visible in the LoanManagers interface nor does it contain persistent attributes. Therefore we implemented it as java class, the simplest refinement possible.

There is no further decision required.

6.4.2 Formalizing Refinements with Refinement Patterns

Figure 27 depicts that refinement is the instantiation of a pattern, i.e. a component of the business model is refined by assigning a refinement pattern to the component and then resolving the respective decisions. These are resolved by assigning values to decision variables. As a prerequisite a whole framework of refinement patterns is required, each pattern describing a technology refinement and the relevant decisions. This framework is then used during implementation performing refinements.


Infrastructure (framework) with EJB Refinement Patterns
Pattern: EJB Session Bean
Sessiontype: stateless | stateful
Transactiontype: Bean | Container
Pattern: EJB Entity Bean
Persistencetype: CMP | BMP
Primary Key: guid | PK Class
Reeentrant: True | False
Pattern: Java Class
No decisions
Instantiating Refinement Patterns
LoanManager Business Model (UML / XMI)
LoanManager
Loanstore
Loan
+
LoanManager Refinement
LoanManager.ref
Loan.ref
LoanStore.ref

```
<Pattern value= "EJB Session Bean">
  <refinement_for>LoanManager</refinement_for>
  <sessiontype>stateless</sessiontype>
  <transactiontype>bean</ transactiontype >
</Pattern>
```

```
<Pattern value= "EJB Enitity Bean">
  <refinement_for>Loan</ refinement_for>
  <persistencetype>cmp</persistencetype>
  <primary_key>guid</primary_key>
  <reentrant>false</reentrant>
</Pattern>
```

```
<Pattern value= "Java Class">
  <refinement_for>LoanStore</refinement_for>
</Pattern>
```

Figure 27: Performing Refinement by instantiating Refinement Patterns.

Please note that the business model is not transformed or thrown away in this approach. Refinement is reduced to the minimal delta information that specifies how the business model is to be implemented in a specific technology.

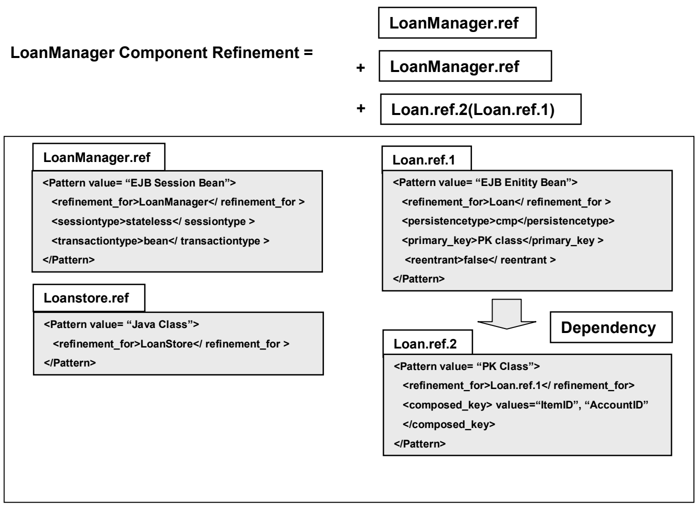
Refinement Sequences

As we have seen, the refinement of the LoanManager component actually consists of a whole set of refinements rather than a single one. One reason for this is that the component LoanManager contains three model elements that need to be refined, each requiring a specific refinement: LoanManager, LoanStore and Loan. Also, the refinement of a model element may require the sequential application of patterns. This leads us to refinement sequences.

Refinements are not necessarily independent, rather they can depend on each other. E.g. the relationship between two model elements A and B is to be refined. But the relation refinement depends on whether A/B is refined as java class or as SessionBean. Therefore the refinements have to be executed in an order - the relation refinement can only be done after the refinement of A and B.

Another type of dependency exists if the refinement of one model element requires the sequential application of several patterns, illustrated in Figure 28. In this example the refinement of the LoanManager differs from the preceding examples. The difference is that the primary key is not modeled as a single guid-attribute but as a Primary Key class (PK class). This requires an additional refinement that specifies the composition of the key, in our example the key is composed from the attributes "ItemID" and "AccountID".

Figure 28: Refinement Sequences & Dependencies



```
<Pattern value= "EJB Session Bean">
  <refinement_for>LoanManager</ refinement_for >
  <sessiontype>stateless</ sessiontype >
  <transactiontype>bean</ transactiontype >
</Pattern>
```

```
<Pattern value= "Java Class">
  <refinement_for>LoanStore</ refinement_for >
</Pattern>
```

```
<Pattern value= "EJB Enitity Bean">
  <refinement_for>Loan</ refinement_for >
  <persistencetype>cmp</persistencetype>
  <primary_key>PK class</primary_key >
   <reentrant>false</ reentrant >
</Pattern>
```

```
<Pattern value= "PK Class">
  <refinement_for>Loan.ref.1</ refinement_for>
  <composed_key> values="ItemID", "AccountID"
  </composed_key>
</Pattern>
```

The Refinement of the LoanManager Component consequently consists of four refinements : LoanManager.ref, LoanStore.ref and Loan.ref.1 and Loan.ref.2, the numbers indicating the sequence in which they have to be applied.

Loan.ref.1 specifies that a Primary Key Class should be used (primary_key value="PK Class"). Loan.ref.2 depends on this first refinement as it specifies the attributes that are combined to form the primary key. This is a clear dependency: the Refinement using Pattern "PK Class" can only be defined after the Refinement using Pattern "EJB Entity Bean" has been specified.

Selection of technology units

We have already seen a case where decisions actually resemble the **selection of technology-units**. The decision that a Primary key class is to be used is one. Another example are Session Beans where instead or in addition of remote interfaces so called local interfaces can be implemented (local interfaces can be applied in non-remote cases for better performance, see [AT02]). The selection of certain technology units is intellectual work (e.g. for performance reasons) that needs to be recorded in the refinement information and therefore is also part of the refinement pattern. Furthermore technology units are the key units for the definition of translation patterns (next chapter).

The examples that we provided so far are only outlinging the refinement idea . However, the simple examples are already evidence for the complexity of the topic due to implied refinement sequences and selection of technology units.

6.5 Translating the LoanManager Component

6.5.1 Analyzing the Translation

We proceed here like we did with the refinement patterns. I.e. first we present the codelevel artifacts that are the result of a traditional implementation. Then we compare them to the preceding artifacts (meta model of component technology, business model, refinements) resulting in a set of translation rules that describe the translation and can be grouped in a pattern. The translation principles that we identify in the following (see Table 1) are the same for all source code artifacts so that it is sufficient to present only the sourcecode of the Remote Interface.

Table 1:
Source Code for Remote Interface of LoanManager Component

```
/*
 * LoanManagerRemote.java
 *
 * Created on 3. Mai 2002, 10:39
 */

package LoanManager;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

/*
This interface defines the `Remote' interface for the `LoanManager' EJB.
*/

public interface LoanManagerRemote extends EJBObject  {
    public String loanItem(String id)  throws RemoteException;
    public String returnItem(String id) throws RemoteException;
}
```

6.5.2 Formalizing Translation with Translation Patterns

Translation Patterns are meta structures but they differ in two important aspects from Refinement Patterns (see also Chapter 3.4):

1 **Mapping**: Translation transforms the implementation level model into source code artifacts, i.e. it describes a mapping between two descriptions at the same level of detail, but different notation. In contrast, Refinement adds low-level technological information to a high level model.

2 **Unit-wise**: There are no decisions that have to be resolved component-wise. This means that the Translation Pattern can be applied directly to the Implementation Model. In contrast, refinements require the component-specific assignment of Refinement Patterns and their decision resolving.

The following example shows a possible translation pattern realized as a XML-file. A full implementation would need some more details to be specified (e.g. signature of the methods). Also, a pure MDA approach would prefer XMI instead of XML.

Table 2: Instantiated Translation Pattern for the Implementation of an EJB-Remote Interface.

```
<?xml version="1.0" encoding="UTF-8"?>

<translation_pattern value=”remote_interface_to_java”>
    <header_comment>created on</Header>
    <package>component_name</package>
    <imports>
      <import value=”javax.ejb.EJBObject”/>
      <import value=”javax.rmi.RemoteException”/>
    </imports>
    <interface_definition>
       <public>public interface</public>
       <interface_name>component_name</interface_name>
       <throws>throws RemoteException</throws>
       <methods>component_methods</methods>
    </interface_definition>
</translation_pattern>
```

Applying the Translation Pattern

In practice the translation is executed by a generator that interprets the contained mapping information. The **bold-marked** information in the pattern is derived from the business model. The Pattern is interpreted as follows:

- **<header_comment>** : specifies that the comment “created on” is to be inserted into the source code
- **<package>** : specifies that the name of the component is used as pacckage name.
- **<imports>**:lists the default packages that are to be imported for EJB Remote Interfaces.
- **<interface_definition>**: lists all information required for the interface definition.
- **<interface_name>** : “component_name” specifies that the component’s name from the business model is taken as interface name.
- **<throws>** : defines “RemoteException“ as the default Exception for Remote Interfaces.
- **<methods>** : the “component_methods” from the business model are used.

The demonstrated translation of the LoanManager’s Remote Interface with a translation pattern has the following characteristics:

- **Transformation of Technology-Units**: the pattern contains a mapping description for elements of the meta model - in our terminology technology units. In the example the EJB technology unit "RemoteInterface" is mapped into java code.

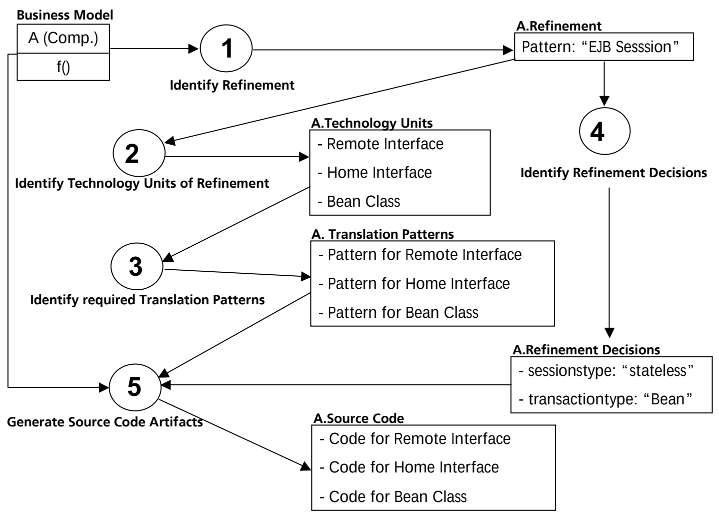
- **Generic Pattern with defaults**: for the translation of each technology unit a generic pattern with "parts" (e.g. <header-comment> or <import>) and corresponding default values (e.g. "Created on", "javax.ejb.EJBObject") can be defined.
- **Component information is retrieved from business model**: the information that is component-specific, e.g. the interface name, the interface methods have to be extracted from the business model.
- **High Automation Rate** : In the example there is no intellectual interference required for the translation. Although we cannot exclude cases where such an interference is required, the achievable automation rate is promising.

6.5.3 Automatic Generation of Source Code Artifacts

In our approach the component-oriented implementation model that needs to be translated into source code artifacts is actually divided / normalized into several elements: business Model, refinements (incl. patterns) and translation patterns. Here we outline the automated process that uses these elements and generates the respective source code artifacts.

Figure 29 depicts the relevant steps, using a simple business example - a component A that provides only one service f() and that does not contain other components or objects.

Figure 29:
Translation Process


Business Model
A (Comp.)
f()
1
Identify Refinement
A.Refinement
Pattern: "EJB Sesssion"
2
Identify Technology Units of Refinement
A.Technology Units
- Remote Interface
- Home Interface
- Bean Class
4
Identify Refinement Decisions
3
Identify required Translation Patterns
A. Translation Patterns
- Pattern for Remote Interface
- Pattern for Home Interface
- Pattern for Bean Class
A.Refinement Decisions
- sessionstype: "stateless"
- transactiontype: "Bean"
5
Generate Source Code Artifacts
A.Source Code
- Code for Remote Interface
- Code for Home Interface
- Code for Bean Class

The single activities of the process are:

1 **Identify Refinement**: First of all, the refinement that has been specified for A is identified. In our example the Pattern is simply “EJB Session“, i.e. A is to be implemented as Session Bean.

2 **Identify Technology Units in Refinement Pattern**: The translation is performed for technology units (elements of the technology meta model), e.g. a Remote Interface. This information is provided by refinement. In the example the refinement of component A provides the translation with the information that three technology units are to be created : (a) Remote Interface, (b) Home Interface and (c) Bean Implementation.

3 **Identify Required Translation Patterns**: The translation of each technology unit is defined in a specific translation pattern. For each technology unit the corresponding translation pattern has to be identified.

4 **Identify Refinement Decisions**: the decisions that were made during refinement are also required for the code generation.

5 **Generate Source Code Artifacts**: The final translation generates the source code artifacts, using three separate informations: (a) a specific translation pattern for each of the technology units, (b) component information of the business model, e.g. interface name and method signatures, (c) refinement decisions, e.g. that the Bean’s sessiontype is “stateless“.

Conclusion

There are some important conclusions from this part

- **Source code is a view on a normalized model**: The process depicted in Figure 29 demonstrates once more the fact that source code is a mixture of different informations. In our approach we keep them separate in a normalized model and derive the source code as a view on this model.
- **Refinement includes decisions regarding the selection of meta model elements**: refinement implies technology units and therefore should keep explicit knowledge about these meta model elements. Although there are patterns where the implied technology units are not likely to change, there are enough examples showing that this information is component specific, depending on the developer’s refinement decisions. This documents that they are just a special kind of decisions. For example in the case of container managed entity beans the existence of a primary key class depends on the decision of the developer and therefore has to be part of the component refinement.

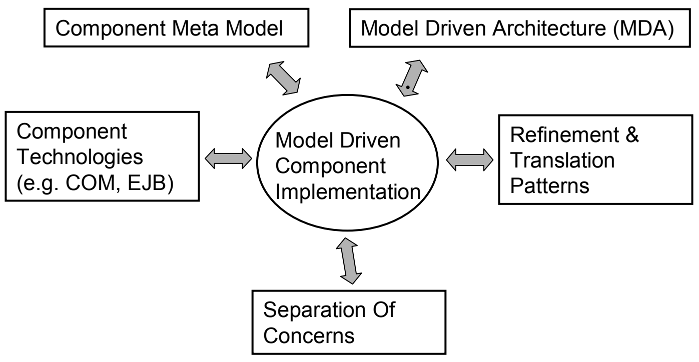
7 Summary and Outlook

This report dealt with one aspect of implementation technologies for software product lines: the component technology view.

7.1 Summary

This survey presented an approach for facilitating the implementation of product lines using component technologies. It integrates concepts from several areas such as model-driven architecture, component-based development, pattern usage and separation of concerns (see Figure 30).

Figure 30: Principles & Techniques in the Component Context.


Component Meta Model
Model Driven Architecture (MDA)
Component Technologies (e.g. COM, EJB)
Model Driven Component Implementation
Refinement & Translation Patterns
Separation Of Concerns

The benefits/improvements that can be accomplished with this approach are:

- **High Automation Rate:** Large parts of the presented implementation process are designed for automation.
- **Efficiency and Quality:** The approach integrates several features that support efficiency and quality. Firstly, it is designed towards consistency, i.e. the design avoids inconsistencies and respective extra synchronization activities. Secondly, it facilitates implementation using component technologies. This is achieved by basing the whole approach on a component-oriented paradigm that covers and integrates the whole lifecycle.

- **Explicit Application Engineering Knowledge**: With the introduction of refinement and translation patterns, relevant application engineering knowledge is explicitly captured and made available, e.g., for maintenance.

7.2 Outlook

As this report only serves as an introduction to component technologies, many of the topics mentioned here should be elaborated in more detail.

Especially the concept of applying refinement and translation patterns in a model-driven approach requires further research, regarding issues like:

- Validate the approach using other component technologies, e.g. .NET
- Transfer refinement and translation patterns into practice.
- Analyze the relevance of refinement sequences and develop suitable theory to handle them in practice
- Motivate other product line implementation dimensions such as the programming language- and configuration management dimension [LM03], [PM02]. E.g., it is hard to capture variabilities in cross-cutting concerns via components because these concerns are spread over all components in the product line. This issue is better addressed by the programming language dimension [PM02].

8 References

[ABB+02]	C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel. Component-based Product Line Engineering with UML, Addison Wesley, 2002.
[AG01]	M. Anastasopoulos, C. Gacek. Implementing Product Line Variabilities, IESE Technical Report, No. 089.00/E, 2000.
[AM01]	C. Atkinson, D. Muthig. Enhancing Component Reusability through Product Line Technology.
[AT02]	S.W. Ambler, T. Jewell. Mastering Enterprise JavaBeans, second edition, John Wiley & Sons Inc., 2002.
[Atk97]	C. Atkinson. Meta-Modelling for Distributed Object Environments, 1st International Enterprise Distributed Object Computing Conference (EDOC '97), October 24-26, 1997, Gold Coast, Australia.
[Bun01]	C. Bunse. Pattern-Based Refinement and Translation of Object Oriented Models to Code, PhD Theses in Experimental Software Engineering, Fraunhofer IRB Verlag, 2001.
[Bus96]	F. Buschmann, Pattern-Oriented Software Architecture - A System Of Patterns, John Wiley & Sons, 1996.
[CD00]	J. Cheesman, J. Daniels. UML Components, Addison Wesley 2000.
[CE00]	K. Czarnecki, U. W. Eisenecker. Generative Programming, Addison Wesley, 2000.
[Cle01]	J. C. Cleaveland. Program Generators with XML and Java, Prentice Hall PTR, 2001.
[Coi96]	P. Cointe (Ed.). ECOOP '96: Object-Oriented Programming. Workshop held at the 10th European Conference, Linz, Austria, 1996.
[Cop99]	J. O. Coplien. Multi-Paradigm Design For C++, Addison Wesley, 1999.
[EE1998]	G. Eddon, H. Eddon. Inside Distributed COM, Microsoft Press, 1998.
[EF00]	A. Eberhart, S. Fischer. Java-Bausteine für E-Commerce Anwendungen, Hanser, 2000.
[GAL+95]	E. Gamma et al. Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[Gre01]	J. Greenfield. UML Profile For EJB, Public Draft, 2001, Rational Software Corporation.

[GT00]	V. Gruhn, A. Thiel. Komponentenmodelle, Addison-Wesley, 2000.
[HKB01]	E. Holz, O. Kath, M. Born. Manufacturing Software Components from Object-Oriented Design Models, IEEE Report 0-7695-1345-X/ 01.
[JBR00]	I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process, Addison Wesley, 1999.
[KHS98]	E. Kamsties, K. Hörmann, M. Schlich. Requirements Engineering in Small and Medium Enterprises: State-of-the-Practice, Problems, Solutions, and Technology Transfer, in: Proceedings of the Conference on European Industrial Requirements Engineering, 1998.
[LM03]	R. Laqua, D. Muthig. Product Line Implementation Technologies: Configuration Management View, IESE Technical Report, No. 017.03/E, 2003.
[MAL+02]	D. Muthig et. al. Technology Dimensions of Product Line Implementation Approaches, IESE Technical Report, No. 051.02/E, 2002.
[MO95]	J. Martin, J. Odell. Object-Oriented Methods: A Foundation, Prentice Hall, 1995.
[Mon01]	R. Monson-Haefel. Enterprise JavaBeans, O'Reilly, 2001.
[OMG00-A]	Meta Object Facility Specification, Version 1.3, Object Management Group, 2000.
[OMG00-B]	OMG News and Information, Competing Data Warehousing Standards to Merge in the OMG, http://www.omg.org/news/releases/ pr2000/2000-09-25a.htm, September 2000.
[OMG01-A]	OMG Unified Modeling Language Specification, Version 1.4, Object Management Group, 2001.
[OMG01-B]	Architecture Board ORMSC, Model Driven Architecture(MDA), July 2001, hhtp:www.omg.org.mda/presentations.htm.
[PM02]	T. Patzke and D. Muthig. Product Line Implementation Technologies: Programming Language View, IESE Technical Report, No. 057.02/E, 2002.
[SO+01]	J. Siegel and OMG Staff Strategy Group. Developing in OMG's Model Driven Architecture, white paper, November, 2001, http:www.omg.org.mda/presentations.htm
[SP01]	B. Staudacher, R. Pichler. CORBA 3.0 - Komponentenmodell: Anwendung und Besonderheiten, pp. 32-39, ObjektSpektrum 1/ 2001.
[Szy98]	C. Szyperski. Component Software - Beyond Object-Oriented Programming, Addison Wesley, 1998.

[Wes02]	R. Westphal .NET kompakt, Spektrum Akademischer Verlag, 2002.

Document Information

Title:	Product Line Implementation Technologies - Component Technology View
Date:	March, 20, 2003
Report:	IESE-015.03/E
Status:	Final
Distribution:	Public