



Evolution Management and Process for
Real-Time Embedded Software Systems

Modeling and System Support for Design-Time Evolution

Deliverable D.2.1-D.2.2

Edited by Hans-Gerhard Groß, FH IESE
& Stefan Van Baelen, K.U.Leuven

15 December 2003

Version 1.0

Status final

Public Version



I T E A

INFORMATION TECHNOLOGY

FOR EUROPEAN ADVANCEMENT

This document is part of the work of the EUREKA Σ! 2023 – ITEA 00103 project EMPRESS.
Copyright © 2002-2003 EMPRESS consortium.

Authors/Partners:

Partner	Author	Email
Fraunhofer IESE	Hans-Gerhard Groß	grossh@iese.fhg.de
European Software Engineering Institute (ESI)	David Sellier	sellier@esi.es
	Gorka Benguria Elguezabal	gorka@esi.es
MSI	Jose Mari Sagarna	sagarna@msi.mcc.es
K.U. Leuven	Stefan Van Baelen	Stefan.VanBaelen@cs.kuleuven.ac.be
UNIS (mainly tools, maybe methodological support)	Dusan Kolar	dkolar@unis.cz
Daimler Chrysler	Wolfgang Köpf	wolfgang.w.koepf@daimlerchrysler.com
BarcoView	Lieven Demeestere	lieven.demeestere@barco.com
Fraunhofer FIRST	Jens Gerlach	jens@first.fhg.de
Uni Magdeburg	Danillo Beuche	danilo@ivs.cs.uni-magdeburg.de

Document History:

Date	Version	Editor	Description
15 Dec 03	1.0	Stefan Van Baelen	Public version based on internal version 1.1

Filename: D2.1_D2.2_v1.0_Public_Version.doc

Abstract

Design-time evolution represents change of an embedded system that is not performed when the system is operational. It means we have an existing version of our system that is amended or changed in an evolutionary step and then released, sold or restarted as a new version. This report outlines a simple step-by-step guide for performing design-time evolution in embedded systems. These steps identify individual notations, techniques, and methods that are important for performing each step in an overall process, or that represent propositions to support each step. The introduced techniques are additionally related to the overall EMPRESS Process.

Keywords:

Component Architecture, Component Adaptor, Component Wrapper, Architectural Support, Tester Components, Testing Interfaces, Flexible Components, Embedded Beans.

ABSTRACT	3
1 INTRODUCTION.....	5
1.1 DESIGN-TIME EVOLUTION.....	5
1.2 TAXONOMY OF CHANGES.....	6
2 MODEL, NOTATION AND PROCESS FOR DESIGN-TIME EVOLUTION	8
2.1 PROCEDURE FOR PERFORMING DESIGN TIME EVOLUTION	8
2.2 PROCESS SUPPORT AND NOTATION FOR RUN-TIME EVOLUTION	9
2.2.1 <i>KobrA Component Specification</i>	9
2.2.2 <i>KobrA Component Realization</i>	10
2.2.3 <i>KobrA's Development Dimensions</i>	10
2.3 HARD- AND SOFTWARE CO-DESIGN.....	11
2.3.1 <i>Embedded Beans in Hardware/Software Component Integration</i>	12
2.4 REAL-TIME AS A QOS REQUIREMENT IN DESIGN-TIME EVOLUTION	14
<i>Memory Constraints</i>	14
2.5 FLEXIBLE COMPONENT APPROACH FOR DESIGN TIME EVOLUTION.....	15
3 ARCHITECTURAL SUPPORT FOR DESIGN-TIME EVOLUTION.....	15
3.1 COMPONENT ARCHITECTURE THAT SUPPORTS DESIGN-TIME EVOLUTION	16
3.2 THE INTERFACE WRAPPER ARCHITECTURE.....	17
3.3 VALIDATION OF COMPONENTS AND COMPONENT ASSEMBLIES	18
3.3.1 <i>Architectural Support for Functional Validation of Component Assemblies</i>	18
3.3.1.1 Tester Components in Built-in Contract Testing	18
3.3.1.2 Testing Interfaces in Built-in Contract Testing	19
3.3.2 <i>Architectural Support for Non-functional Validation (QoS) of Component Assemblies</i>	21
3.3.2.1 Search-Based Validation.....	21
3.3.2.2 Validation of Components.....	22
3.3.2.3 Validation of Component Assemblies.....	23
3.3.3 <i>Flexible Component Usage Validation</i>	25
4 APPENDICES.....	26
4.1 ADDITIONAL DOCUMENTS	26
4.2 LITERATURE.....	26

1 Introduction

Szyperski regards *extensibility* and *evolvability* of software systems as driving factors to move to component technology [Szy00]. He refers to a system as *extensible* “if new stuff can be added in such a way that old stuff can interact with it”. According to this, a system is evolvable if it provides the ability to replace old parts by new parts in order to improve its quality, or in order to add new features. Both extensibility and evolvability require to load new components into an existing system and both these concepts represent severe problems for the design and implementation of a system. The main problem of the design of an extensible system is the placement of “extension hooks” so that extensions can be attached to a deployed system. This is a particular hard problem for embedded systems with their restricted resources. The design of an evolvable system is difficult because the replacement of a particular component can have a great impact on the entire system. Szyperski [Szy00] also regards *component versioning* as an “atomic form of evolution”. The main advantage of supporting several versions of a component in a system is that it reduces the impact on dependent components. This simplifies the process of shifting the whole system to a new version.

Designing evolvable and extensible software systems is also closely related to Parnas’ concept of *program families* [Par76] or the more recent work of Weiss and Lai [Wei99] on *product line engineering*. Parnas’ classical definition [Par76] reads: “We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members”. The insight is that if a designer carefully considers the commonalities and differences of the family members then the cost of development and maintenance of the programs will be reduced. The main reason for this is that members of a program family share basic design decisions and can use identical or similar artifacts.

Program families provide solutions that satisfy individual requirements and avoid performance deficiencies caused by services that are not necessary for a less demanding user. Commonality analysis and variability analysis are the basic activities for designing program families. Commonality analysis identifies basic abstractions that are shared by all members of a family. Variability analysis differentiates a family into individual members or subfamilies.

Weiss’ Product Family-oriented Abstraction, Specification and Translation (FAST) method uses commonality analysis to gather domain knowledge and to construct a structured vocabulary as a base for a Domain Specific Language (DSL). The acquired knowledge on commonalities goes as a “design secret” into the DSL, and in that way it becomes part of all family members. The domain specific language keeps the “design secret” and is used to express variations between family members. Program families are considered under the product-line engineering paradigm in sections 2.2, 2.5 and 3.1 of this report.

1.1 Design-Time Evolution

Design-time evolution refers to system change in embedded application development that is not performed when the system is operational. It means we have an existing version of our system that is amended or changed in an evolution step and then released, sold or restarted as a new version.

Design-time evolution has two independent driving forces (dimensions).

- One is the need for reuse of existing software artifacts (libraries, components,

1 Introduction

applications, etc.) in different application domains not yet supported by the artifacts. This can be seen as evolution in quantity of application domains.

- The other dimension is the evolution of software quality. As most software is not perfect there is often need for changes which address found bugs, or provide a better match to a possibly changed or extended specification.

Additionally, the changes that are made for the purpose of evolution can be divided in two classes,

- externally visible changes (interface, protocol, timing) and
- internal changes (implementation, design and structure or internal timing).

Both dimensions of evolution can cause external as well as internal changes. However in most cases, evolution of quantity requires externally visible changes while evolution of quality usually tries to minimize these changes to maintain compatibility to previous versions. While there is comparatively good support for evolution of quality, both with tools (configuration management, debuggers, analysers) and with methodologies (roundtrip engineering), evolution of quantity is still an open issue.

Today in most cases software configuration management is based mainly on software versioning. Evolution of quality as well as evolution of quantity is based on the organization of textual changes to files by revision numbers with branches, for example. The drawback of this approach is that only the result of a change not the reason for performing the change is stored and handled in a structured and traceable way. With the recognition that evolution of components introduces new problems similar to problems found in software product-family- or product-line-based development, better solutions should be possible. Evolutionary changes made to the component(s) introduce differences and commonalities between different versions of components. The main difference is that in a product family the changes are analysed and modelled before the design and implementation are made. A possible way to model the commonalities and differences are feature models [Kang90]. Feature models represent an easy to understand and graphical visualization [Emp03], and they are mainly used in software family development for representing the problem domain of the family. Because software family development and software evolution share many properties, feature models may be used for modelling the (changing) application domains of components, i.e. the evolution of quantity. While feature models are a promising way to model the variation, it is in general not possible to model structural and behavioural aspects of software systems with it. The coupling of modelling languages like the UML with feature models can provide a uniform solution for this problem. However, feature modelling and its UML integration are not further discussed in this report.

1.2 Taxonomy of Changes

Design-time evolution must be able to cope with different types of changes, for example new features are added to the system, existing features are changed, or features are removed. In any case, the change in the requirements/specification documents must be traced to the location in the system or to some components that implement these requirements. This requires a procedure that supports the application engineer with such decisions. Another important issue is whether we are concerned with anticipated changes, so that we have already thought about possible amendments, or whether we are concerned with unanticipated changes, and dealing with these is a lot more difficult.

- Anticipated changes may be planned in advance and integrated in the overall system

1 Introduction

architecture. So, whenever an anticipated change is required, the system integrator will plug-in different or new functionality into the sockets that have already been devised for a specific change. This is sufficiently supported by product-line architectures and typically well documented and traceable.

- Unanticipated changes are a lot more difficult to handle, since the system does not support any of these changes a priori. The identification of the components that implement the changed requirements is not supported through the documentation. However, unanticipated changes must also be somehow supported in a development process.

Additionally, we can have changes on the –

1. Product level. This is the functionality that will be implemented according to a customer's request.
2. Framework level (product family level). This is functionality that serves as the product basis, and it is used in a number of specific implementations. Product-line engineering will be considered in Chapter 2 in more detail.
3. Platform level. This is the functionality that serves as the display unit environment to be used by a specific product or a product-line. The support for the display unit keyboard can be an example of the platform functionality.

The following changes are examples from the development of Barco's avionics display unit software. It is typically characterized by evolutionary steps according to the previously defined change levels:

1. Evolution on product level, these are changes to the customer-specific product functionality. Some typical cases in which this type of evolution is recognized:
 - In case of maintenance, an operation requires intervention. Example: resolving a software problem
 - In case a change in the specification is requested. Example: as result of integration problems with the rest of the avionics, a spec change is required
 - In case that new functionality is requested by the customer. Example: the customer wants to have an additional display page for showing application statistics
2. Evolution on product line level, these are changes to the baseline from which products can be derived. Some typical case in which this type of evolution is recognized:
 - In case of maintenance, an operation to the baseline requires intervention. Example: resolving a software problem.
 - In case that a change to the baseline specification is requested. Example: a new release of the standard avionics communication protocol specification has been issued.
 - In case that a new baseline functionality is required. Example: new functionality is needed for the product implementation.
3. Evolution on platform level, these are the changes to platform that serves as a basis for products over several product lines. Some typical case in which this type of evolution is recognized:
 - In case a hardware evolution influences the platform. Example: a new hardware device is introduced.

2 Model, Notation and Process for Design-Time Evolution

The taxonomy of changes is important for determining the purpose of a technique.

While this chapter has introduced the problems with design time evolution in embedded systems, the next chapter is concerned with the notation and development process that may support design time evolution issues.

2 Model, Notation and Process for Design-Time Evolution

2.1 Procedure for Performing Design Time Evolution

Design-time evolution is regarded as a development step or process that adapts an existing system to changed functionality, or augments the system with additional functionality. It means an existing system is stopped, its functionality changed, then re-configured, and started again. That is, the system does not provide any service during re-configuration.

Run-time evolution (considered under Task 2.3 and 2.4 of the EMPRESS project) may be seen as an extension of design-time evolution with the significant difference that the system must carry on providing its service regardless of the change or extension of functionality.

Design-time evolution typically follows a few specific steps:

1. Identify changes in the requirements. Every single piece of changed or additional functionality must somehow condense in a system requirement. These are high-level requirements that define and reflect the expected change of system functionality and behaviour. The new requirements must be somehow traced to the existing requirements. How this is done depends on whether functionality is changed or whether it extends the existing system. However this is a typical requirements engineering problem that is taken care of under Tasks 3.1 and 3.2 of the project.
2. Trace the difference in requirements to individual components that must be changed or replaced. A prerequisite for this is that the existing requirements are linked to individual components or component assemblies. This is typically done by stereotypes in the models that associate requirement numbers or use case sections with the location of their implementation, usually in the form of *ComponentName::Operation*.

Product-line developments are typically already providing the infrastructure to cope with anticipated changes.

Standard component-based development methods such as the KobrA method provide partial support for identifying components from requirements (see section 2.2).

3. Search for possible candidate components that may replace the existing components. The requirements that a component implements semantically determine the search criterion. This may be based on structural, behavioural or on feature modelling. How well a component fits the search criterion depends upon how cohesively the requirements can be mapped to individual units. The initial organization of the application has considerable impact on that, and product-line approaches or flexible components provide significant support for facilitating design-time evolution. These technologies are introduced and discussed in section 3.1 of this report that is concerned with the architectural support for software evolution. These techniques follow the principle that early investment in the organization of a system and early planning of anticipated changes may greatly reduce the effort of performing the change later on.

2 Model, Notation and Process for Design-Time Evolution

If the altered requirements may not be mapped to suitable existing components, the new or changed functionality can only be introduced through changing or augmenting existing units, and this in fact comes down to own development efforts.

4. Integrate components syntactically and semantically. Only components that have some semantic compliance to their respective requirements will be fit for integration into the application that is undergoing change. It is already difficult to find a component that is completely satisfying the expected requirements semantically. The fact that such a component additionally satisfies the syntactic requirements (i.e. it provides the right signatures) is very unlikely, however. Integration therefore requires a syntactic and semantic adaptation that may be implemented through component wrappers, adaptor components, or a contemporary component platform. A flexible adaptor technology that is providing such mechanisms is developed in section 3.2 of this report.
5. Check the semantic match at deployment time. Only if the components can communicate syntactically through activities performed in step 4, they can do something useful together. Whether they do some meaningful things can only be assessed through a run-time test that checks the semantic compliance of each component to its contracts with its other associated components or its environment. This comprises a functional check, and a timing check. How this is achieved through built-in contract testing is subject of section 3.3.

2.2 Process Support and Notation for Run-Time Evolution

The Kobra development method [Atk01] represents a feasible and useful approach for supporting run-time evolution under the EMPRESS project. The Kobra method uses the UML as primary notation. This means most software documents that are created during the development with this method are UML models. However, there are other artifacts in natural language or in tabular form, but Kobra follows the concepts of OMG's Model Driven Architecture (MDA), so models are the primary development documents. Additionally, Kobra is fully in-line with the Rational Unified Process (RUP), and it may be considered as an instance of that process. Any other development method, or even no development method at all may be used to come up with UML models for embedded systems. Though, in the following sections we describe the artifacts that are typically created in a Kobra development project since this supports component-based embedded system development quite naturally. All specification artifacts may also be developed completely arbitrarily, for instance in natural language. In fact, the Extensible Markup Language (XML) is more and more being used to express graphical specification artifacts such as models, for instance. XML is a tagged language that is often used in component technologies (e.g. CORBA Components) and for generative programming.

2.2.1 Kobra Component Specification

A Kobra specification is an assembly of descriptive documents that collectively define what a component can do. Typically, each individual document in this assembly represents a distinct view on the subject, and thus only concentrates on a particular aspect of what it can do. A component specification may be represented through natural language, or through some graphical representations and formal languages. Whichever notation is used, a specification

2 Model, Notation and Process for Design-Time Evolution

should contain everything that is necessary in order to fully use the component and understand its behaviour. As such, the specification can be seen as defining the provided interface of the component. Therefore, the specification of a component comprises everything that is externally knowable of its structure (e.g. associated other components, in form of a structural specification), function (e.g. provided operations, in form of a functional specification), and behaviour (e.g. pre- and post-conditions, in form of a behavioural specification). These parts are not mandatory and may change from project to project or from component to component. They rather represent a complete framework for a component specification. Additionally, a specification should comprise non-functional requirements. These represent the quality attributes stated in the component definition. They are part of the quality assurance plan of the overall development project or the specific component. A complete documentation for the component is also desirable, and a decision model that captures the built-in variabilities that the component may provide. These variabilities are supported through configuration interfaces.

2.2.2 Kobra Component Realization

A realization is an assembly of descriptive documents that collectively define how a component is realized. A realization should contain everything that is necessary in order to implement the specification of a component. A higher-level component is typically realized through a combination of lower-level components that are contained within and act as servers to the higher-level component. Additionally, the realization describes the items that are inherent to the implementation of the higher-level component. This is the part of the functionality that will be local to the subject component and not implemented through sub-components. In other words, the realization defines the specification of the sub-components, this is the expected or required interface of the component, and additionally it contains its own implementation. These items correspond to its private design that the client of the component does not see. A component realization describes everything that is necessary in order to develop the implementation of the specified component. This comprises the other server components upon which the subject component relies, as well its internal structure, and the algorithms by which it performs its specified functionality. Therefore, the realization comprises documents for specifying its internal structure, the algorithms by which it calculates its results, and the interactions with other components, these are its own associated servers.

The Kobra method is specifically aimed at product line development, and so it provides many mechanisms to support system and component variability such as decision models, variant components, etc. The method is fully described in [Atk01]. Kobra is based on the UML as standard notation. It basically provides a guide on how to use the UML for component-based and model-driven component and application development. The UML syntax is fully described in the UML 1.4 standard. This will be the primary notation for the EMPRESS project including the derivatives and extensions that have been developed in the ITEA-DESS project [DES01].

2.2.3 Kobra's Development Dimensions

The Kobra method provides natural support for the concept "separation of concerns" that is readily applied in typical engineering-style system developments. It means that the product is separated from the process, and the process is separated in very distinct activities that guide the development team throughout their tasks. In that way developers always know what they are doing, when they are doing it and how they have to do it. These guidelines can be

2 Model, Notation and Process for Design-Time Evolution

reflected by a three-dimensional model, in which every development activity will take place. This is depicted in figure 2.2.1.

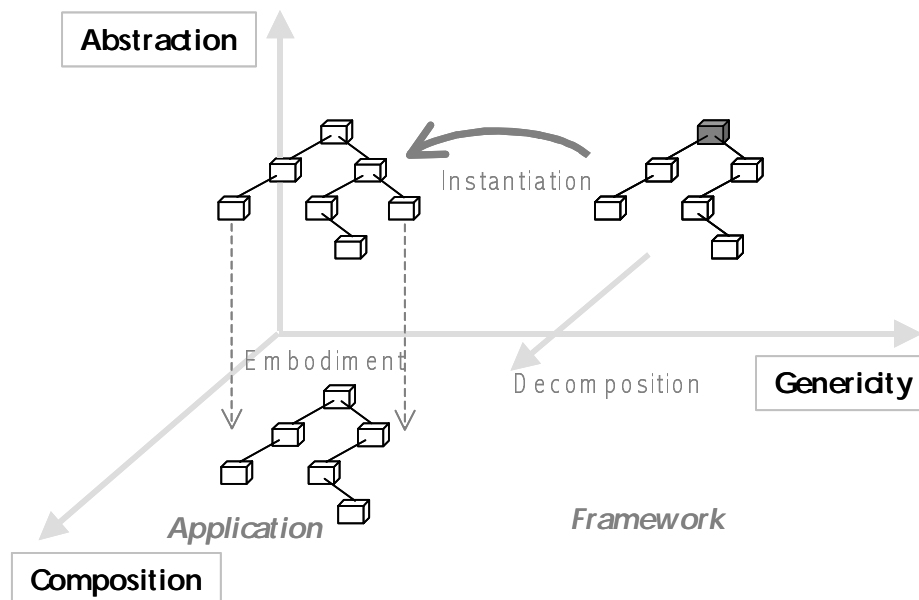


Figure 2.2.1 KobrA's development dimensions.

In KobrA, a development project always starts with the shaded box on the right hand side of the diagram in Figure 2.2.1. This represents the entire system that we would like to have. The first development activity is decomposition of that system into finer-grained parts. Here, we try to separate the system into smaller sub-components that will eventually map to reusable existing modules, and this can be seen as a movement down the composition dimension. Once we have decomposed the system onto the desirable level of detail, we can implement the components, e.g. write some source code, define some hardware components, etc. This activity represents a movement on the abstraction dimension that turns the abstract model notation into subsequently more concrete representations that are eventually executable. It is also termed embodiment in the KobrA method. For single system development these two dimensions are sufficient. However, since we are also concerned with product families, we have an additional “genericity dimension”. This represents an instantiation activity in which a generic framework of a product family is augmented with concrete components in order to obtain a final single product.

2.3 Hard- and Software Co-Design

Hard- and software co-design is typically concerned with the decision of which parts of an embedded system will be implemented as hardware, and which as software. It can clearly be attributed to KobrA's abstraction dimension with its embodiment activity. If the decision has been made it is important that the hardware components are embedded into the software components, or the other way round, that the software components are made to run on the hardware components. This integration of hard- and software components is not fundamentally different from the integration of two software components in embedded system design. This is because hardware components can be wrapped (and usually they are) in software components that act as interfaces to the hardware. Such components are typically

2 Model, Notation and Process for Design-Time Evolution

referred to as drivers and they translate high-level operation invocations from high level programming code into low level hardware operations that access and change hardware addresses. Hence, hardware components and software components can be treated in exactly the same way. The following paragraphs describe a proposed tool that supports hardware and software integration issues during embodiment.

2.3.1 Embedded Beans in Hardware/Software Component Integration

Processor Expert™ is a tool for rapid application development in the domain of embedded systems. Fast application development is enabled especially due to a hardware abstraction layer, which is delivered as a set of components by the tool. Components of Processor Expert are called Embedded Beans™. For every target system (CPU, MCU...), there is a set of Embedded Beans which covers all modes of all peripherals of the given target system. These Embedded Beans provide ready-to-use functionality in a form of tested code provided by the Beans. Key feature of Embedded Beans is their uniform and standardized interface across the target platforms, which enables easy retargeting of the application by simple interchange of the target system in the project because the target CPU is also an Embedded Bean. The interface of a Bean for the runtime is denoted by methods and events, which are provided by that Bean. Processor Expert enables to select just those of provided methods/events that will be used in the application. Thus, the code is generated only for such methods and events that a user selects as required. Moreover, for the design-time configuration, there is a set of properties that enable full configuration of the properties of an Embedded Bean. The settings of these properties done at design-time by the user, and they are instantly checked by the expert knowledge system that is running in the background of the tool. Thus, a user is immediately informed about possible erroneous settings. Moreover, a user can be advised by the system about possible correct settings under active configuration. The code for active Embedded Beans in a development project is generated only when all settings are correct and the Processor Expert reports no error. An application that is created by using Processor Expert can be build from a hierarchy of Embedded Beans with necessary core application-logic code. Nevertheless, Processor Expert enables utilization of already existing code/libraries in the project. If such code uses some part of the target CPU then it is possible block out used peripherals in the Embedded Bean representing target CPU so that the knowledge system disables the usage of these peripherals by the Embedded Beans. Additionally, Processor Exert delivers a thin abstraction layer for direct access to the hardware. This is done via Processor Expert Support Library (PESL). An application structure is depicted in Figure 2.3.1.

2 Model, Notation and Process for Design-Time Evolution

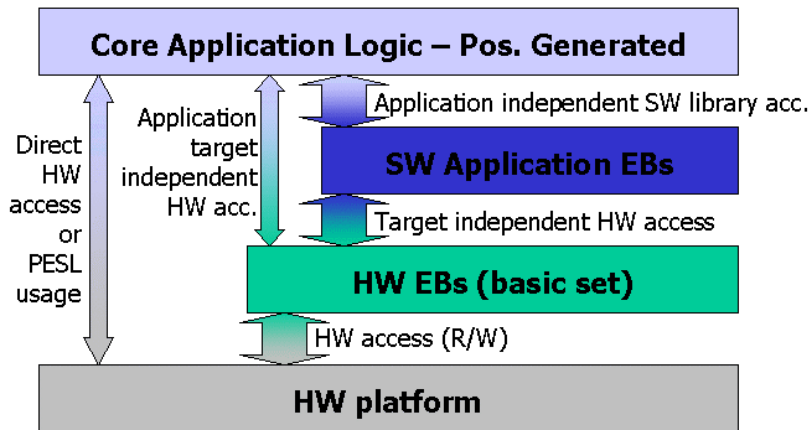


Figure 2.3.1: Organization of an application under Processor Expert.

The so-called “hardware Embedded Beans” create platform independent layers that are suitable for application development. These Embedded Beans come with the Processor Expert software suite. A block of so-called “software Beans” is typically developed by a user.

These Beans usually encapsulate certain algorithms or combine several Beans into a new one delivering new quality. This block can be internally structured as displayed in Figure 2.3.2.

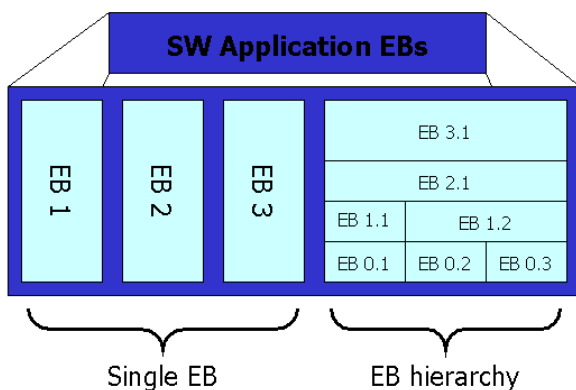


Figure 2.3.2: Organization of a block of Beans.

Embedded Beans can be structured using just their interface (Single EB in the previous picture) and/or using inheritance (EB hierarchy in the picture). Inheritance provided by Embedded Beans is based on the interface inheritance. This enables usage of even non-object oriented languages and it is not necessary to add overhead delivered by object-oriented code. A scheme of the inheritance through interfaces is shown in Figure 2.3.3.

For every Embedded Bean, it is possible to save its pre-set values of properties. Such a

2 Model, Notation and Process for Design-Time Evolution

saved set is called a *template*. Such a template is then the basis for an *interface* definition. An interface is a set of methods and events plus sets from a template. A Bean that is created by inheritance from another Bean uses this interface for communication with the Bean from which it was inherited. For one Embedded Bean, the tool may define a number of templates, and thus several interfaces. On the other hand several Beans can propose the same interface. Embedded Beans can therefore be easily interchanged on any level of inheritance hierarchy. This enables fast modification of complex systems especially when they are re-targeted or when they are used under new conditions. The tool Bean Wizard supports the development of new or inherited Beans. As a part of this tool, the Bean Creator may be used, which is devoted to faster development of Embedded Beans based on inheritance. If a user creates a modification of some Bean they can use built-in features for tracking the performed changes. Moreover, Embedded Beans can be saved under new names so that one can have former and newer version of the Bean.

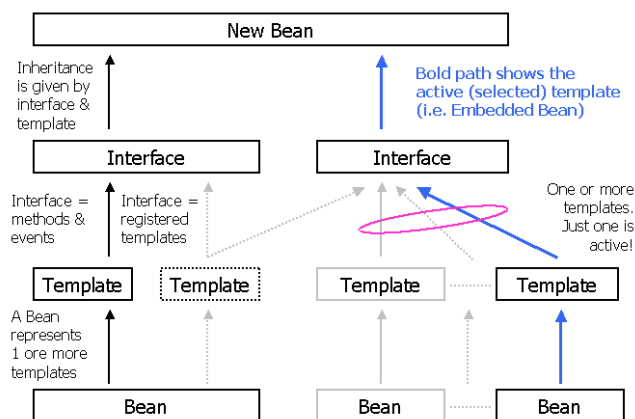


Figure 2.3.3: A scheme for inheritance through interfaces.

2.4 Real-time as a QoS Requirement in Design-Time Evolution

The notation for quality-of-service (QoS) aspects has been initiated and taken care of by the French partners who have unfortunately resigned from the project. However, since we do consider real-time we have to provide a notation for specifying QoS, at least with respect to response time specifications. For real-time specifications during the design phase (decomposition in Kobra) we choose to apply Real-Time UML concepts as put forward in the ROOM modeling language described by Selic, Gullekson and Ward [SGW94].

Memory Constraints

Memory constraints will not be further investigated by the consortium due to changes in the organization of the project through the resigning of the French partners.

The previous sections have considered a few techniques notations and processes that are

3 Architectural Support for Design-Time Evolution

capable of supporting design time evolution. These provide the lever for implementing all aspects of system change at design time. The next section discusses ways of building changeability into an embedded system. This is concerned with the question of how the architecture of an embedded system should be organized so that it supports its own design-time evolution. This comprises functional aspects as well as non-functional aspect plus the considerations that need to be taken into account in order to build up testable embedded systems.

2.5 Flexible Component Approach for Design Time Evolution

A slightly different perspective to product line engineering as described in one of the previous sections is represented by the flexible component paradigm. This aims to thoroughly apply the component paradigm to the product-line paradigm. Flexible components are not only realizing the variable parts of a product-line architecture that is the components that make up the instance of the framework, hence the final product. Additionally, the framework itself is realized through configurable components that may well be used in different frameworks. This in effect realizes a product line of a product line, so that variability is not only defined between products but also between product lines. In a development effort we are not only considering a framework that represents the core plus the components that make up the individual products, but also flexible components that may be assembled to built up a number of different product line frameworks within a distinct domain. The flexible component approach is technique related to the genericity dimension of the KobrA method.

The principles behind product line engineering with flexible components are explained in the contribution “Flexible Component Approach for Design Time Evolution” that is contained in the Appendix A of this document.

3 Architectural Support for Design-Time Evolution

The following sections comprise the contributions of the project partners describing the techniques that concentrate on the architecture for supporting system evolution at design time. An important and powerful approach for achieving design time evolution is realized through product line architectures. They provide the fundamental architectural support for implementing system variants or evolving systems. They are the primary technology for coping with anticipated change. In this respect, change may be seen as a product variant, as a deviation from the common core in product line development. Section 3.1 is concerned with this technology, and it concentrates on how product lines may support design-time evolution.

Further powerful techniques for coping with change, which are currently readily applied in component-based development, are interface wrappers and adaptors. Wrappers are used to give a component a new appearance, so that the client of a component sees what it expects rather than the real thing. Wrappers are typically applied at the server role in a client server relationship, whereas an adaptor is applied at both roles, the client and the server. In fact, an adaptor is an additional component between two interconnected components that merely manages the translation of their interactions. The architectural support of wrappers and adaptors is described and considered in section 3.2.

Wrappers and adaptors facilitate the syntactic mapping between components, however, the fact whether component interactions are also semantically correct can only be assessed when they have been semantically connected. A semantic check can only be performed at run-time, through a built-in integration test. The fact that such a test is performed by built-in test

3 Architectural Support for Design-Time Evolution

infrastructures makes the planning, organization and development of such infrastructures necessary. The architectural support for achieving this is subject of the section 3.3.

3.1 Component Architecture that supports Design-Time Evolution

Product lines are a way to support the evolution of software systems. A product line encompasses a number of related software products that are planned and developed together in a systematic way. The result is a reuse infrastructure that supports the location, the evaluation, and the adaptation of reusable assets, as well as a more accurate planning of development projects. The coverage of multiple software products increases the chances that the evolution of a specific product line member is already covered by the product line. If this is not the case, however, the evolution is facilitated by the flexible nature of the product line and of the assets it consists of. Additionally, product lines also support the decision whether an evolution step of a product line member is to be done at all, or whether it would extend the product line too far. This is possible due to the scoping activities that determine the boundaries of the product line as well as the explicit handling of variations throughout the development of the product line.

An architecture provides an abstraction for reasoning about a software system, as well as a description of its structure and behaviour. There are several reasons why an architecture-based development and careful evaluations of software architectures are important for software development:

1. Architectures facilitate communication about a system in an early phase of system development. There are several stakeholders concerned with different aspects of a system. Examples are customers, users, project managers, developers, or testers. Architectures provide a means to express, negotiate, and resolve competing concerns at an early stage of the development of a software system.
2. Important design decisions are made at that phase in the software life cycle. Several decisions have to be made in the design phase of the software development life cycle. The earlier design decisions are made, the harder they are to change in later phases and the more far-reaching effects they have. The decisions made in the architecture phase have to be made carefully, being aware of their impact.
3. Architectures are transferable abstractions of a system that can be reused. Examples are entire families of systems sharing a common architecture (i.e., product lines) or a component-based approach to development, where systems can be built by plugging externally developed components to a pre-defined architecture that complies with a given standard.

Software development with product-families is typically subdivided into two phases: domain engineering and application engineering. Domain engineering is concerned with identifying the commonalities in a product line. This is what makes the different products related enough, so that they can be expressed by a common core. The variable parts of the common core are determined through the individual products of the product line. Application engineering instantiates the core and generates a product out of the product line architecture by resolving the variable parts of the common core. These variable parts are identified in terms of locations of variability. This variability must be planned and implemented a-priori in the common core. Therefore product-line engineering initially only supports anticipated product evolution because the specific parts must be included in the general part in form of variation points. However, as long as the unanticipated changes fit into the same variation points as the

3 Architectural Support for Design-Time Evolution

anticipated ones, product-line architectures also support such kind of unanticipated changes. If not, the common core must be augmented or changed.

In embedded system development with product lines it is also important to consider quality attributes of an embedded system that affect the design and organization of the product line. The contribution “Pattern-Based Architecture Analysis and Design of Embedded Software Product Lines” considers this subject in much more detail. It is contained in the Appendix B as a separate document.

3.2 The Interface Wrapper Architecture

Since components are loosely coupled exchangeable and reusable software elements, their interaction should be well defined. Therefore, components are defined by a set of interfaces to provide access to their services (provided interfaces). Components are also defined by a set of interfaces to the services they expect from other components in the system (required interfaces). As such, component compositions can be made by connecting (wiring) a provided interface of a component to a required interface of another component.

Although interfaces clearly describe the interaction of a component with all other components in the system, they can impose a burden on the evolution of connected components. Since components can and should be able to evolve individually, the provided and required interfaces of components involved in a connection will sooner or later diverge. Such interface changes often cause an exponential ripple effect, since components can be connected to a significant number of other components. A solution should be provided to enable interfaces to evolve while maintaining the consistency of the defined connections, at least as long the interfaces retains a certain degree of compatibility.

The interface wrapper architecture provides a generally applicable architecture that can provide support to component and interface evolution, diminishing the potential exponential ripple effects of such changes. Interface wrappers are message transformers that are applied each time a message is send or received through an interface. As such, each special-purpose interface message is transferred into a general-purpose inter-component message containing all relevant information, or vice versa. Interface wrappers are concerned with the composition/decomposition and the abstraction/concretization dimension of the Kobra method.

Interface wrappers are able to solve mapping incompatibilities between message names and signatures, parameters, return values, synchronous versus asynchronous messages and message order. Both message and wrapper will contain the necessary version transformation logic to solve such incompatibilities. This is based on the greatest common factor principle, allowing each version of the interface to have only one wrapper to communicate with all other versions. As such interface wrappers enable evolution of interfaces and provide the necessary message transformations.

The principles behind the interface wrapper architecture are explained in more detail in the Appendix C of this document.

3 Architectural Support for Design-Time Evolution

3.3 Validation of Components and Component Assemblies

3.3.1 Architectural Support for Functional Validation of Component Assemblies

The vision of component-based embedded system development is to allow software vendors to avoid the overheads of traditional development methods by assembling new applications from high-quality, prefabricated, reusable parts that are specified and realized with models. Since large parts of an application may therefore be constructed from prefabricated pieces, it is expected that the overall time and costs involved in application development will be reduced, and the quality of the resulting applications will be improved. This expectation is based on the implicit assumption that the effort involved in integrating components at deployment time is lower than the effort involved in developing and validating applications through traditional techniques. However, this does not take into account the fact that when an otherwise fault-free component is integrated into a system of other components, it may fail to function as expected. This is because the other components to which it has been connected are intended for a different purpose, have a different usage profile, or are themselves faulty. Current component technologies can help to verify the syntactic compatibility of interconnected components (i.e. that they use and provide the right signatures), but they do little to ensure that applications function correctly when they are assembled from independently developed components. In other words, they do nothing to check the semantic compatibility of inter-connected components, so that the individual parts are assembled into meaningful configurations. Software developers may therefore be forced to perform more integration and acceptance testing in order to attain the same level of confidence in the system's reliability. In short, although traditional development time verification and validation techniques can help assure the quality of individual components, they can do little to assure the quality of applications that are assembled from them at deployment time.

The correct functioning of a system of components at run time is contingent on the correct interaction of individual pairs of components according to the client/server model. Component-based development can be viewed as an extension of the object paradigm in which, following Meyer [Mey97], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract. This characterises the relationship between a component and its clients as a formal agreement, expressing each party's rights and obligations. Testing the correct functioning of individual client/server interactions against the specified contract therefore goes along way towards verifying that a system of components as a whole will behave correctly.

Built-in contract testing is based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" dynamically at deployment time will fulfil their contract. Although built-in contract testing is primarily intended for validation activities at deployment and configuration time, the approach also has important implications on the development phases of the overall software life cycle. Consideration of built-in test artefacts begins early in the design phase as soon as the overall architecture of a system is developed and/or the interfaces of components are modelled. Built in contract testing is therefore integrated with the overall Kobra software development methodology [Atk01].

3.3.1.1 Tester Components in Built-in Contract Testing

3 Architectural Support for Design-Time Evolution

Meyer [Mey97] defines the relationship between an object and its clients as a formal agreement or a contract, expressing each party's rights and obligations in the relationship. This means that individual components define their side of the contract as either offering a service through a provided interface (this is the server in a client-server relationship) or requiring a service through a required interface (this is the client in a client-server relationship). Built-in contract testing focuses on verifying these pair-wise client/server interactions between two components, when an application is assembled. This is typically performed at deployment time when the application is configured for the first time, or later during the execution of the system when a reconfiguration is performed.

Configuration involves the creation of individual pair-wise client/server relations between the components in a system. This is usually done by an outside "third party", which we refer to as the context of the components. This creates the instances of the client and the server, and passes the reference of the server to the client (i.e. thereby establishing the client-ship connection between them). The context that establishes this connection may be the container in a contemporary component technology, or it may simply be the parent object. In order to fulfil its obligations towards its own clients, a component that acquires a new server must verify the server's semantic compliance to its client-ship contract. It means the client must check that the server provides the semantic service that the client has been developed to expect. The client is therefore augmented with in-built test software in form of a tester component. This is called a server tester component, and it is executed when the client is configured to use the server [Cmp01]. In order to achieve this, the client will pass the server's reference to its own in-built server tester component. This is represented through an <<acquires>> association between the server tester component and the server. If the test fails, the tester component may raise a contract testing exception and point the application programmer to the location of the failure. The following figure (Figure 3.5.1) outlines the organization of a client-server relationship with built-in testing support. The *ServerTester* component comprises tests that are literally built into the client and check the server's compliance to its contract.

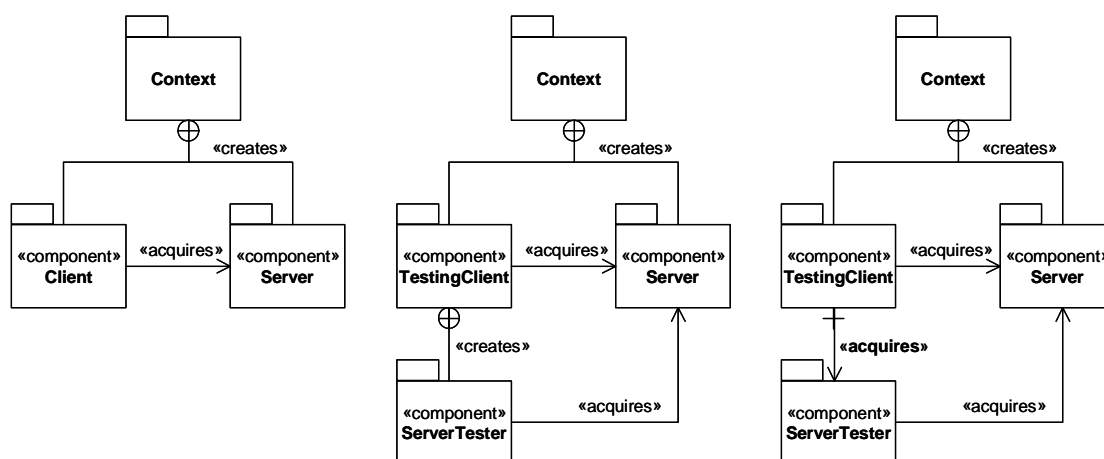


Figure 3.5.1: Component containment hierarchy without (left) and with built-in contract testing (middle, right).

3.3.1.2 Testing Interfaces in Built-in Contract Testing

The basic principles of encapsulation and information hiding dictate that external clients of a

3 Architectural Support for Design-Time Evolution

component should not see the internal implementation and internal state information. The external test software of a component therefore cannot get or set any internal state information. The user of a correct component simply assumes that a distinct operation invocation will result in a distinct externally visible state of the component. However, the component does not usually make this state information visible in any way. This means that expected state transitions as defined in the specification state model cannot normally be tested properly. The contract-testing paradigm is therefore based on the principle that components should ideally expose externally visible state information by extending the normal functional server. In other words, a component should ideally not only expose its externally visible signatures, but additionally it should provide the model of its externally visible behaviour openly. A testing interface therefore provides additional operations that read from and write to internal state attributes that collectively determine the states of a component's behavioural model.

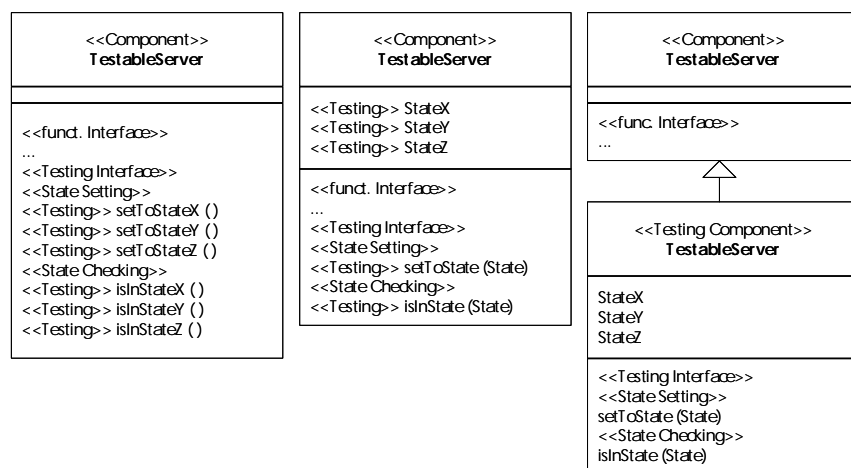


Figure 3.5.2: Concepts of testable components and testing interfaces represented through a class diagram.

It is important to note that the testing interface will not permit direct access to a component's internal attributes, because they are not typically known to external clients of the component. The testing interface only permits access to the externally visible logical states of a component that are represented by internal state variables. The state-setting operations must therefore be implemented in a way that only permits to set valid logical states.

A component that supports its own testing by external clients through an additional testing interface is called testable component. Figure 3.5.2 shows three alternative ways of implementing a testable component.

1. The first class on the left hand side represents a testable server component that has all built-in testing artifacts directly built-in. This is the normal functional interface plus the additional testing interface that comprises operations for setting and getting internal state information (setToStateXYZ and isInStateXYZ). These are the specified externally visible states according to the component's behavioural model (e.g. UML state chart diagram). In the Kobra specification the state setting and checking operations are augmented with the <<testing>> stereotype in order to indicate their special purpose.
2. The diagram in the middle represents an alternative implementation that has public state variables and only one state setting and one state checking operation that take the state

3 Architectural Support for Design-Time Evolution

variables as input instead of two setting and checking operations per state as in the first instance.

3. The third diagram on the right hand side of Figure 3.5.2 logically separates the testing interfaces from the original functional interface. This way of organizing a testing interface facilitates the deployment of testable and non-testable components quite considerably. In that way, if we integrate a component into a system for the first time, we can instantiate a its testable version, after the integration has been found successful we can instantiate its non-testable version and carry on using it without any built-in testing.

3.3.2 Architectural Support for Non-functional Validation (QoS) of Component Assemblies

Validating the timing behavior of a component-based application within the embedded domain is as important as validating the functional correctness. Therefore, component architectures for embedded systems must also support the non-functional validation of component assemblies. In this case it is only the timing aspect that is considered. In the following, a simple architecture for this purpose is introduced, which is based upon the built-in contract testing approach presented in the previous subsection.

3.3.2.1 Search-Based Validation

Dynamic timing analysis can be defined as software testing with the violation of the timing schedule as test criterion. This in fact, represents a typical search/optimisation problem that can be tackled by a search/optimisation method with the actual test cases representing the optimisation parameters. Each parameter set is a vector that defines an input scenario for the task under test. The cost function is determined by the time it takes to execute the task with a given input combination. Advanced search algorithms such as hill-climbing or tabu-search realize optimisation that is guided through the fitness function. Evolutionary algorithms are also belonging to this class, and they have been found most effective for timing validation purposes [Trac01]. This applies genetic algorithms or evolution strategies to typical software testing problems. The target is to find tests that represent the longest or shortest execution time of a program, or to violate its schedule.

However, the technique has not yet been applied to timing analysis of component-based embedded systems. The essential difference between the procedural paradigm and the more recent object-oriented or component-based development paradigm lies in how data and functionality are treated. Whereas the first propagates a strict separation of data and functionality, the second encourages the exact opposite, the combination and encapsulation of data and functionality. All internal state variables are by definition hidden to outside entities of an object. This also includes the test software, in this case represented by the search technique. States can only be accessed and changed through the provided functional interface of the class or component. In general, only a distinct history of interface operation invocations will define the combination of the internal attributes that makes up an internal state. The execution time is fundamentally dependent upon a component's internal state information, and therefore the state information must be part of the optimisation process.

The invocation history of an operation call depends upon the states through which execution must proceed in order to be able to perform an operation. For search-based timing analysis it means that we have to optimise not only the input parameter set for the operation under consideration, but also all input parameter sets for operations that have been executed prior to the subject since these determine the unit's current state. A test case for this example

3 Architectural Support for Design-Time Evolution

comprises an invocation history that must be executed in order to set the required internal state of the tested object, and the optimisation of the input values that these invocations require. This *collectively* defines a test that is incorporated together with the optimisation process into a client of the component. Such an augmented client (a so-called TestingClient) can then assess the timing compliance of its server whenever the two are plugged together and integrated into a new application.

Search based execution-time analysis that is applied to object-oriented and component-based embedded systems is relying on the optimisation of input parameters according to the method invocation history plus a specific architecture. The organisation of the test software (e.g. an evolutionary algorithm) draws its concepts from the built-in contract testing technology for functional validation of component assemblies (see subsection 3.3.1). Contract testing has been specifically developed in order to check the contract compliance of pair wise interacting components at integration and configuration time. In a real-time embedded system, a contract between two components additionally comprises timing properties that a server component is expected to fulfil in order to satisfy the contract of its client. Contract testing augments components with built-in test software and test interfaces that implement an automatic checking mechanism for component interactions. In the following, a simple architecture is introduced that supports search-based validation of components and component assemblies, based upon the built-in contract testing architecture of the previous section.

3.3.2.2 Validation of Components

In search-based timing analysis, validation of components is concerned with determining the timing behavior of single isolated components. The following figure shows the most basic relationship between two components, which is the *client-server* (or clientship in Kobra terms [Atk01]) relationship. The server offers its services through a functional interface, which the client can access and use to invoke public methods on the server.

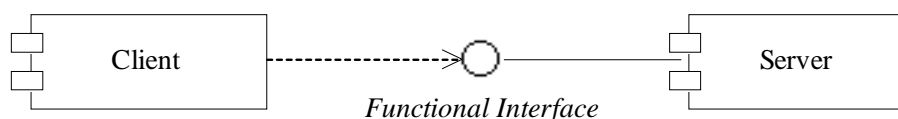


Figure 3.6.1: Client-Server Relationship

If the client is interested in testing the timing of method executions on the server, an additional infrastructure is needed (displayed in Figure 3.6.2) to support the client in measuring time intervals and generating test patterns.

3 Architectural Support for Design-Time Evolution

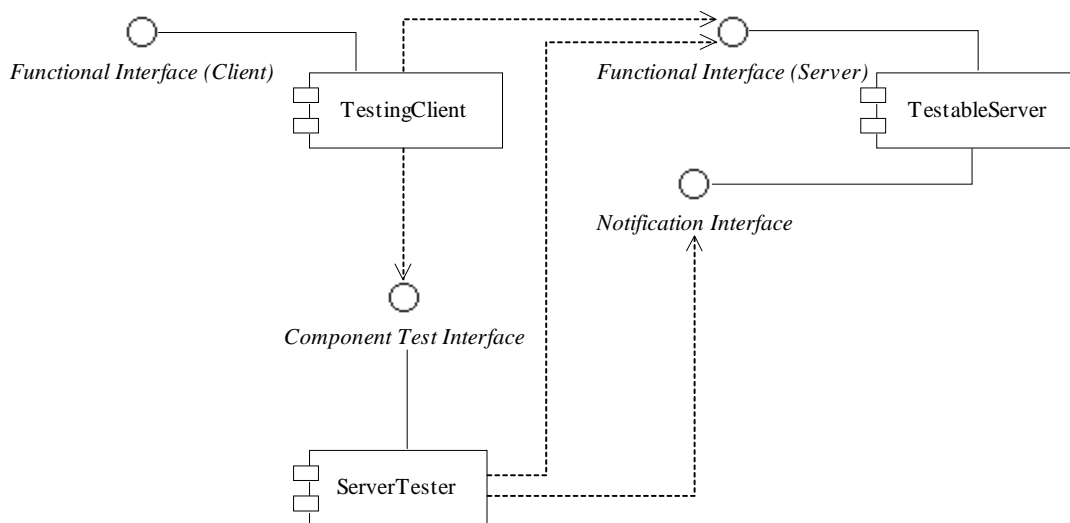


Figure 3.6.2: Client-Server Relationship with Testing Extension

Compared to the basic client-server relationship, now we have a relationship between a so-called *TestingClient* and a *TestableServer*. A *TestableServer* is a server that offers, besides its standard functional interface, an additional notification interface. This interface is used by a *ServerTester* to determine when control-flow enters and leaves a particular method on the server (i.e. the *ServerTester* is notified at the beginning and at the end of a method). The *ServerTester* itself is a new component, which is created or dynamically acquired by the *TestingClient*, for the purpose of testing the server with respect to timing. To do this, the *ServerTester* automatically generates test patterns, by means of search algorithms, and invokes methods on the *TestableServer* with the generated data. The notification interface is used to feed back timing information to the *ServerTester* in order to optimize test pattern generation in a closed loop. The *Component Test Interface* of the *ServerTester* is needed to enable the *TestingClient* to pass over the *ServerTester* a reference to the *TestableServer*. The *TestingClient* itself is a client with built-in test capabilities, so it can handle *ServerTester* generation or acquisition, and decide whether a server fulfils the timing requirements of the client or not.

3.3.2.3 Validation of Component Assemblies

Testing of component assemblies is not conceptually different from testing single components. In the Kobra method, this is expressed by the *principle of uniformity*, which says, "somebody's component is somebody else's system" [Atk01]. For example, a simple component-based application, whose run-time scenario is depicted in figure 3.6.3, has an associated test set up as shown in figure 3.6.4, which is very similar to the one depicted in figure 3.6.2. Again, there is a *TestingClient*, a *TestableServer*, and a *ServerTester*. Notification interfaces allow connecting the *ServerTester* to the server and its various (sub-) components, so the timing behaviour of its parts (components and sub-components) of the component-based application can be fed back into the *ServerTester* to optimize test pattern generation.

Every component in the application reports its timing behaviour to the *ServerTester*, so after

3 Architectural Support for Design-Time Evolution

optimisation, the ServerTester has detailed information about the time spent in each component. Therefore, the test set up of figure 3.6.4 can be used for profiling a component-based application with respect to its timing behaviour.

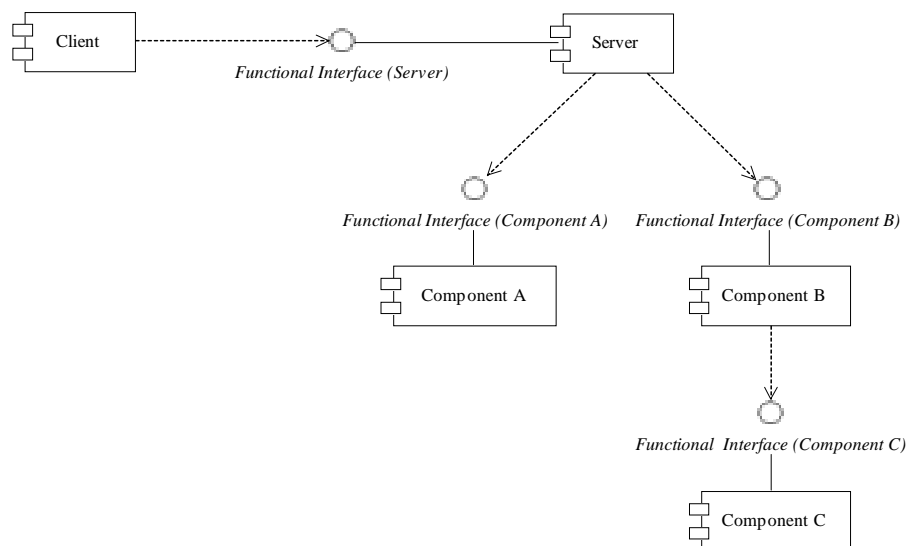


Figure 3.6.3: Client-Server Relationships in Component Assemblies

Before the ServerTester can use the various notification interfaces, it needs to know the references of the components to profile. This is achieved by using an extended functionality of the notification interfaces in a stepwise manner. Such extended notification interfaces can be used to query the references of a component's associated components or sub-components, thereby retrieving the link structure of the application. The process of collecting component references by the ServerTester starts with the TestingClient using the Component Test Interface of the ServerTester to pass over the reference of the server to the ServerTester. Then, the ServerTester uses this reference to query the notification interface about associated components of the server, thereby obtaining their references. However, the ServerTester must know or check that the components are of type "Testable" and in fact possess notification interfaces (in Java, this could be checked with the "instanceof" operator). The process continues until the ServerTester has collected the references of all components that should be evaluated, so it can register itself with the components and get notified, when control flow enters and leaves a method on the components.

3 Architectural Support for Design-Time Evolution

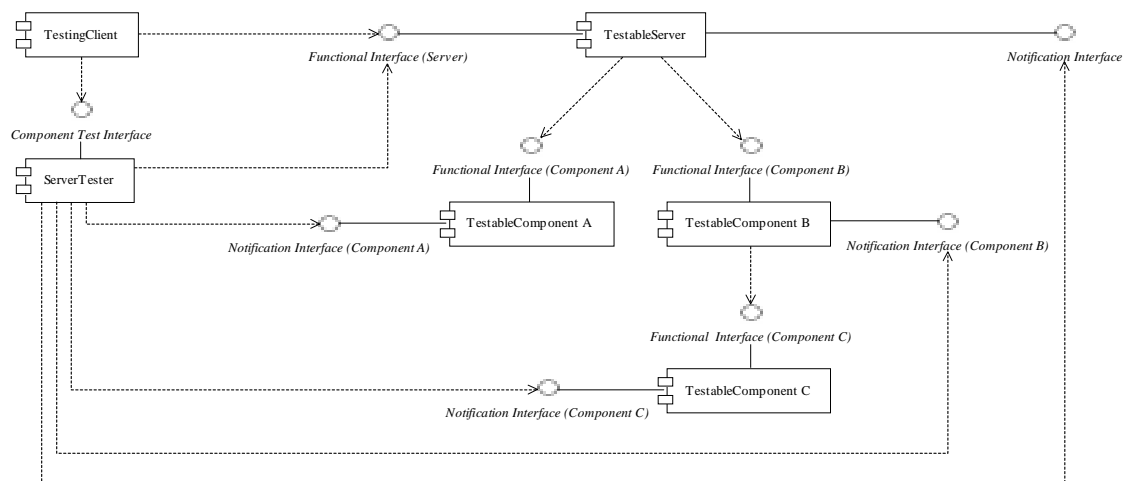


Figure 3.6.4: Component Assembly with Testing Extension

Stimulation of the component assembly with test patterns is performed exclusively through the functional interface of the server component. This is needed to preserve the usage context of the components (i.e. a component is used in a particular way by its clients) and prevents method invocations with infeasible parameter sets.

Furthermore, since in most cases components represent encapsulated state machines, it is necessary to consider internal states of components during testing. Because of the importance of the component context, such states can only be set from outside via the functional interface of the server (or through a specific testing interface operation).

- The first method requires a history of method invocations on the server with input values generated by the ServerTester.
- In the second case with the testing interface operation, the invocation history will be subsumed by a parameterized method in the testing interface that brings the component into the same initial state.

Each test case now comprises a set of method invocations on the server with parameters optimized according to the timing behavior of the component assembly. The architecture as introduced above support this kind of test method by feeding back the timing characteristic of the assembly, but yet preserving the particular usage scenario or context of each single component.

3.3.3 Flexible Component Usage Validation

The validation of the usage of a flexible component (FC) is focused on checking that the parameters that will be used to parameterise the FC are valid. This is different from the FC validation that consist in performing some testing over the FC in order to ensure that it works, that it fulfils its objective.

The principles behind product line engineering with flexible components are explained in the contribution “Flexible Component Usage Validation” that is contained in the Appendix A of this document.

4 Appendices

4.1 Additional Documents

See Appendix A for details on a Flexible Component Approach for Design-Time Evolution, Appendix B for details on Pattern-Based Architecture Analysis and Design of Embedded Software Product Lines, and Appendix C for details on the Interface Wrapper Architecture.

4.2 Literature

- [Atk01] Atkinson, C., et al. Component-based Product-Line Engineering with UML. Addison-Wesley, 2001.
- [Cmp01] Component+, Technical Reports. www.component-plus.org. 2001.
- [DES01] DESS Report D1.1: The DESS Methodology (Version 01.3), <http://www.dess-itea.org/>, 2001.
- [Emp03] EMPRESS Report D.1.2: Essentials and requisites for the management of evolution, March 31, 2003.
- [Kang90] Kang, K.C., Cohen, S.G, Novak, W.E., and Peterson, A.S., Feature-oriented Domain-Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
- [Mey97] Meyer, B. Object-oriented Software Construction. Prentice Hall, 1997.
- [Par76] Parnas, D.L. On the Design and Development of Program Families. In IEEE Transactions on Software Engineering, pages 1–9. IEEE, 1976.
- [SGW94] Selic, B., Gullekson, G. and Ward, P.T. Real-Time Object-Oriented Modeling (ROOM), Wiley, New York, 1994.
- [Szy00] Szyperski, C. Point, Counterpoint. Software Development Magazine February 2000.
- [Trac01] Tracey, N., Clark, J. and Mander, K.: A search-based automated test data generation for high integrity systems. In: Systems Engineering for Business Process Change (Henderson, ed), Springer, 2001.
- [Wei99] Weiss, W.M. and Tau, R.L. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999.