AN ARCHITECTURAL MODEL FOR COMPONENT-BASED APPLICATION DEVELOPMENT

Ad Strack van Schijndel

Cap Gemini Nederland

Daltonlaan 300, 3584 BK Utrecht

e-mail: Ad.StrackvanSchijndel@capgemini.nl

Guus van der Stap

Cap Gemini Nederland

Daltonlaan 300, 3584 BK Utrecht

e-mail: GStap@inetgate.capgemini.nl

Hans Goedvolk

Cap Gemini Nederland

Daltonlaan 300, 3584 BK Utrecht

e-mail: Hans.Goedvolk@capgemini.nl

Professor D.B.B. Rijsenbrij

Vrije Universiteit, Department of Mathematics & Computer Science

De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands

e-mail: daan@cs.vu.nl

Abstract

Component-Based Development (CBD) is a promising concept for the delivery of software systems. CBD is aimed at the development of increasingly complex software systems in a shorter time and with fewer resources. Software systems with CBD are highly adaptable and scaleable in order to meet both new business opportunities and emerging technologies.

The current debate revolves around the industry standard for the execution of components, primarily leading to a race between COM and CORBA. But this discussion does not focus on what types of components can be distinguished and how components are organised in a software system.

In this paper the principles of a component architecture are discussed. This architecture consists of a limited set of building blocks in a two-dimensional layering. The crucial construct in this model is the concept of the 'framework' that serves as context provider for the assembly of components.

1 Component Architecture

1.1 The role of a component architecture

When CBD is put into practice, it is necessary to divide the development organisation in component developers in the role of *component supplier* at one side and developers of software systems assembled of these components in the role of *component consumer* at the other side.

This division can only be effective when a common basis is provided for suppliers and consumers of components. A Component Execution Environment (CEE) like COM or CORBA provides a partial solution. CEE's mainly focus on the technical issues concerning the communication between running software components. A component architecture is needed to provide suppliers and customers a set of rules, standards and guidelines that prescribe what components are and how they are assembled in software systems.

1.2 Component architecture principles

A component architecture in our view is a set of principles and rules according to which a software system is designed and built with the use of components.

This paper focuses on the principles that are independent from the business domain or the technology of the application. This architecture is the base for more specific architectures defining the principles and rules for components in a specific business domain.

The component architecture covers three aspects of a software system. These are:

1. **Building blocks**
   The architecture specifies the type of building blocks systems are composed of.
2. **Construction of the software system**
   The architecture specifies how the building blocks are joined together when developing an application. The architecture describes the *role* that the building blocks play in the system.
3. **Organisation**
   Components are divided in categories based on their functionality. These categories are placed in a two-dimensional layering.

1.3 Benefits of the Component Architecture

The component architecture supports the developers in realising a coherent and consistent design of a component-based software system. Such a design based on the component architecture must be profitable for the following reasons:

1. **Flexibility**

The component architecture allows software developers to adjust a software system smoothly to changing business requirements and new technology

2. **Specialisation**

The component architecture supports specialisation of component developers. Technology-oriented people build technical components and business-oriented people focus on business components.

3. **Minimal impact of changes**

The component architecture separates in a software system the more stable components from the more volatile components. Most changes in a software system remain restricted to the more volatile components.

2 Building blocks

2.1 What is a component?

A component is a piece of software or software design with a well-defined interface and hidden internals. The component is based on a concept, which is recognisable and of value for its user such as another component, a software system or a human user. The interface of a component contains its features, showing what the component can deliver. Examples of features are services, timer events, attributes, tools etc.

So, a user knows *what* the component can deliver, but does not know *how* the features are performed.

There is little to say about the size of a component. A component that keeps stock of a CD collection could be small. If the component keeps track of trends in music, register information about artists and relate the CD collection to all this, it is considerably larger.

The behaviour of a component has different aspects. The CD-component consists of a GUI that allows to enter data, it defines domain-objects such as 'CD' and perhaps 'Artist', it stores objects on your hard disk and it provides business logic 'behind the screen' to make it look intelligent. All these aspects may be combined in one component. However a complex component is more flexible and easier to maintain when it is constructed as a collaboration of a number of components, each realising some specific aspect of the total behaviour.

2.2 What is a framework?

The literature provides different definitions of frameworks.

The Butler Group [butler] defines:

"*A framework is a pre-built assembly of components, together with the glue logic which binds them together and is designed to be extended. A framework is designed to offer a large number of related services centred on a broad theme. The framework may also be treated as a component by other clients which use its services, so the whole framework may offer its services through IDL specified interfaces."*

The definition of Jacobson [jacobson] adds an object-oriented flavour:

"An abstract subsystem is a subsystem that contains abstract and concrete types and classes designed for reuse." and *"Ideally a good framework is designed so that specialisations can be constructed from its abstract types and classes with ease."*

There is a harmony in the different definitions of what a framework is. Both definitions say that a framework creates *a context for the assembly of components in an application.*

A framework is a special kind of component. It has the interfaces of a component and it also offers *plug-points* on which other frameworks or components can build.

A plug-point prescribes a role that must be played by a component or framework that uses the plug-point and it defines the rules that the developer must obey when developing a component or framework that plays the role.

Plug-points represent concepts defined by the framework that provide *a context for development of applications*. A framework creates de facto standards because developers must follow the rules that are put down in the framework. Therefore these frameworks are powerful elements for organising component-based software development.

3 Construction principles

The construction principles prescribe how a software system is composed from components and frameworks.

3.1 Dependencies between components

Component A depends on component B means that component A uses the functionality of component B. Dependencies between components must go one way.

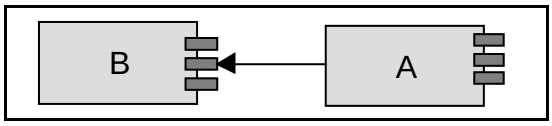

Figure 1 Component A calls component B

Component A can make use of component B in two ways. Component A can call the services of B that are defined in its interface (figure 1). If B is a framework however component A can create a special relationship with B by plugging into B's plug-points (figure 2).

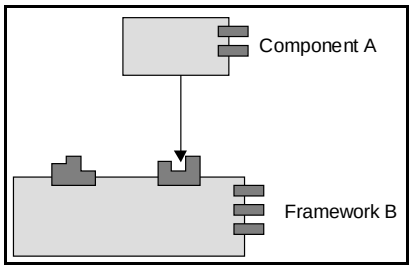

Figure 2 Component A is "plugged" into a plug-point of Framework B

There are essentially three types of relationship possible between a component and a framework:

1. Component A *extends* framework B
2. Component A *uses* framework B
3. Component A *inherits from* framework B

These three types seem sufficient to describe the way in which a framework can give context to the construction of components.

3.2 The *extends* relationship

In the extends-relationship the plug-point represents an *interface*. The plug-point defines services that the component using the plug-point should deliver. The functionality of the component is an *extension* to basic functionality delivered by the framework.

An example is the Windows printing framework that plays a small but important role in many Windows applications. The printing framework provides the basic functionality that is common to all printing services such as printing and viewing of a document. The framework does not deliver the specific functionality for all the different printer types that Windows users utilise. Therefore the plug-points of the framework prescribe abstract components such as *printer* and *viewer* that must extend the functionality of the framework for printing and viewing services for a specific type of printer. The

plug-point provides a clear definition of these components, their interfaces and standards that printer manufacturers must follow when developing the printer and viewer components for their specific printers.

3.3 The *uses* relationship

In the *uses* relationship the plug-point represents a *class*, which can be abstract or concrete, for instance the class Person. The plug-point defines the common behaviour of all persons. The component that plugs into the plug-point creates an ‘alter ego’ of the plug-point-class, for instance the class Customer or Employee. These classes are roles that persons can play in the organisations. The 'alter ego' Customer uses the functionality of Person and adds own features for the Customer role.

The uses relationship is useful for shared data. In many, if not all organisations registration of persons is centralised. When the only issue is the sharing of data, person can be an ‘ordinary’ component and other components simply share the person-data by calling the interface of this component. The uses relationship offers more: alter ego’s can have their own attributes and associations.

3.4 The *inherits from* relationship between components

In the *inherits from* relationship the plug-point represents an *abstract class* in the framework. Plugging into the plug-point means creating a specialisation of the plug-point in the component.

An example is an ‘Agent’ framework. The components in the framework support complex task management functionality. A plug-point provides this functionality through an abstract class called ‘Agent’. Developers realise business components with *agent* functionality as a specialisation of this class such as task managers or workflow managers. This Agent components have the behaviour for controlling tasks, delegation of sub-tasks to other Agent components, handling conditions and more. The components can suspend a task and ‘freeze’ its state so that it can be continued at some other time.

4 Organisation principles

These principles focus on grouping of components and the boundaries between components to provide a basis for organisation of development and maintenance of components and software systems.

4.1 The component architecture is two-dimensional

Layered architectures based on frameworks, components and OO have proven to be very suitable to organise software, designed for change. Jacobson [jacobson] defines a layered architecture as a software architecture that organises software in layers, where each layer is built on top of another more general layer.

This paper proposes a two-dimensional component architecture. The first dimension defines a software system in *layers of generality* in conformance with Jacobson. The second dimension defines a software system in *layers of volatility*.

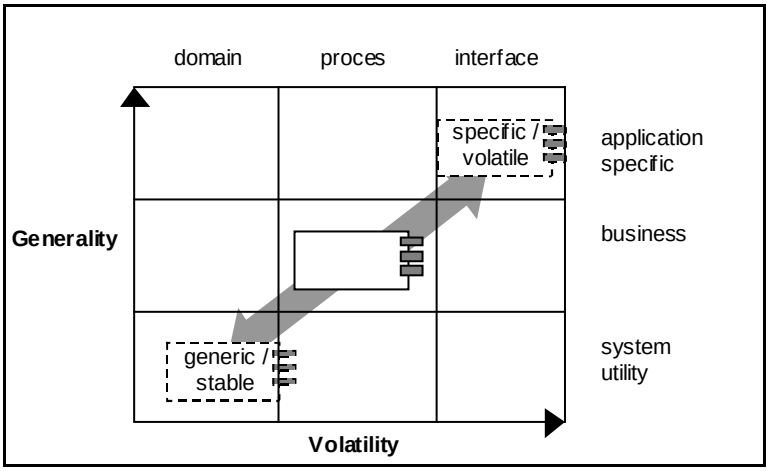

Figure 3: Components in a two-dimensional architecture

4.2 Components in layers of generality

In the layers of generality the upper layers contain more application specific components and the lower layers contain more general components. A typical layered architecture comprises three layers: a system utility layer, a business layer and an application layer, but more layers are allowed.

The *system utility layer* forms the bottom layer providing business independent system utilities such as printing, notify mechanisms, logging and so on. The layer contains components that provide generic components and frameworks that support concepts like agents and printers.

The *business layer* forms the middle layer providing components with generic business functionality and frameworks that define and support complex business concepts through their plug-points. The number of middle layers is not fixed. Sometimes a distinction between a common business, business, sector specific and company specific layers are suitable.

The *application specific layer* is the topmost layer containing the components that are only used in a specific application. These components are not used in other applications.

The system utility and business layer support reuse of general business and system utility components and frameworks in different applications. This accelerates the application development.

The system utility layer separates complex IT-functionality from the business-oriented functionality in the higher layers. The developers can perform changes in business logic and system utility logic more independent of each other. The layering also supports the separation of the development organisation in IT-oriented and business-oriented developers.

4.3 Components in layers based on their volatility

Volatility is a second argument for layering of software components. This layering is orthogonal to the generality layering.

The objective of this layering is to separate the more volatile functionality of software systems from the more stable functionality.

The layering discerns the following components:

Interface components that support on the most volatile part of a software system: the interface with human users and other systems.

Process components that focus on *processes*. These components support for instance workflow logic and transaction processing.

Domain components that support stable information and procedures in a certain business or technology domain.

There is a one-way dependency relationship between interface, process and domain components. Domain components are unaware of the process components that use them. The process components are independent of the interfaces that employ them.

The layers of volatility support independent changes in interface, process and domain components. Domain components supporting financial accounts are used in different transaction processes. New transaction types are added and existing transactions changed without changing the domain components. The same holds for the interfaces. The transaction processes are used by bank-employees through a Windows interface and by the customers through an ATM. New interfaces are easily added, for instance an Internet interface for the customer supporting home banking.

References

[butler]
CBD Forum (1997/1998). *The Butler CBD Reference Model*, East Yorkshire

[jacobson]
Jacobson I., Griss M., Jonsson P. (1997*). Software Reuse - Architecture, Process and Organization for Business Success*. New York, New York: ACM Press.

More details : http://www.cs.vu.nl/~daan/arch/publ.htm