



Information Technology for European Advancement

Task 1.4 - Definition of Components and Notation for Components (D.1.4.4)

Version 02 - Public
Edited by Stefan Van Baelen

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and services**

December 2001

Table of Contents

Table of Contents	1
Purpose of this document	3
WP1.4 partners	3
Document history	4
1. Introduction	5
2. Definition of Components	5
2.1. <i>High-level definition of a component</i>	5
2.2. <i>Characteristics of a component blueprint</i>	6
2.2.1. List of possible component blueprint characteristics	6
2.2.2. Description of the component blueprint characteristics	9
2.3. <i>Characteristics of a component instance</i>	13
2.3.1. List of possible component instance characteristics	13
2.3.2. Description of the component instance characteristics	14
2.4. <i>Characteristics of an interface</i>	16
2.4.1. List of possible interface characteristics	16
2.4.2. Description of the interface characteristics	17
2.5. <i>Characteristics of a component plug</i>	18
2.5.1. List of possible component plug characteristics	18
2.5.2. Description of the component plug characteristics	19
2.6. <i>Characteristics of a connector</i>	21
2.6.1. List of possible connector characteristics	22
2.6.2. Description of the connector characteristics	23
3. The component system	24
3.1. <i>Introduction</i>	24
3.2. <i>Message passing</i>	25
3.3. <i>Creating, removing and connecting components</i>	25
3.4. <i>Scheduling</i>	25
3.5. <i>Component glue</i>	26
3.6. <i>Logging and debugging</i>	26
3.7. <i>Component system interfacing</i>	26
3.8. <i>Stability enforcement.</i>	26
4. The DESS Component Notation	26

ITEA

4.1.	<i>Notation for component blueprints</i>	26
4.2.	<i>Notation for component instances</i>	28
4.3.	<i>Notation for interfaces and connections</i>	29
4.4.	<i>Notation for connectivity restrictions on component connections</i>	33
4.5.	<i>Notation for component composition, decomposition, frameworks and plugs</i>	34
4.6.	<i>Notation for message delegation</i>	36
4.7.	<i>Notation for decomposition into classes and objects</i>	37
4.8.	<i>Notation for decomposition into subcomponents, classes and objects</i>	38
4.9.	<i>Notation for the outside view of a component</i>	39
4.10.	<i>Notation for the unique component and interface identification</i>	39

Purpose of this document

The purpose of this document is twofold:

- To define a precise definition of the term "component" within the context of the DESS project
- To define a notation for the specification and documentation of components within the context of real-time embedded software development. This notation will be in accordance with the UML notation standard, as defined by the Object Management Group (OMG).

This document is a first deliverable of the component definition and notation within the DESS project. During the second stage of the DESS project, further research will refine the definition and notation described in this document. The refined component definition and notation will be described in deliverable D.1.4.4 that will become available at the end of the DESS project.

WP1.4 partners

The WP1.4 partners are listed below in alphabetical order:

1. Barco Display Systems (B)
2. Bull Italia (I)
3. DaimlerChrysler Research Information Technology (G)
4. France Télécom Cnet (F)
5. GMD-FIRST (G)
6. *K.U.Leuven (B) [Task Leader]*
7. Magdeburg University (G)
8. Paderborn University C-LAB (G)
9. Siemens C-LAB (G)
10. Thomson CSF (F)
11. Thomson Multimedia (F)
12. Unis (CZ)

ITEA

Document history

December 1999	1.4.2 V00 Draft A	-- NEW DOCUMENT --
December 1999	1.4.2 V00 Draft B	Inserted remarks from the Paderborn meeting
March 2000	1.4.2 V00 Draft C	Inserted definition of blueprint and instance
		Inserted difference between has- and gives-characteristic
		Inserted characteristics table
		Inserted precise definition of characteristics
		Inserted section on the component framework
October, 2000	1.4.2 V00 Draft D	Inserted introduction section
		Inserted definition of interface and its characteristics table
		Merged non-functional and extra-functional interface
		Reformulated semantics of active component instance
		Inserted UML models of defined terminology
October, 2000	1.4.2 V01	Inserted purpose and partners
		Inserted component decomposition description
		Integrated document UML Component Notation V00 Draft A
March, 2001	1.4.4 V00 Draft A	Addition of component UML stereotype notations
April, 2001	1.4.4 V00 Draft B	Integrated discussions of Leuven workshop
December 2001	1.4.4 V00 Draft C	Update of Chapter 2 and 3 according to the DESS Methodology Document
December 2001	1.4.4 V01	Update of Chapter 4 according to the DESS Methodology Document

1. Introduction

This section provides a common definition and vocabulary of what a component can and must be within the context of the development of real-time embedded software systems. It defines a common terminology that allows people from different application domains to understand each other about component development.

In addition, a notation for the specification, development and documentation of components for real-time embedded software development is developed within the context of the DESS Project. This is done as a UML profile, a collection of stereotypes, tagged values, constraints and notation icons, in accordance with the extension facilities of the UML notation standard, as defined by the Object Management Group (OMG).

2. Definition of Components

2.1. High-level definition of a component

A **component** is a logically highly cohesive, lowly coupled, documented software module that can be used as a unit of development, reuse, composition and adaptation. It therefore is an exchangeable architectural element of a software system that acts as a part within a larger whole. It provides dedicated functionality that can be used in a specific application environment in order to accomplish higher-level goals.

In order to define a clear terminology, we make a distinction between a **component blueprint**, as a description of a reusable software element, and a **component instance**, as a real run-time usage and therefore an instantiation of its blueprint. The term **component** is more general. By using it, we want to indicate both component blueprint as well as component instance aspects, or we do not want to make a distinction between them.

A **component blueprint** is a reusable documented entity that is used as a building block for software systems. It is used to perform a particular function in a specific application environment. Component blueprints are composed using their interfaces. An **interface** describes an interaction point between components, specifying well-defined services. An interface can be applied within a component in two manners, on the one hand as a provided interface, an incoming interaction point, or on the other hand as a required interface, an outgoing interaction point. A provided interface describes how the functionality of a component has to be accessed. A required interface describes what other functionality a component needs in order to perform its own functionality.

A **component instance** is an instantiated component blueprint. This instance behaves as described in the blueprint. Every component instance needs a kind of component system to operate in. A component instance can have its own data space and possibly also its own control flow. As such, a component blueprint does not have a state, while a component instance can have an own state. Also, more than one component instance can exist at the same time, all based on the same component blueprint. As an example, it doesn't make sense to talk about the runtime properties of a component blueprint, since only component instances have runtime properties.

A component blueprint **specification** contains the description of the component that

is aimed to other components that wants to make use of the services of the component. It contains all necessary information necessary to have proper access to the provided interfaces of the component. As such, the component specification acts as the only outside visible description of the component, that for the rest can be seen as a black-box. A component blueprint **design** contains the description of the internal realization structure of the component. This information is not necessary for a proper usage. It describes the internal structure in terms of other components and objects. As such, the component design can act as a glass-box view on a component. A component blueprint **implementation** contains the actual code necessary to implement and execute the component.

A **Component Plug** is a specification of a set of related interfaces that describes the necessities for a component to be plugged into a component framework. It can be seen as an abstract component or as an interface set. A framework containing a number of component plugs can be instantiated by plugging a number of component instances into the framework, each one compliant to the corresponding component plug specification.

A **Connector** is a communication connection between two components, linking a required interface of a component to a provided interface of another component. Connectors can be developed by composing a number of simple connectors and components (a kind of middleware-like components) into a more complex high-level connector. In analogy with components, a distinction can be made between connector blueprints and instances, as well as between connector specification, design and implementation.

2.2. Characteristics of a component blueprint

2.2.1. List of possible component blueprint characteristics

A description of the characteristics of a component blueprint is given as a has/gives table. The '**has**' column of the table specifies whether the component blueprint must, should or can have a specific characteristic. The '**gives**' column specifies whether this characteristic must, should or can be made visible to the outside world. This distinction is important, since component blueprints typically hide a large number of items inside.

The last column indicates whether the characteristic is part of the component blueprint specification (S), design (D) or implementation (I).

ITEA

	Component Blueprint Characteristics	Has	Gives	S/D/I
A	Unique identification	Mandatory	Mandatory	S
B	Outside view: <ul style="list-style-type: none"> • Provides view • Requires view (when not self-contained) 	Mandatory Mandatory	Mandatory Mandatory	S
C	Boundary view: <ul style="list-style-type: none"> • Provides interface • Requires interface (when not self-contained) 	Mandatory Mandatory	Mandatory Mandatory	S
D	Boundary view - multiple interfaces	Recom- mended	Recom- mended	S
E	Inside view	Mandatory	Optional	D
F	Inside view - source code	Mandatory	Optional	I
G	Inside view - component decomposition	Optional	Optional	D
H	Binary form	Optional	Optional	I
I	Verification and validation information	Recom- mended	Optional	D
J	Extra- and non-functional interface for component management, configuration, fine-tuning and adaptation	Optional	Optional	S
K	Specification of Quality of service (QoS) attributes	Recom- mended	Recom- mended	S

Figure 2-1 Component Blueprint Characteristics

The UML diagram in Figure 2-2 expresses the dependencies between a component blueprint, a component instance and an interface. The characteristics of a component blueprint are schematically expressed in the UML model of Figure 2-3.

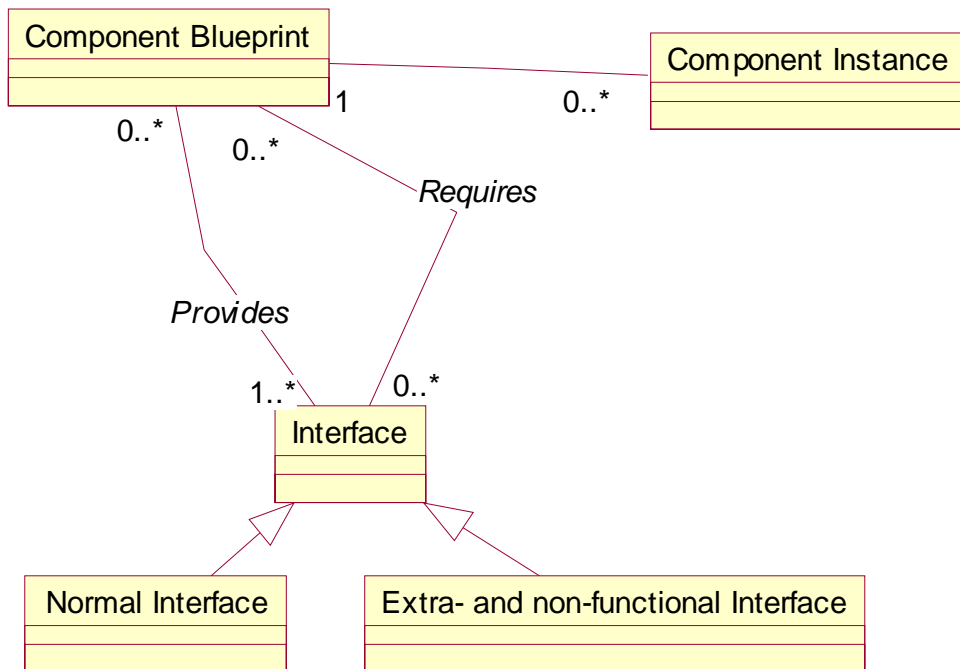


Figure 2-2: UML model of the Dependencies between Component Blueprints, Component Instances and Interfaces

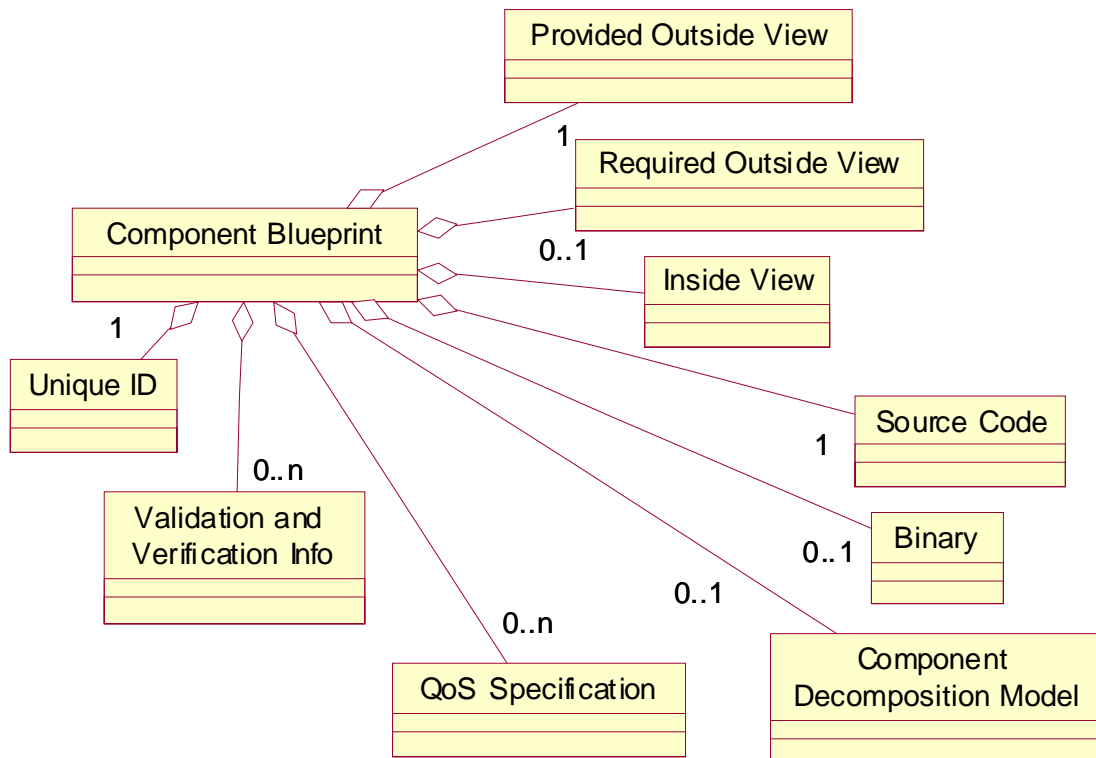


Figure 2-3: UML model of the Component Blueprint Characteristics

2.2.2. Description of the component blueprint characteristics

A. Unique identification

The unique identification of a component blueprint is a name that uniquely identifies the component blueprint. It can be used to specify an unambiguous reference to a component blueprint or as a key attribute for the component blueprint within a component catalogue. It should also be possible to have multiple versions of a component blueprint. Therefore the identification of a component blueprint should consist of 2 distinguishable parts: an **identification name** and a **version number**.

Not only the whole component, but also its interfaces should be named and versioned. The interface characteristics are defined further on in this document. It is even possible that a certain version of a component blueprint supports more than one version of its interfaces, e.g. for backward compatibility reasons. Note that when using the term *version* here, the *implementation version* of the component blueprint is meant, not merely the version of the interfaces of the component blueprint.

B. Outside view:

A component is always used within a certain *application environment*. In addition, it also creates an *application environment* to the components it requires. The outside view contains a description of the context dependencies of the component, defining problem domain terminology for the component and the elements it depends upon. It actually specifies a common language and a number of concepts towards a correct usage of the component or towards a correct interaction with its suppliers. So two views are distinguished:

- **Provides view:**

This view is aimed at the users of the component. With user, we indicate component instances that use services of another component, not the actual person deploying a component. The '*provides view*' is the application environment that the component uses during the interaction with its users and/or that the component offers to its users.

- **Requires view:**

This view is aimed at the suppliers of the component. With supplier, we indicate component instances that supply services to another component, not the component supplier companies that produce components. The '*requires view*' is the application environment that the component uses during the interaction with its suppliers and/or that the component expects from its suppliers. This can also describe some requirements for the deployment environment, OS or execution platform, or specify a specific HW component in case the SW component is a wrapper for that HW component. As an example, a component could need a specific piece of hardware to run on. In that case, this piece of hardware makes part of the '*requires outside view*', the environment needed for the component in order to be fully functional.

Notice that the '*requires view*' is empty when a component is self-contained.

The outside view can also contain an architectural description of the environment of a component. A component instance always functions in a certain en-

ITEA

vironment. A component can impose certain restrictions on its usability, both to its users as to its suppliers. A component can demand a certain mode of operation, and therefore imposes a certain form of architectural style for component instance bindings, such as

- Pipeline system, MVC-based, event-based system, multi-tier client/server
- Active (driving other components, using a certain middleware) or passive (usable within a certain framework)
- Polling-based, interrupt- or event-driven run-time environment

C. Boundary view:

The boundary view describes which interfaces a component offers and eventually which interfaces it needs. Thus, two views are distinguished:

- **Provides interface**

This interface represents the services a component supports. The '*provides interface*' shows the operations can be performed on the component, the events the component can generate and the properties the component can have.

- **Requires interface:**

Via this interface the component describes which external services it needs to perform its own internal services. The '*requires interface*' shows which services the component requires, which events it observes and which properties it accesses. Note that a component does not generally require a specific implementation of an interface, but it can require at least a specific version of an interface. It thus only requires the services the interface describes, not a specific supplier component. When a component really needs a specific implementation, it should be described in its outside view.

It is possible that a component does have an empty requires interface, in which case it is a fully self-contained, independent component.

Notice that Rose RT uses a different terminology that can more or less be mapped to the terminology in this component definition. The Rose RT capsules map to the component blueprints, the protocol role with its incoming and outgoing signals map to the provided and required interfaces, and the protocols map to the synchronization level of the interface description.

D. Boundary view - multiple interfaces

A component is free to offer more than one interface and it is free to require more than one interface. In fact, it is better to isolate distinct usage roles to separate interfaces for each role.

E. Inside view

The inside view contains the description of the logical internal design and architecture. The inside view of a component comprises all documents that explain the internal design and implementation of the component. This ranges from analysis documents to the implementation documentation and source code of the component. The inside view of a component gives more insight in the workings and the use of a component.

F. Inside view - source code

A component can also be delivered with its source code. Using such component mostly means first compiling the component. When a component has its source code associated with it, it can be more easily adapted and understood. Components bought from third parties will not always be delivered with their source code.

G. Inside view - component decomposition

A component can be developed using other components, decomposing the outside component into a number of smaller inside components. Although this decomposition can be considered as an ordinary 'requires' dependency from the outside component to its inside components, it puts additional requirements on the subcomponents and defines more precise semantics for their interdependency.

The component decomposition defines a specific containment relationship between the outside component and its inside components, through a mapping between a number of requires interfaces of the outside component and the corresponding provides interfaces of the inside components. It defines the outside component as a composite component, containing a number of part components. Component composition creates a strong ownership between the composite and its part, with coincident lifetime of the part with the whole. This means that the part components are automatically created when the composite is created, and that they are removed when the composite is been removed by the component system. See the section about the component system for more information about creating and removing components.

Also there is a dedicated use of the part components by the composite component, so the part components cannot be shared or used by another component. Notice that component decomposition can be done hierarchically, decomposing one or a number of subcomponents at their time into subcomponents, and so on. Ultimately, a subcomponent is directly realized at a certain level, by implementing it directly, mapping it onto a device driver of a hardware device, wrapping a software library, ...

A decomposition of a component into part components creates the possibility of delegating provided interface to such subcomponent. This means that any incoming messages through a specific provided interface of the composite component are automatically redirected to the provided interface of that subcomponent. Also, required interfaces of the subcomponents can be delegated to a required interface from the composite component. This means that any outgoing messages through a specific required interface of the subcomponent are automatically redirected to the required interface of the composite component.

When a number of subcomponents have been defined for a composite component, it is possible to define a number of component links, wiring subcomponents together. Notice that components can only be wired together via their interfaces. Moreover, this is only possible when their corresponding provides and requires interfaces are compatible. Such design-time connections between subcomponents are part of the inside view of a composite component.

ITEA

When decomposing a component into smaller components, the subcomponents should best not be hard-wired within the overall component. In fact, it could be useful to decide at a later time which (version of a) component will be used to realize the overall behavior of the complex component. Therefore, the DESS methodology splits the decomposition of a component into smaller components in 2 steps:

- Firstly, the component to be decomposed is designed as a component framework, built using component plugs (abstract components) and connectors between them. As such, the overall structure is defined, including the properties and dependencies that the subcomponents ultimately should possess.
- Secondly, the subcomponents can be plugged into the framework, thereby instantiating the component framework. The resulting component instance model is obtained by replacing each plug by its corresponding subcomponent that has to comply with the plug specification. The plug acts as a placeholder for the concrete component within the overall framework.

Of course, these 2 steps can directly be performed together during system development. But logically, a component decomposition will always be split in two resulting actions: a component framework definition and a framework instantiation.

H. Binary form

A component can be deployed in a binary form. That means that the component composer cannot directly see the internals of the component in the form of its source code. Delivering a component in binary form also means that the component can only be used on platforms that do 'understand' this binary form. So in this case, the outside requires view should at least mention the platform for which the binary is compiled.

I. Verification and validation information

To prove the correct working of a component it is important to provide a kind of component certification. This can be under the form of testing, verification and validation information. As information, one could for example provide black-box and/or white-box test cases, together with the results that were achieved when performing those test cases. Formal proofs of correctness could also be used as a validation of the component. From a user standpoint, the verification and validation information can be very valuable to trust the correct working of a reusable component. Notice that it is not always possible for component developers to anticipate all application contexts in which the component can be used. Therefore, it can be possible that test cases are incomplete and do not cover all possible usages.

J. Extra- and non-functional interface for component management, configuration, fine-tuning and adaptation

An extra-functional interface allows the component composer to make specific functional-related choices for the component. As such, a component could support a choice between a shared or a dedicated (exclusive) use to its users

and/or demand a shared or an exclusive use from its suppliers. In an exclusive usage right, the user can be sure that a component is more or less in the same state as it was after its last usage. In a shared usage, the component could have changed its state due to the usage of another simultaneous user. Also the name and version number of the component blueprint and/or the component instance could be available through the extra-functional interface. An extra-functional interface can be seen as a special type of a provided or a required interface.

A non-functional interface allows the component composer to make a flexible choice between the Quality of Service (QoS) options that a component supports (see QoS attributes). A non-functional interface can also be seen as a special type of a provided or a required interface.

In some situations it is necessary that components negotiate with each other in order to agree on the properties of their interaction. Especially this is the case when one talks about Quality of Service. Both the user and the supplier could negotiate such QoS parameters. Negotiation between components can be useful in a networked environment, where components do not know about each other a priori. Such negotiation can be supported by a specific non-functional negotiation interface.

K. Specification of Quality of Service (QoS) attributes

In order to make components useful for real-time embedded systems, the non-functional, QoS properties of the components, also called operational characteristics, must be known. Therefore, a component blueprint should provide a specification of its QoS attributes in order to be reusable. This is of course strongly dependent on the context of the component. Based on the QoS attributes of the individual components, a system developer can better verify if the QoS constraints of the overall system are met. QoS attributes can also include estimation information, measurements, etc. Non-functional aspects are for example persistency, security, transactional, robustness and timing requirements, such as the memory usage, the time complexity, maximum response delay (worst-case time complexity), average response time, precision and quality of the result and loss of messages. It is possible that a component offer its users the choice between a number of QoS attributes through the usage of a non-functional interface.

2.3. Characteristics of a component instance

2.3.1. List of possible component instance characteristics

The description of the characteristics of a component instance is also given as a has/gives table.

	Component Instance Characteristics	Has	Gives	S/D/I
a	Unique identification	Mandatory	Recommended	S
b	Replaceable at run-time	Optional	Optional	S
c	Component migration	Optional	Optional	S
d	Persistent	Optional	Optional	S
e	Introspection at runtime	Optional	Optional	S
f	Change of non-functional properties at run-time	Optional	Optional	S
g	Own control flow (active component instance)	Optional	Optional	S

Figure 2-4 Component Instance Characteristics

The characteristics of a component instance are schematically expressed in Figure 2-5, using a UML model.

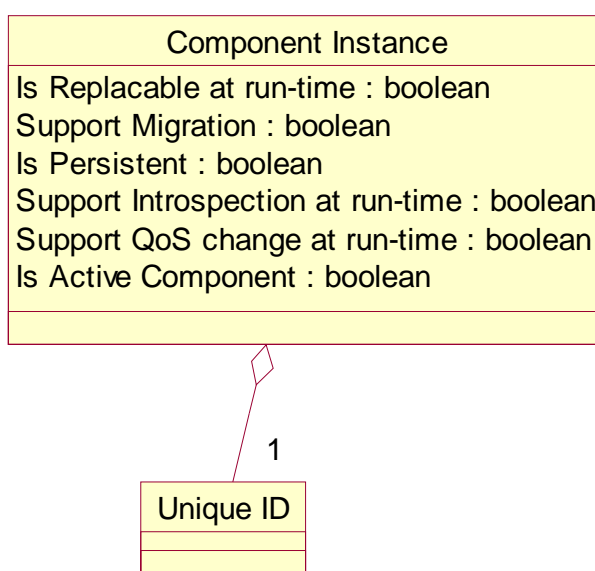


Figure 2-5: UML model of the Component Instance Characteristics

2.3.2. Description of the component instance characteristics

a. Unique identification

Every component instance is unique in the component system. Even when component instances are instantiated from the same component blueprint, they can be made distinguishable. This is indeed necessary when the component instance is able to capture state during its lifetime. Note that the unique identification of a component blueprint and of a component instance is not the same.

The difference between name, reference and binding must be outlined here. Name is unique in a deployment space and is implementation independent.

ITEA

Reference is the local handle in a domain, e.g. its memory address, and is thus implementation dependent. Binding is a list of references that are used to map the name onto the reference of the component instance. With unique identification, only the implementation independent name is meant here.

b. Replaceable at run-time

Some component systems support dynamic plug-in components. In these component systems, component instances can be removed, added or replaced at run-time. This feature can help to maintain, adapt, upgrade or fix a running application without stopping it. As such, a component can have the ability to be dynamically plugged into a hosting system without requiring any recompilation of the total application code.

c. Component migration

Some component instances could migrate from one processor or machine to another. This feature can be used to balance the load in a system. When the component instance captures some state, it is of course necessary to maintain this state of the component instance.

d. Persistent state

In a running system, a component instance can have a state associated with it. When the component instance has to remain persistent, it is necessary to store this state at certain moments in time, and recreate it at a later time. This can be needed to overcome system restarts, reboots, crashes or power failures. An example of such a component is an 'Address Book' component: when the system shuts down, the component instance has to be made persistent; otherwise the content of the 'Address Book' is lost.

e. Support for introspection at run-time

Sometimes it is necessary that a user can query a component instance about its services. Classically, the user is bound to the component blueprint interface at user construction time (e.g. when the user is compiled). When introspection is possible, the user is not bound at user construction time, but it can dynamically (at runtime) find the services of a component instance. Notice that the Java reflection mechanism supports this automatically from within its environment (language & Virtual Machine).

f. Change of non-functional properties at run-time

When a component blueprint offers a non-functional interface for fine-tuning and adaptation, there still are several possibilities to support a QoS choice. It can be a fixed choice on the level of the component blueprint (one choice for the whole component blueprint), a variable choice at creation time of the component instance, or a dynamic choice during run-time (the lifetime of the component instance). In the last case, a component instance could be able to change its non-functional properties and state at run-time when a certain user component asks it to.

g. Own control flow (active component instance)

Components can be classified as passive or active regarding the control flow in a system. Active and passive regarding control flow could be defined based on the possible concurrent execution. If a component which uses an internal control flow, but uses its internal thread only in a synchronous manner, it is passive. A synchronous manner means that the internal control flow is activated only when another component has sent a synchronous request to the component and the calling control flow is blocked while the internal control flow is executing. A component that executes its services in the context of the calling control flow is passive too. An active component has a control flow that executes concurrently to other control flows in the system.

This own control flow can independently initiate communication to other components. As such, active components can be able to create outbound communication without the occurrence of communication events. For passive components, every outbound communication automatically relies on the occurrence of communication events sent by other components.

2.4. Characteristics of an interface

2.4.1. List of possible interface characteristics

A description of the characteristics of an interface is given in the following table, specifying whether the interface must, should or can have a specific characteristic. Since an interface consists only of a specification, the interface will always gives all characteristics it has to the outside world.

	Interface Characteristics	Has	S/D/I
I	Unique identification	Mandatory	S
II	Syntactic interface specification level	Mandatory	S
III	Semantic interface specification level	Mandatory	S
IV	Synchronization interface specification level	Recom- mended	S

Figure 2-6 Interface Characteristics

The characteristics of an interface are schematically expressed in Figure 2-7, using a UML model.

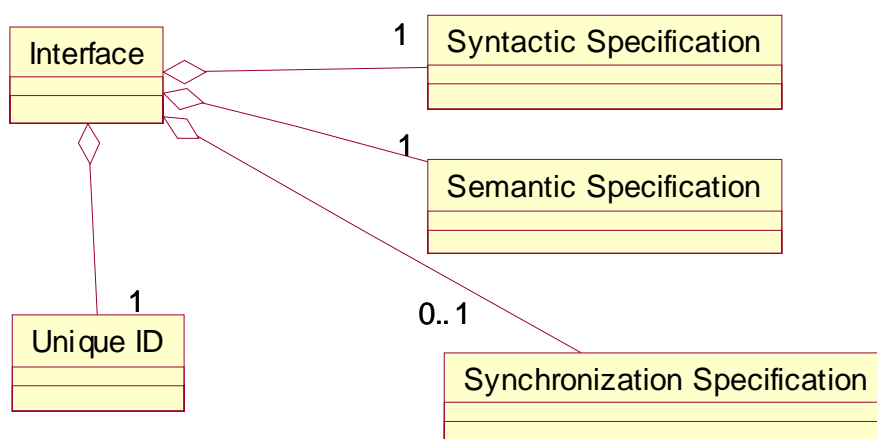


Figure 2-7: UML model of the Interface Characteristics

2.4.2. Description of the interface characteristics

I. Unique identification

The unique identification of an interface is a name that uniquely identifies the interface. It can be used to specify an unambiguous reference to a specific interface or as a key attribute to find a component blueprint within a component catalogue that implements this interface as a provided interface. It should also be possible to have multiple versions of an interface. Therefore the identification of an interface should consist of 2 distinguishable parts: an **identification name** and a **version number**. It is advisable that a newer version of an interface is always a pure extension of a previous version, so that client components could remain operational when server components upgrade to a higher version of a specific interface.

II. Syntactic interface specification level

An interface can be specified at three different levels. Only the first level, the syntactic specification level, is supported in most component systems today. On this level, the supported services are specified in a syntactical manner. The specification of each operation of the interface consists of the name of the operation, the parameters, the return value and the exceptions that can occur.

III. Semantic interface specification level

This second level, the semantic specification level, is mostly supported in a rather restricted, informal way. The semantic level provides a description of the behavior of the services at a semantic level. The specification of behavior can be done informally, but it can also be done in a (possibly partial) formal, contractually based style with pre- and post-conditions, invariants and exception conditions.

IV. Synchronization interface specification level

This level specifies the protocol that has to be followed between two components. This protocol provides a description of the synchronization rules and

the restrictions on the order of its usage. This is also called a protocol contract. Notice that order rules can also be described using preconditions of the semantic level, but it is more difficult and unclear. A separate specification at the synchronization level is therefore advisable.

2.5. Characteristics of a component plug

2.5.1. List of possible component plug characteristics

The description of the characteristics of a component instance is also given as a has/gives table. Since a component plug is a kind of 'abstract component', the characteristics of a component plug are a subset of the characteristics of a component blueprint.

	Component Plug Characteristics	Has	Gives	S/D/I
i	Unique identification	Mandatory	Mandatory	S
ii	Outside view: <ul style="list-style-type: none"> Provides view Requires view (when not self-contained) 	Mandatory Mandatory	Mandatory Mandatory	S
iii	Boundary view: <ul style="list-style-type: none"> Provides interface Requires interface (when not self-contained) 	Mandatory Mandatory	Mandatory Mandatory	S
i v	Boundary view - multiple interfaces	Recom- mended	Recom- mended	S
v	Extra- and non-functional interface for component management, configuration, fine-tuning and adaptation	Optional	Optional	S
v i	Specification of Quality of service (QoS) attributes	Recom- mended	Recom- mended	S

Figure 2-8 Component Plug Characteristics

The UML diagram in Figure 2-9 expresses the dependencies between a component plug, its interfaces, the component blueprint and framework design to which it belongs and the component instance that is plugged in at run-time. The characteristics of a component plug are schematically expressed in the UML model of Figure 2-10.

ITEA

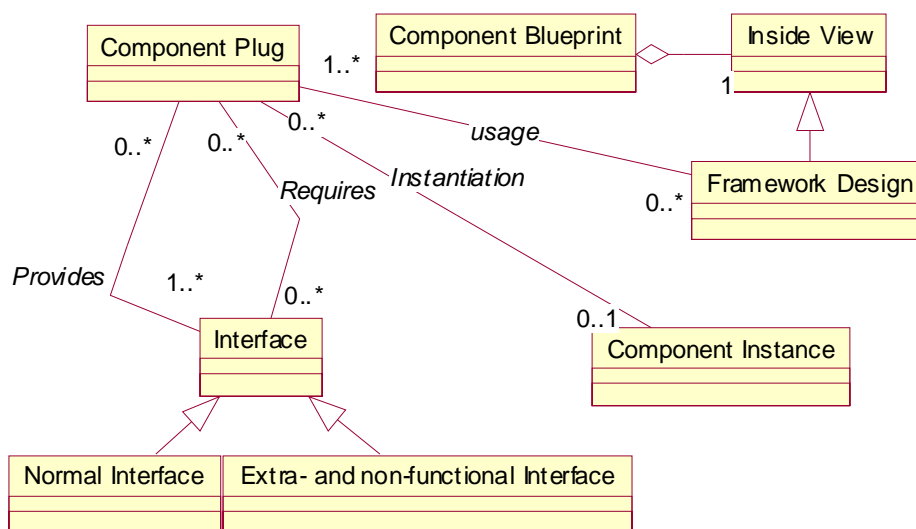


Figure 2-9: UML model of the Dependencies between Component Plugs, Interfaces, Component Blueprints, Component Instances and Frameworks

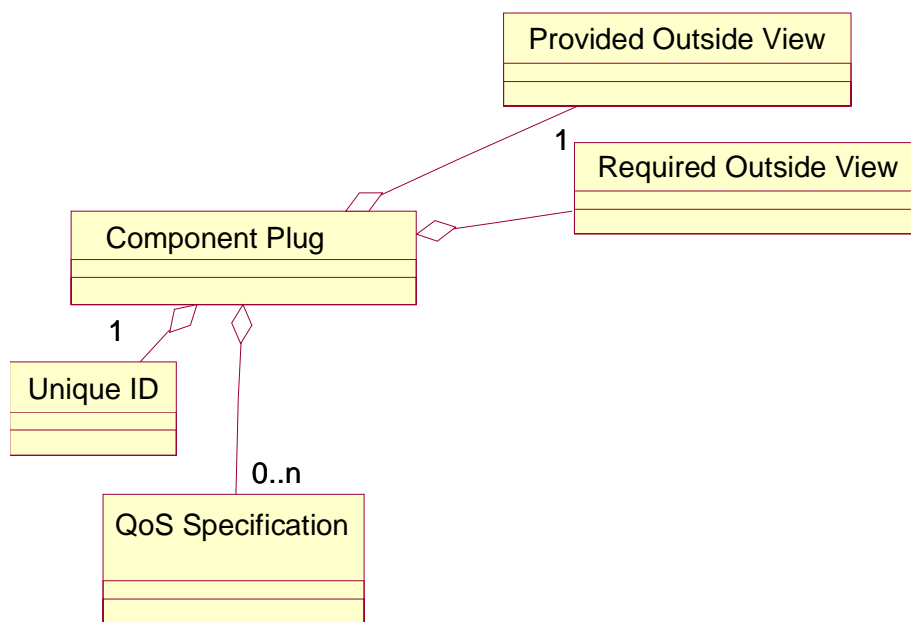


Figure 2-10: UML model of the Component Plug Characteristics

2.5.2. Description of the component plug characteristics

i. Unique identification

The unique identification of a component plug is a name that uniquely identifies the component plug. It can be used to specify an unambiguous reference to a component plug, or as a key attribute to find a component blueprint within a component catalogue that is compliant to this component plug. It should also be possible to have multiple versions of a component plug. Therefore the identification of a plug should consist of 2 distinguishable parts: an **identification name** and a **version number**. It is advisable that a newer version of a plug is always a pure extension of a previous version, so that component

blueprints could remain compliant to a lower version when they are proven compliant to a higher version of a specific component plug.

ii. **Outside view:**

A component plug is always used within a certain *component framework*. The outside view contains a description of the context dependencies of the component plug, defining problem domain terminology for the component plug and the elements it depends upon within the framework. It actually specifies a common language and a number of concepts towards a correct usage of the component plug or towards a correct interaction within the frameworks it will be applied. So two views are distinguished:

- **Provides view:**

This view describes the 'provides' side of the component plug. The '*provides view*' is the application environment that the component plug uses during the interaction with the framework and/or that the component offers to the framework.

- **Requires view:**

This view describes the 'requires' side of the component plug. The '*requires view*' is the application environment that the component plug uses during the interaction with the framework and/or that the component plug expects from the framework. Notice that the '*requires view*' is empty when a component plug must be self-contained.

iii. **Boundary view:**

The boundary view describes which interfaces a component plug offers and eventually which interfaces it needs. This defines a restriction on the component blueprints that can be plugged during instantiation time in this component plug within a framework. Only component blueprints that comply to this boundary view description can successfully instantiate the framework. Two views are distinguished:

- **Provides interface**

This interface represents the services a component plug supports. The '*provides interface*' shows the operations can be performed on the component plug, the events the component plug can generate and the properties the component plug can have. Notice that a component blueprint that will be used as a substitute for this plug must provide all defined operations, events and properties.

- **Requires interface:**

Via this interface the component plug describes which external services it is going to use from the surrounding framework in order to perform its own internal services. The '*requires interface*' shows which services the component plug requires, which events it observes and which properties it accesses. Notice that a component blueprint that will be used as a substitute for this plug must define this interfaces as its own required interfaces. Notice that there is no '*requires interface*' when a component plug must be self-contained.

iv. Boundary view - multiple interfaces

A component plug will consist most of the time of more than one interface. This is actually the reason why interfaces are being grouped into a plug as a kind of 'abstract component'. When the component plug consists of only 1 provided interface, the framework design of the component to which the plug belongs could easily be transformed into an additional required interface for that component. The component plug is also free to require more than one interface. In fact, it is better to isolate distinct usage roles to separate interfaces for each role.

v. Extra- and non-functional interface for component management, configuration, fine-tuning and adaptation

An extra-functional or non-functional interface allows the framework to which the component plug belongs, to make specific choices at run-time for the actual component that is actually going to be plugged in the framework at instantiation time. In this manner, the framework can choose and change the Quality of Service (QoS) options that the plugged component supports. As said earlier, an extra-functional and non-functional interface can be seen as a special type of a provided or a required interface.

vi. Specification of Quality of Service (QoS) attributes

When QoS properties are defined for a component plug, the actual components that are plugged into the framework at instantiation time must comply with these additional properties. This defines further restrictions on the components blueprint that will be used as a substitute for this plug.

2.6. Characteristics of a connector

The connection between interfaces of components is the responsibility of the component system. For real-time systems, the knowledge about properties of the inter-component connections plays an important role. E.g., the time a request requires to arrive at the called component is needed when a schedulability analysis is done.

The DESS component notation allows different basic types of connections:

- **bounded data** : bounded amount of data is sent from required to provided interface, after which the called component may reply with bounded amount of data. Examples are:
 - signals/events
 - method calls with or without return data
- **streamed data** : (unlimited amount of) data is sent from required to provided interface, after which no data is to be sent back to the required interface side. Or bounded amount of data is sent from required to provided interface, after which the called component replies with unbounded amount of data. Examples are:
 - video streams
 - voice transmissions

As with components, there has to be made a distinction between connector blueprints and connector instances. A connector blueprint defines the properties of a set

of connector instances. Connector instances are realizations of a connector blueprint and provide the defined properties. A connector instance is by default not related to other connector instances. However, connector instances may compete for the same resources (media access, processing time, ...).

The set of properties attached to a connector depends on the application domain of the component system. In the simplest case a connector blueprint doesn't have to define any properties beside the mandatory properties given in the next section. A connector instance has to have at least one additional property attached to it, which is the protocol used to transmit messages between interfaces.

Important properties in the context of embedded real-time systems are

- protocol (local method call, remote method call (CORBA, DCOM, RMI, RPC), ...)
- end-to-end communication delays for allowed set of messages (worst case, average case, etc.)
- QoS (Reliability, ...)
- consumed resources (media, processor cycles, ...)

A connector is always used within a certain *component system*. A connection between components can be provided either by the component system (basic connectors) or can be realized by combining existing connector types and connector components into new connector types (compound connectors).

Compound connectors can also be used to define properties of a specific set of connectors, like state machines, protocols, etc.

2.6.1. List of possible connector characteristics

The description of the characteristics of a connector is also given as a has/gives table. Since a connector can be seen as a kind of 'abstract component', the characteristics of a connector are a subset of the characteristics of a component blueprint.

	Connector Blueprint Characteristics	Has	Gives	S/D/I
0 1	Unique identification	Mandatory	Mandatory	S
0 2	Inside view - connector decomposition	Optional	Optional	D
0 3	Extra- and non-functional interface for connector management, configuration, fine-tuning and adaptation	Optional	Optional	S
0 4	Specification of Quality of service (QoS) attributes	Recommended	Recommended	S
0 5	Connection type	Mandatory	Mandatory	S
0 6	Protocol	Optional	Optional	S
0 7	Media type	Optional	Optional	S

Figure 2-11: Connector Blueprint Characteristics

2.6.2. Description of the connector characteristics

01. Unique identification

The unique identification of a connector is a name that uniquely identifies the connector. It can be used to specify an unambiguous reference to a connector, or as a key attribute to find a connector within a connector catalogue that is compliant to this connector. It should also be possible to have multiple versions of a connector. Therefore the identification of a connector should consist of 2 distinguishable parts: an **identification name** and a **version number**. It is advisable that a newer version of a plug is always a pure extension of a previous version, so that connectors could remain compliant to a lower version when they are proven compliant to a higher version of a specific connector.

02. Inside view - connector decomposition

A connector can be developed by composing other connectors and components (a kind of middleware-like components), decomposing the more complex high-level connector into a number of simpler connectors and additional components. As such, a complex connector between a requires component and a provides component can be decomposed into a middle-ware component, a simple connector between the requires component and the middle-ware component, and a second simple connector between the middle-ware component and provides component. As such, the middle-ware component can transform the messages passed over the connection anyway it likes. Another example where a connector decomposition can introduce more detail is when a specific protocol must be used to communicate between 2 components. A high-level connector can represent this communication protocol. Afterwards, the high-level connector can be decomposed into 3 simple connectors and 2 protocol-transforming components:

- A first simple connector can link the requires component with a message-to-protocol transformation component.
- A second simple connector can link the message-to-protocol transformation component on the requires side with the protocol-to-message transformation component on the provides side.
- A third simple connector can link the protocol-to-message transformation component to the provides component.

03. Extra- and non-functional interface for connector management, configuration, fine-tuning and adaptation

An extra-functional or non-functional interface allows the component system in which the connector is applied, to make specific choices at run-time for the actual connector that is actually being used at instantiation time. In this manner, the component system can choose and change the Quality of Service (QoS) options that the connector supports. As said earlier, an extra-functional and non-functional interface can be seen as a special type of a provided or a required interface.

04. Specification of Quality of Service (QoS) attributes

When QoS properties are defined for a connector, the actual components

that are connected using the connector at instantiation time must comply with and can rely on these additional properties.

05. Connection type

The connection type defines the kind of data which can be transmitted using this connection. There two different types of connection possible:

- **bounded data** : bounded amount of data can be send from required to provided interface. The called component may reply with bounded amount of data.
- **stream data** : (unlimited amount) can be send from required to provided interface, after which no data is to be sent back to the required interface side. Or bounded amount of data is sent from required to provided interface, after which the called component replies with unbounded amount of data.

06. Protocol

The protocol defines how the to be transmitted data is received from the sending component and delivered at the receiving component.

07. Media type

The media type defines the physical layer of the underlying connection mechanisms. This allows defining properties of the connector by referencing the properties of the media.

3. The component system

3.1. Introduction

A component instance can interact with other component instances through the use of an implicit or explicit component system. The component system is the context in which a component has to operate. It is the supporting architecture that makes components work together, that glues them and creates a homogenous environment for them. This system is first of all responsible for handling the communication and interaction between component instances, but can also be responsible for the support of the characteristics of component instances. The component system will also contain an implementation part, a run-time component system supporting components, in which components can be executed. This infrastructure is similar to the one provided by an operating system to the application running on the top of the OS. What is important in this approach is that the hosting system is not aware of the functions provided by the component it hosts.

Depending on the application domain, component systems enabling component-oriented development vary and can allow the following key points:

- The system is extensible using plug-in components at runtime
- There is a polymorphic base of component type through the use of component plugs and/or interfaces
- There is a late linking mechanism such as a loader

ITEA

- There is a centralized and automatic management of resources such as garbage collection or processor load balancing
- There is something such as a low level API allowing components to access to the component system and its services mentioned above

The Microsoft COM/ActiveX, the Unix OpenDoc and CORBA technologies are examples of existing systems enabling component-oriented development in a certain way.

3.2. Message passing

The run-time component system can handle message passing between components. If components want to obtain a specific goal from other components or whenever the state of a component has to be changed, a message is sent to the component in question. The component system can also provide support for asynchronous and/or synchronous events. In some situations it is necessary that components negotiate with each other. Negotiation between components should be handled by the component system. The component system can also play the role of an Object Request Broker (ORB) by offering a distributed platform for component interaction. A strong requirement on the component system is the support of various binding and interaction models.

The component system also takes care of sending data over the network, calling the right function on components and eventually other ways of passing messages between components. This includes changing the data format if necessary (when 2 interacting components run on different environments and OS), as is done in CORBA for example.

3.3. Creating, removing and connecting components

The component system should be able to create new component instances based on their component blueprint, remove component instances, connect component instances to each other, disconnect component instances from each other or change components plugged into a component framework. Components can only be wired (connected) together via their interfaces. Moreover, this is only possible when their corresponding provides and requires interfaces are compatible.

When working with components one also needs the ability to find, name and rename components. These abilities should best be provided by the component system. This naming facility should be supported at creation time of the component.

3.4. Scheduling

The component system can handle the scheduling between components. Because components can be thought of as active entities, it is necessary to map this view to a real operating environment. This can be done by the run-time component system, which ensures priorities of messages between components, which takes care of real-time constraints and scheduling in general. For real-time embedded systems, it is crucial that the component system achieves real-time performance and behavior.

3.5. Component glue

The component system could ensure that interfaces are used correctly, as specified by the interacting components, and provide some standard glue components to adapt interfaces between different components. The glue components provided by the component system should be generic, well designed components with as little as possible overhead towards the global system. They can eventually be removed when compiling. For example, a certain component can return a callback with a specific name, while the receiver expects a message with another name. This adaptation can be performed by certain glue components.

3.6. Logging and debugging

The component system can help in debugging by checking whether interfaces are used in the right way. The component system understands the synchronization interfaces provided by the components and can automatically check whether the right calling sequence is used. Another possibility is logging all sent messages.

3.7. Component system interfacing

The interfacing of a component instance towards its component system should be as simple as possible. Calling the component system can be done by sending a message to the run-time component system. The component system can also be seen as a component itself. It can be an advantage to do the interfacing to the component system as if it were a component itself.

3.8. Stability enforcement.

A good component system should take care of stability enforcement. One of the characteristics a component must present is its ability to be safely introduced in an existing system. When introducing a component in a system, the global integrity of the system could require additional checking. The intrusion of a new component must involve a limited perturbation on the state of the running system. This principle of continuity, that is a local perturbation has a local consequence, must be a very important goal when developing new components. This is especially important when considering the continuity principle from the point of view of the resource consumption in an embedded real-time system. The newly included component shouldn't question the performances of the system.

4. The DESS Component Notation

4.1. Notation for component blueprints

The UML v.1.3 standard does cover the concept of component, and provides a specific notation for it. UML defines a component as follows:

"A component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical, distributable piece of implementation of a system, in-

ITEA

cluding software code (source, binary or executable) or equivalents such as scripts or command files, also including business documents etc., in a human system. Components may be used to show dependencies, such as compiler and run-time dependencies or information dependencies in a human organization. A component instance represents a run-time implementation unit and may be used to show implementation units that have identity at run time, including their location on nodes. Software component instances represent run-time manifestations of code units. Components can exist at compile time, at link time, at run time, or at more than one time. "

Implementation diagrams, such as component and deployment diagrams, are defined in UML as follows:

"Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms: (1) component diagrams show the structure of the code itself, the organizations and dependencies among components, including source code components, binary code components, and executable components, and (2) deployment diagrams show the structure of the run-time system, the configuration of run-time processing elements and the software components, processes and objects that live on them. They can also be applied in a broader sense to business modeling in which the 'code' components are the business procedures and documents and the 'run-time structure' is the organization units and resources (human workers and organizational units) of the business. A component diagram has only a type form, not an instance form. To show component instances, one should use a deployment diagram (possibly a degenerate one without nodes)."

The standard UML notation for a component is shown in Figure C-1.

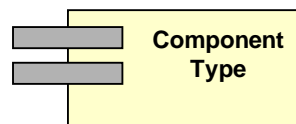


Figure C-1: UML component notation

Although component blueprints are defined in UML v.1.3 as a specific type and treated as a kind of package, we think this is not sufficient for component modeling and notation. A number of problems arise with the current UML notation:

- Components in UML are merely defined as a distributable piece of implementation of a system, including software code and business documents. Within the DESS project, we want to put more stringent requirements on components.
- Component blueprints in UML do not own the model elements that are shown within the component, but are only using their implementation. Within the DESS project, we want to make a strict separation between code usage and dedicated ownership. This can be done by choosing between the usage of required interfaces or component decomposition.
- Components in UML can be associated directly, without using their interfaces. Within the DESS project, we want to oblige the developer to make explicit use

ITEA

of component interfaces in order to wire components together or to decompose components into subcomponents.

Therefore, we propose a specific notation for component blueprints within the DESS UML profile:

- A component blueprint is represented as a class, using a specific <<component>> stereotype.
- UML classes with a component stereotype cannot have any attribute or operations
- UML classes with a component stereotype have two specific class compartments:
 - A provides compartment lists the names of all provides interfaces of the component blueprint
 - A requires compartment lists the names of all required interfaces of the component blueprint

The DESS UML notation is shown in Figure C-2.



Figure C-2: DESS component blueprint notation

4.2. Notation for component instances

UML component instances are represented as UML components, with both an (optional) instance name and a type. The name and type of component instances are underlined, representing the instance-property. A property may be used to indicate the life-cycle stage that the component describes (source, binary, executable, or more than one of those).

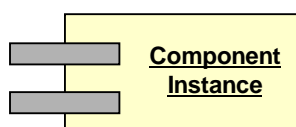


Figure C-3: UML component instance notation

Since component blueprints are represented as a <<component>> stereotyped class within the DESS UML profile, component instances therefore are represented as instances of these UML classes, using the UML mechanism of name underlining. The DESS UML notation is shown in Figure C-4.



Figure C-4: DESS component instance notation

DESS active components will be modeled with a thick border or using the {active} property keyword, analogous with active objects in UML.



Figure C-5: DESS active component instance notation

4.3. Notation for interfaces and connections

An interface can be specified as an UML interface, which actually is a class with an <<interface>> stereotype (formally equivalent to an abstract class with no attributes and no methods, and only abstract operations). Each usage of a specific interface by a component is modeled as an association between the interface and the component.

However, a clear distinction must be made between provides and requires interfaces of a component. UML attaches provides interfaces with a realize relationship to the component (or a solid line when the interface 'lollipop' notation is being used), whereas requires interfaces are connected using a <<uses>> dependency. Moreover, interfaces are shared between all classes and components that provide or use them. The standard UML notation is shown in Figure C-6.

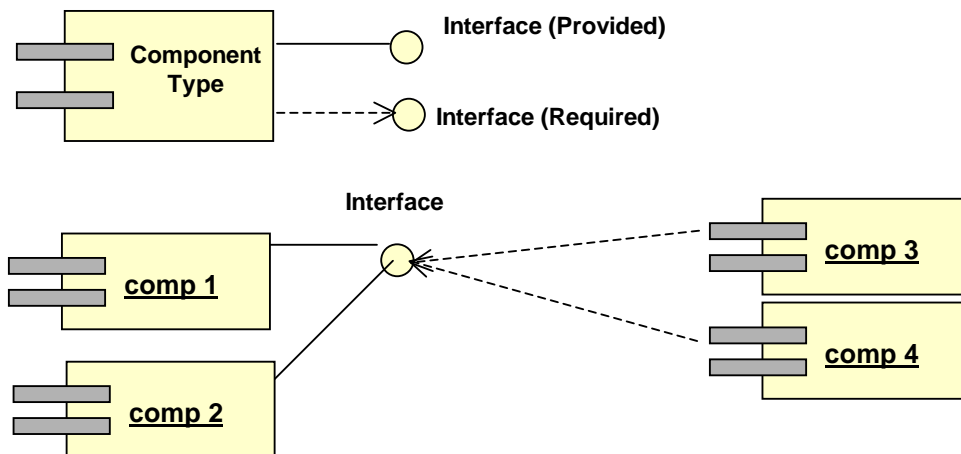


Figure C-6: UML interface notation

Within the DESS project, 2 new stereotypes for interfaces are introduced: <<provided>> and <<required>>. As such, the interfaces stereotyped as <<provided>> can only be attached to the component actually providing the interface, whereas the interfaces stereotyped as <<required>> can only be attached to the component actually requiring the interface. As such, the components do not share the same interfaces, but actually have their own (eventually slightly different) provides and required interface specifications. The association between the interface and the component class must be a directed association (one-way association): from the provided interface to the component on the one hand, and from the component to

the required interface on the other hand.



Figure C-7: DESS interface notation

Two components can only be wired together by linking a provided and a compatible required interface using an association. This is a stereotype for a direct association between the two components. As such, the DESS UML profile defines 3 direct associations for each component link: a first one linking the provided interface to the supporting component, a second one linking the required interface to the dependent component, and a third one linking the provided interface of the first component to the required interface of the second component, expressing the actual link. Normally, a connection is made between two component instances. A connection between two component blueprints is used when a component is decomposed into smaller subcomponents, as illustrated in the next sections.

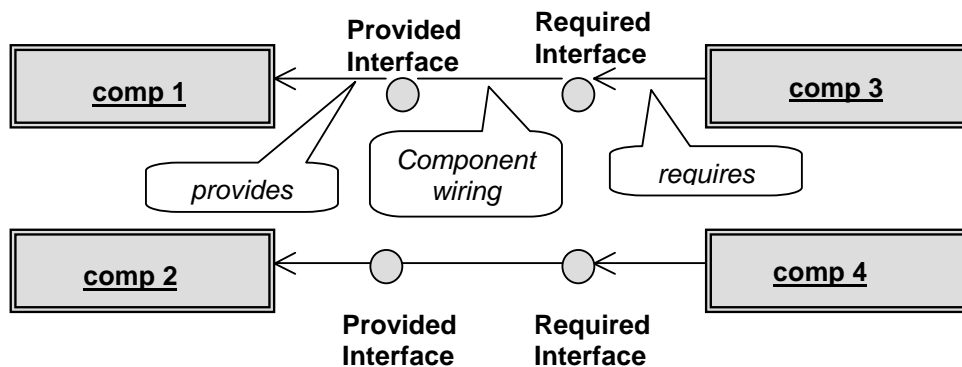


Figure C-8: DESS component wiring notation

Notice that UML only uses 2 associations for a component link: a first one (an abstraction dependency, stereotyped as <<realize>>) linking the interface to the component providing the interface, and a second one (a uses dependency) linking the interface to the component requiring the interface. As such, the actual link between the two components is heavily neglected in the standard UML notation.

The DESS component connection between 2 component interfaces can be seen as a stereotype for a direct association between the two components. In addition, the interfaces are represented as interface specifiers on the association ends, indicating that the behavior that is expected from the connected component is defined by the specified interface. Notice that the provided and required interfaces have to be compatible in order to establish the component connection.

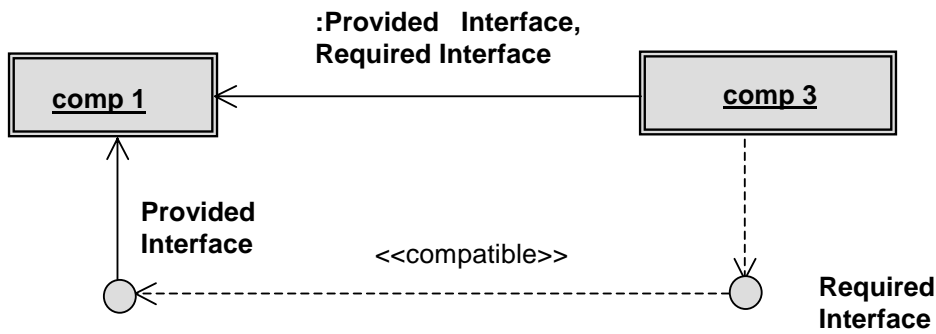


Figure C-9: Mapping of DESS component wiring notation to UML

Component connections can be static or dynamic. When a connection is static, it is defined at design time or at composition time, and cannot be changed anymore at runtime. A dynamic connection can be changed, replaced or removed at runtime. To indicate the status of a component connection, the UML {frozen} property can be used to specify a static connection. Two kinds of static connections can be identified:

- When {frozen} is specified on a required interface of a component blueprint, every required component instance that is connected at run-time to an instance of the blueprint cannot be changed anymore. So the restriction is put on every component instance of the involved blueprint.

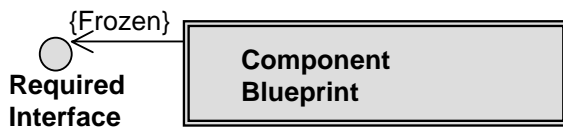


Figure C-10: DESS notation for frozen component wiring

- When {frozen} is specified on a connection of a component instance, this connection of the instance cannot be changed anymore. However, other connections of component instances of the same blueprint can be changed. So the restriction is put on a specific component instance

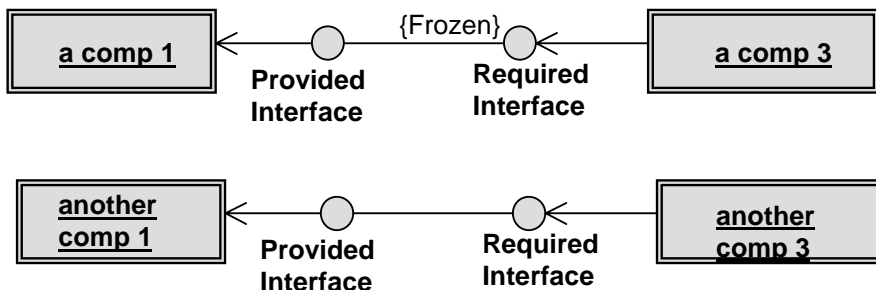


Figure C-11: DESS notation for frozen connections

ITEA

The DESS methodology allows to decompose connections into a middle-ware component and 2 simple connections. Such middle-ware connections must be able to receive and re-send all messages sent over the connection. To model such general interfaces, a universal interface is introduced. The notation for such universal interfaces is shown in Figure C-12.

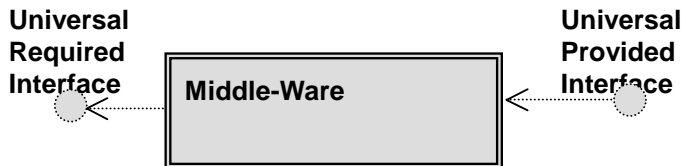


Figure C-12: DESS universal interface notation

Universal interfaces can always be further detailed by introducing more detail about the protocol translations:

- A message-to-protocol transformation component can accept the original interface and generate message compliant to a specific protocol.
- The middleware component can now be shown with its specific dedicated protocol interface.
- A protocol-to-message transformation component can resend the message, accepting messages of the specific protocol, and generating messages of the original interface

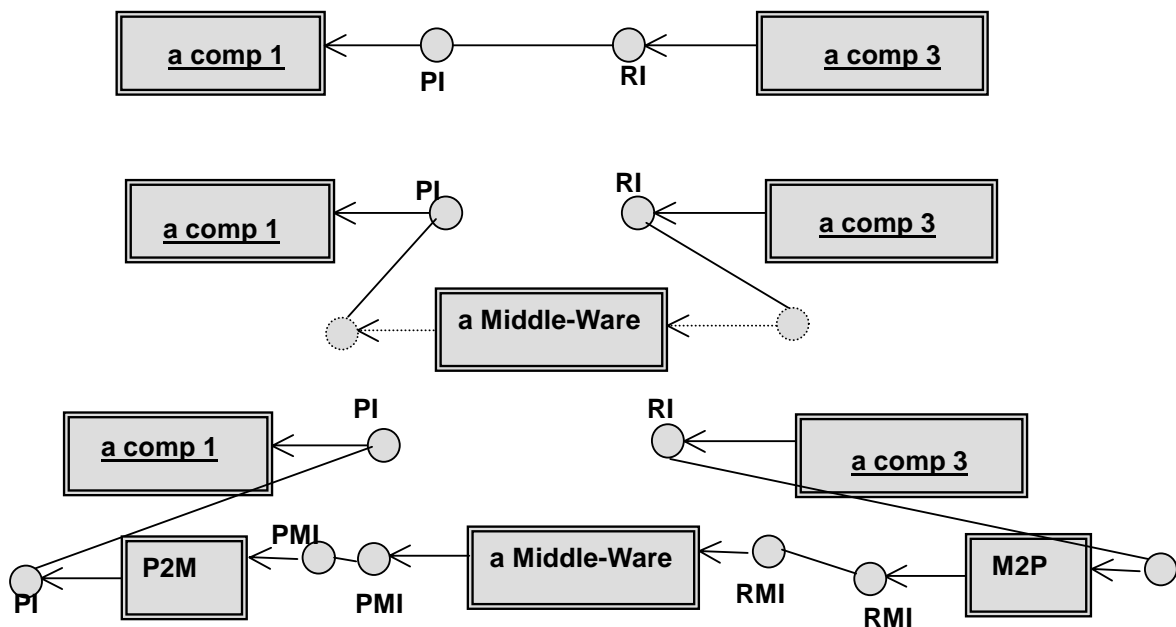


Figure C-13: DESS connection decomposition notation

4.4. Notation for connectivity restrictions on component connections

Defining required and provided interfaces for components is not always sufficient. In a classic component configuration, a component requires exactly 1 component that is compatible with the required interface. On the other hand, a component can provide its services to an unlimited number of other components.

But in some situations, components can put restrictions on the number of components it can serve. As such, a component can restrict its usage to at most one or a specific maximum number of components.

On the other hand, a component can at run-time be connected to a number of components that can provide a specific service. This can be the case when a number of duplicate components are present in the system, or when an undefined number of plug-in components can be added at run-time to the system.

Since UML already defines a multiplicity restriction on associations, the DESS profile adapts this multiplicity item on an association end to express additional component restrictions. In fact, the restriction is placed in between the component and its provided interface on the one hand, and between the component and its required interface on the other hand. The multiplicity values will be applied to the underlying UML association, connecting the two components through their interfaces.



Figure C-14: DESS connectivity restriction notation

When the multiplicity value is unspecified, the DESS profile defines the default value as follows:

- The default multiplicity of a required interface is 1..1
- The default multiplicity of a provided interface is 0..*

Notice the following special values for multiplicity constraints on component interfaces:

- The multiplicity of a required interface is 0..x: This means that the component can function without having to be connected to another component. As such, the availability of a component providing the required interface is optional for the functionality of the component.
- The multiplicity of a provided interface is 1..x: This means that the component cannot be present in the system without the presence of at least one component requiring the functionality of the component. When the last client component is stopped, the component itself will also cease to exist.

When the interface multiplicity allows many components to be attached, it is possible to create a number of connections, each one directed from or to a different component. However, analogous with multi-objects in UML, it is possible to use one connection to a multi-component instead of drawing a number of connections to copies of a component.

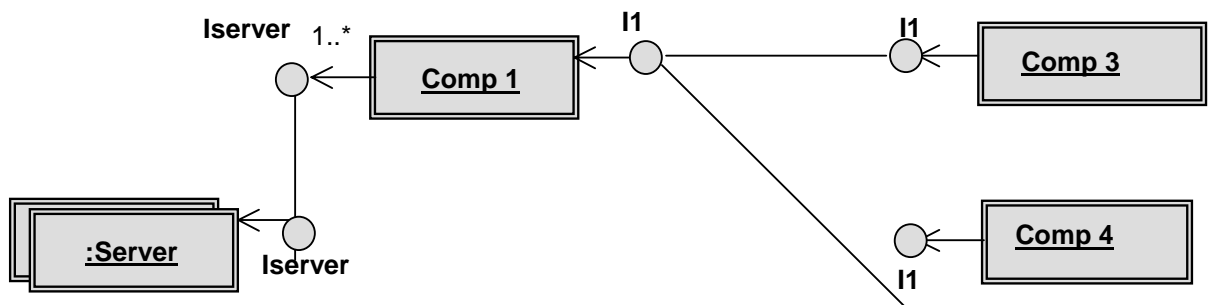


Figure C-15: DESS multi-component notation

4.5. Notation for component composition, decomposition, frameworks and plugs

Large components can be realized by means of a number of smaller components, that all offer a part of the functionality of the large component. As such, a collection of components and connections between components form a new component of its own. This assembly of components into a bigger component is called *component composition*. From the viewpoint of the large component, the disassembly into a number smaller sub-components connected together, is called component *decomposition*.

One could consider the dependency between the large components and its sub-components at the same level as other dependencies through required interfaces. This is in fact true. However, defining a component as a sub-component instead of a required external component puts a number of additional properties and restrictions on the relationship between the 2 components:

- Component composition creates a strong ownership between the composite and its part, with coincident lifetime of the part with the whole. This means that the part components are automatically created when the composite is created, and that they are removed when the composite is removed by the component framework.
- Also there is a dedicated use of the part components by the composite component, so the part components cannot be shared or used by another component.

UML provides a composition aggregation as a special kind of association. As such, a composite object can be defined, containing a number of sub-objects. The DESS profile for UML adopts this idea of composition aggregation and defines the relationship between a composite component and its sub-components as a composition aggregation between the components, through a number of required and provided interfaces.

As presented in section **Error! Reference source not found.**, the DESS methodology splits a component decomposition in 2 steps to allow a looser coupling between the internal component design and the subcomponents to be used at run-time:

- Firstly, the component to be decomposed is designed as a component framework, built using component plugs and connectors between them, defining the overall structure. The notation for such component framework is presented in Figure C-16.

ITEA

- Secondly, the subcomponents can be plugged into the framework, thereby instantiating the component framework. The resulting component instance model is obtained by replacing each plug by its corresponding subcomponent that has to comply with the plug specification. The notation for such instantiated framework is presented in Figure C-17.

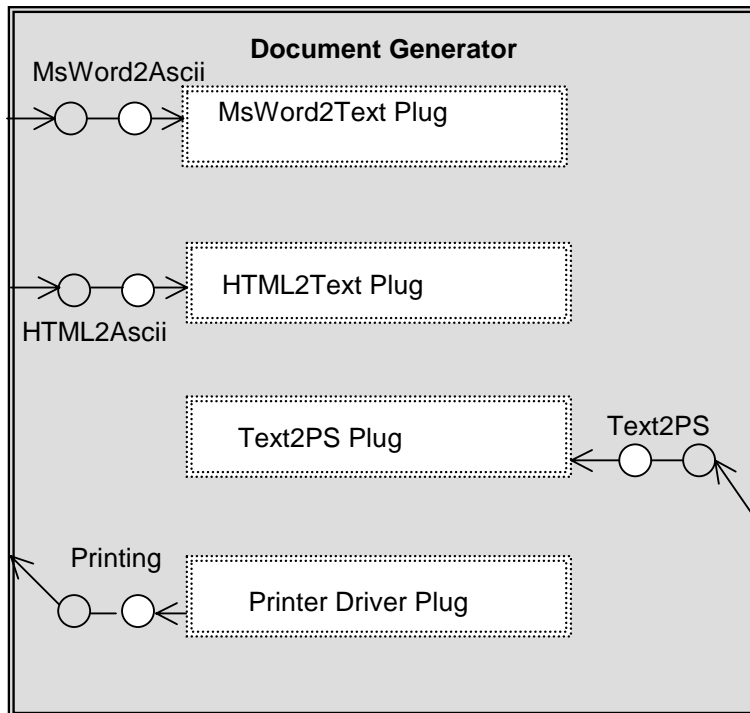


Figure C-16: DESS framework and plug notation

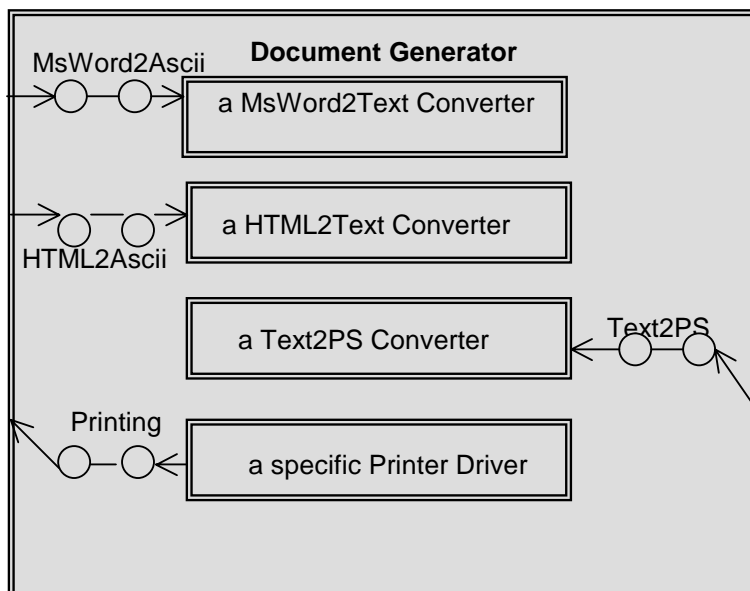


Figure C-17: DESS instantiated framework notation

Also for component decomposition, there is no direct dependency between the outer component and a sub-component. The component decomposition always defines a specific containment relationship between the outside component and its inside components through a mapping between a number of requires interfaces of the outside component and the corresponding provides interfaces of the inside components. Notice that component decomposition always creates a dependency with connectivity restriction exactly equal to 1..1, due to the fact that there exists a dedicated use of the part component by the composite component.

It is also possible that a sub-component on its turn is partly dependent on the composite component. This can be the case when the sub-component requires specific additional services, or when the sub-component has the need for callback abilities to inform the composite component about its internal state change.

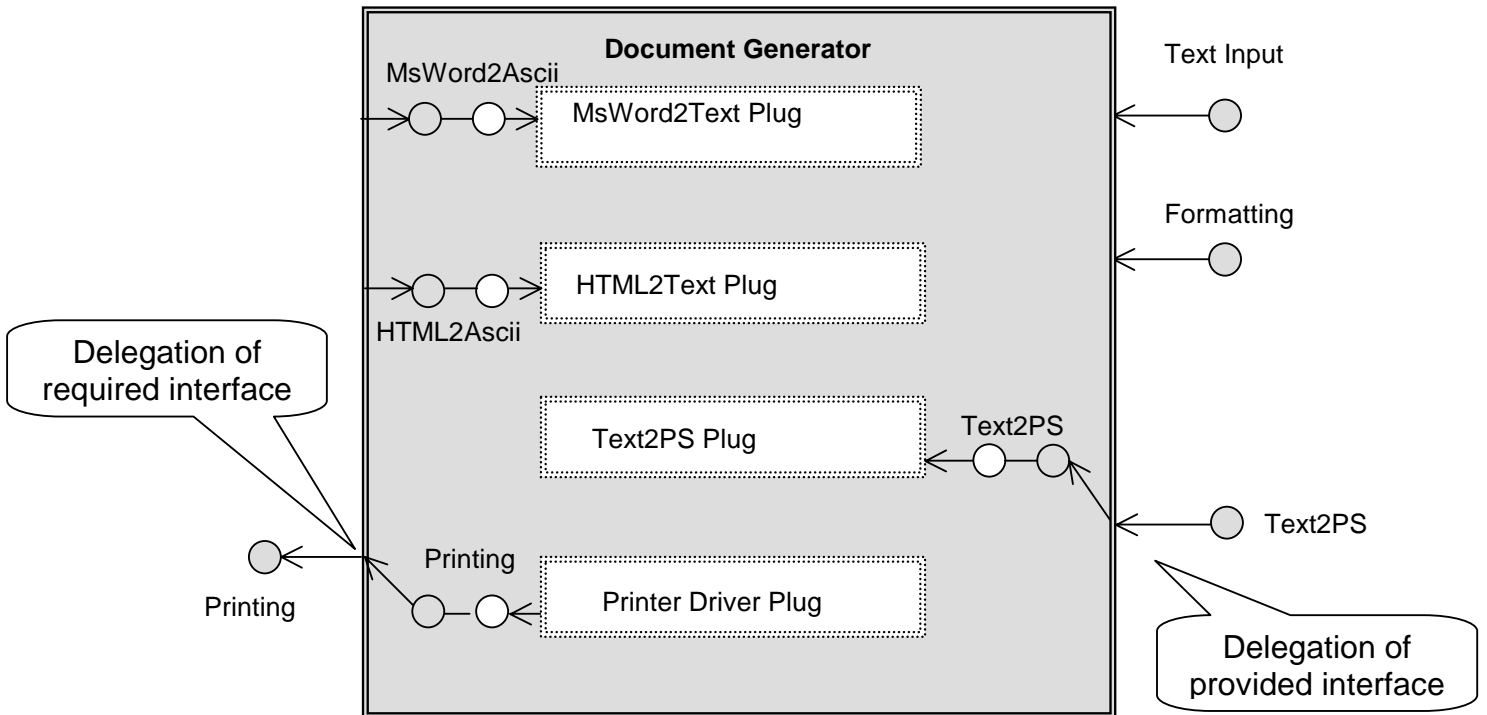
Ultimately, a system can be viewed as the top composite component, containing all components that are used to realize the system. The environment of a system can also be viewed as a subcomponent of the system itself, or as a component of which the system requires a number of services.

4.6. Notation for message delegation

When a composite component is decomposed into a number of smaller subcomponents, these subcomponents can provide a number of services directly to the users of the composite component. As such, a provided interface of the composite component can be delegated to a provided interface of a subcomponent. This means that any incoming messages through a specific provided interface of the composite component are automatically redirected to the provided interface of that subcomponent.

Also, required interfaces of the subcomponents can be delegated to a required interface from the composite component. This means that any outgoing messages through a specific required interface of the subcomponent are automatically redirected to the required interface of the composite component.

To define a message delegation through a provided interface (from an outer component through a composite component to a subcomponent) or through a required interface (from a subcomponent through a composite component to an outer component), the interface arrows inside and outside the composite component must be directly connected.



4.7. Notation for decomposition into classes and objects

Components cannot be decomposed indefinitely into other components. Ultimately, a component has to be realized through the use of real code, which can be of the form of objects and classes. To model the internals of a component, a component can also be decomposed into classes and objects. As such, a large component can be implemented using a number of objects to realize the total component functionality.

An UML composition aggregation is also used to realize a component into a number of objects. As such, a component can be implemented as a composite component, containing a number of sub-objects instead of sub-components. The provided interfaces of a component must be mapped to messages to one or a number of components. The *Facade* design pattern can be useful for delegating the incoming messages to the appropriate objects.

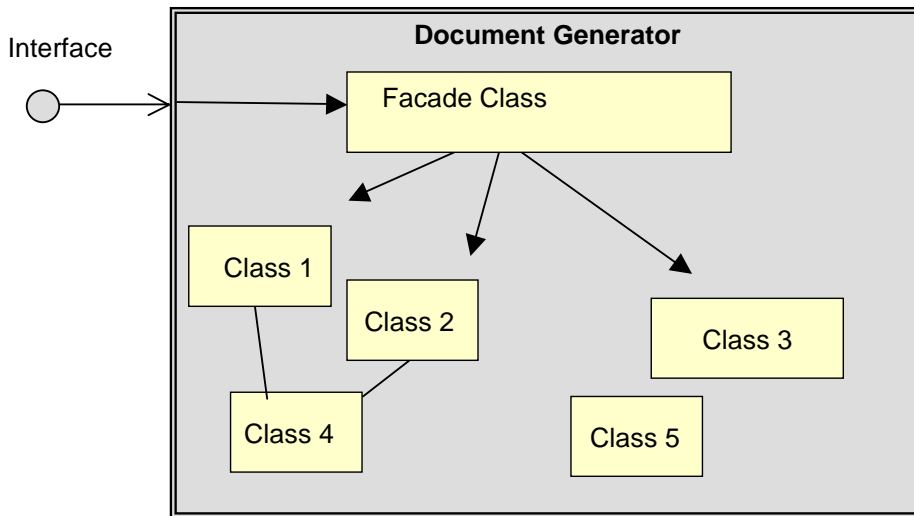


Figure C-19: DESS OO component design notation

4.8. Notation for decomposition into subcomponents, classes and objects

It is also possible that a component implementation makes use of both subcomponents and objects. In this case, the internal decomposition view of a component blueprint can highlight other component blueprints as well as plain classes. To model the internals of a component, a large component can be implemented using a number of other components as well as other objects to realize the total component functionality.

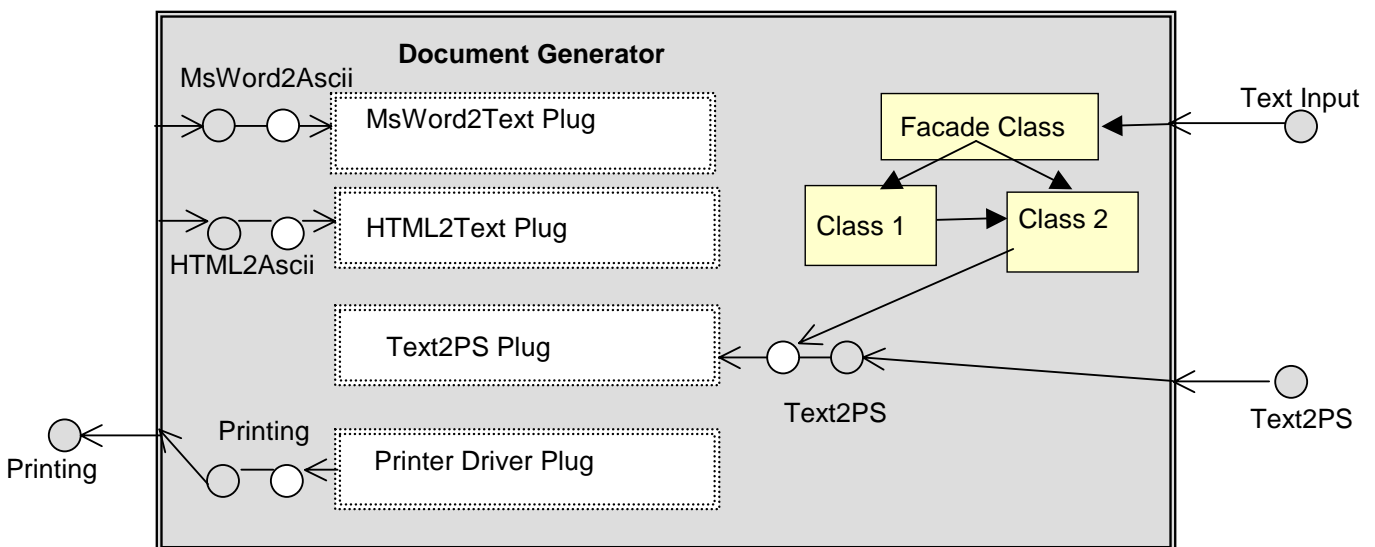


Figure C-20: DESS mixed framework & OO component design notation

4.9. Notation for the outside view of a component

The outside provides and requires views of a component can be specified by classical UML models, texts or other artifacts. Use cases, scenarios, class models and textual requirement specifications can be used. It is even possible that a component is dependent on a specific implementation version of another component. When this is the case, it should also be specified in the required outside view.

4.10. Notation for the unique component and interface identification

A unique ID is necessary to distinguish one specific version from another. The identification can contain different levels, depending on the evolution of the component. A distinction is necessary for

- Interface version
- Component specification version
- Component implementation version

Run-time component instantiation