8 Components

8.1 Overview

The Components package specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, the package specifies a component as a modular unit with well-defined interfaces that is replaceable within its environment. The component concept addresses the area of component-based development and component-based system structuring, where a component is modeled throughout the development life cycle and successively refined into deployment and run-time.

An important aspect of component-based development is the reuse of previously constructed components. A component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and/or required interfaces (potentially exposed via ports), and its internals are hidden and inaccessible other than as provided by its interfaces. Although it may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible. As a result, components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together via their provided and required interfaces. The aspects of autonomy and reuse also extend to components at deployment time. The artifacts that implement component are intended to be capable of being deployed and re-deployed independently, for instance to update an existing system.

The Components package supports the specification of both logical components (e.g., business components, process components) and physical components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around components will be developed for specific component technologies and associated hardware and software environments.

Basic Components

The BasicComponents package focuses on defining a component as an executable element in a system. It defines the concept of a component as a specialized class that has an external specification in the form of one or more provided and required interfaces, and an internal implementation consisting of one or more classifiers that realize its behavior. In addition, the BasicComponents package defines specialized connectors for 'wiring' components together based on interface compatibility.

Packaging Components

The PackagingComponents package focuses on defining a component as a coherent group of elements as part of the development process. It extends the concept of a basic component to formalize the aspects of a component as a 'building block' that may own and import a (potentially large) set of model elements.

8.2 Abstract syntax

Figure 8.1 shows the dependencies of the Component packages.

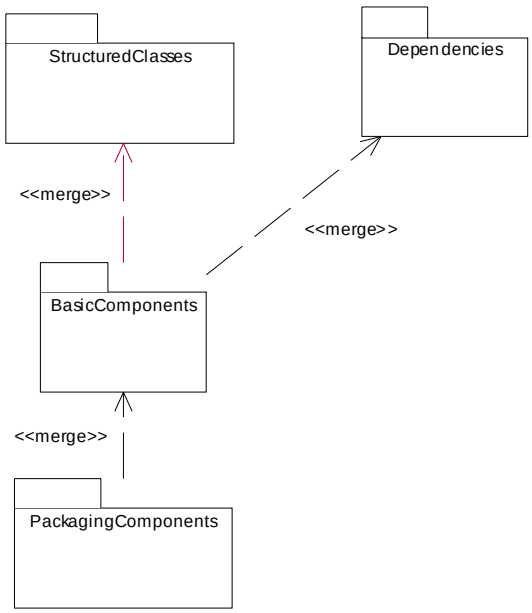


Figure 8.1 - Dependencies between packages described in this chapter (transitive dependencies to Kernel and Interfaces packages are not shown).

Package BasicComponents

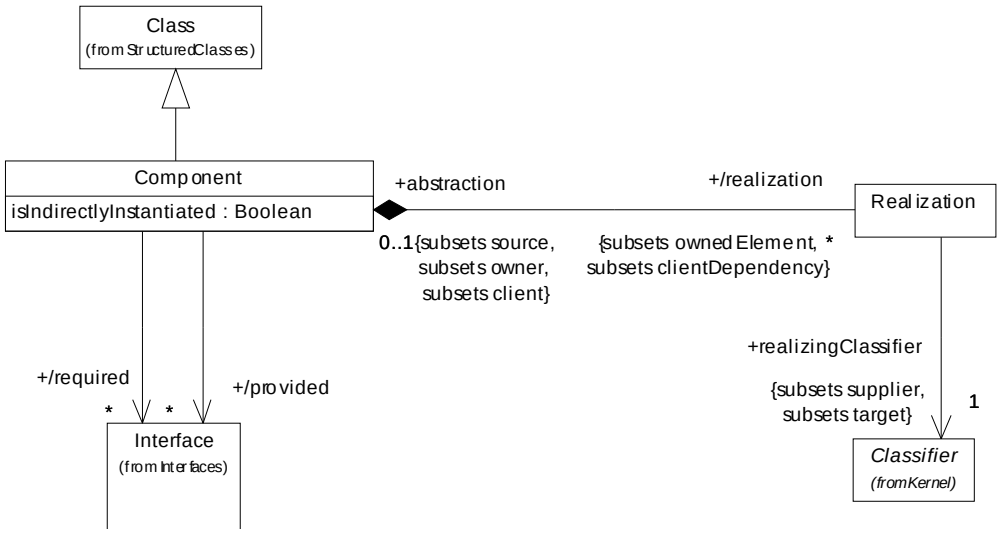


Figure 8.2 - The metaclasses that define the basic Component construct

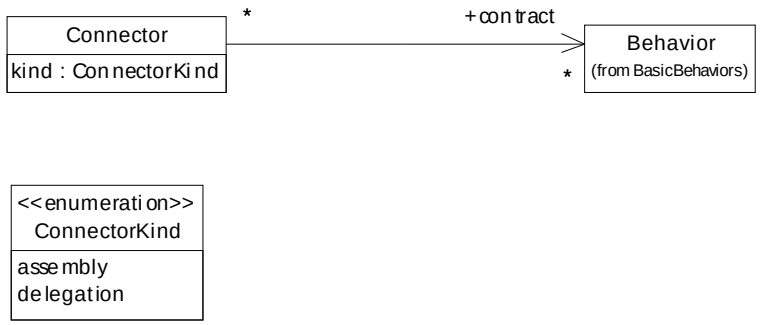


Figure 8.3 - The metaclasses that define the component wiring constructs

Package PackagingComponents

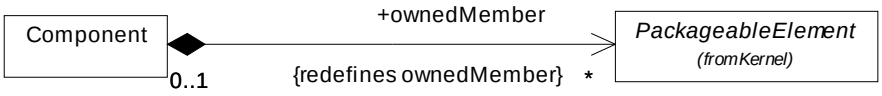


Figure 8.4 - The packaging capabilities of Components

8.3 Class Descriptions

8.3.1 Component (from BasicComponents, PackagingComponents)

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces.

A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parameterize the component's execution. (See Deployment chapter).

Generalizations

- "Class (from StructuredClasses)" on page 160

Description

BasicComponents

A component is a subtype of Class that provides for a Component having attributes and operations, and being able to participate in Associations and Generalizations. A Component may form the abstraction for a set of realizingClassifiers that realize its behavior. In addition, because a Class itself is a subtype of an EncapsulatedClassifier, a Component may optionally have an internal structure and own a set of Ports that formalize its interaction points.

A component has a number of provided and required Interfaces, that form the basis for wiring components together, either using Dependencies, or by using Connectors. A provided Interface is one that is either implemented directly by the component or one of its realizingClassifiers, or it is the type of a provided Port of the Component. A required interface is designated by a Usage Dependency from the Component or one of its realizingClassifiers, or it is the type of a required Port.

PackagingComponents

A component is extended to define the grouping aspects of packaging components. This defines the Namespace aspects of a Component through its inherited ownedMember and elementImport associations. In the namespace of a component, all model elements that are involved in or related to its definition are either owned or imported explicitly. This may include, for example, UseCases and Dependencies (e.g., mappings), Packages, Components, and Artifacts.

Attributes

BasicComponents

- isIndirectlyInstantiated : Boolean {default = true}
  The kind of instantiation that applies to a Component. If *false*, the component is instantiated as an addressable object. If *true*, the Component is defined at design-time, but at run-time (or execution-time) an object specified by the Component does not exist, that is, the component is instantiated indirectly, through the instantiation of its realizing classifiers or parts. Several standard stereotypes use this meta attribute (e.g., «specification», «focus», «subsystem»).

Associations

BasicComponents

- /provided: Interface [*]
  The interfaces that the component exposes to its environment. These interfaces may be Realized by the Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports. The provided interfaces association is a derived association:

  ```
  context Component::provided derive:
    let implementedInterfaces = self.implementation->collect(impl|impl.contract) and
    let realizedInterfaces = RealizeInterfaces(self) and
    let realizingClassifierInterfaces = RealizedInterfaces(self.realizingClassifier) and
    let typesOfRequiredPorts = self.ownedPort.provided in
      (((implementedInterfaces->union(realizedInterfaces)->union(realizingClassifierInterfaces))->
        union(typesOfRequiredPorts))->asSet()
  ```

- /required: Interface [*]
  The interfaces that the component requires from other components in its environment in order to be able to offer its full set of provided functionality. These interfaces may be Used by the Component or any of its realizingClassifiers, or they may be the Interfaces that are required by its public Ports. The required interfaces association is a derived association:

  ```
  context Component::required derive:
    let usedInterfaces = UsedInterfaces(self) and
    let realizingClassifierUsedInterfaces = UsedInterfaces(self.realizingClassifier) and
    let typesOfUsedPorts = self.ownedPort.required in
      ((usedInterfaces->union(realizingClassifierUsedInterfaces))->
        union(typesOfUsedPorts))->asSet()
  ```

- /realization: Realization [*]
  The set of Realizations owned by the Component. Realizations reference the Classifiers of which the Component is an abstraction (i.e., that realize its behavior).

PackagingComponents

- ownedMember: PackageableElement [*]
  The set of PackageableElements that a Component owns. In the namespace of a component, all model elements that are involved in or related to its definition may be owned or imported explicitly. These may include, for example, Classes, Interfaces, Components, Packages, Use cases, Dependencies (e.g., mappings), and Artifacts.

Constraints

No further constraints

Additional Operations

[1] Utility returning the set of realized interfaces of a component:

```
def: RealizedInterfaces : (classifier : Classifier) : Interface = (classifier.clientDependency->
    select(dependency|dependency.oclIsKindOf(Realization) and dependency.supplier.oclIsKindOf(Interface)))->
      collect(dependency|dependency.client)
```

[2] Utility returning the set of required interfaces of a component:

```
def: UsedInterfaces : (classifier : Classifier) : Interface = (classifier.supplierDependency->
   select(dependency|dependency.oclIsKindOf(Usage) and dependency.supplier.oclIsKindOf(interface)))->
      collect(dependency|dependency.supplier)
```

Semantics

A component is a self contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces.

A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality.

The required and provided interfaces of a component allow for the specification of structural features such as attributes and association ends, as well as behavioral features such as operations and events. A component may implement a provided interface directly, or, its realizing classifiers may do so. The required and provided interfaces may optionally be organized through ports, these enable the definition of named sets of provided and required interfaces that are typically (but not always) addressed at run-time.

A component has an *external view* (or “black-box” view) by means of its publicly visible properties and operations. Optionally, a behavior such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit. Other behaviors may also be associated with interfaces or connectors to define the ‘contract’ between participants in a collaboration (e.g., in terms of use case, activity, or interaction specifications).

The wiring between components in a system or other context can be structurally defined by using dependencies between component interfaces (typically on structure diagrams). Optionally, a more detailed specification of the structural collaboration can be made using parts and connectors in composite structures, to specify the role or instance level collaboration between components (See Chapter Composite Structures).

A component also has an *internal view* (or “white-box” view) by means of its private properties and realizing classifiers. This view shows how the external behavior is realized internally. The mapping between external and internal view is by means of dependencies (on structure diagrams), or delegation connectors to internal parts (on composite structure diagrams). Again, more detailed behavior specifications such as interactions and activities may be used to detail the mapping from external to internal behavior.

A number of UML standard stereotypes exist that apply to component. For example, «subsystem» to model large-scale components, and «specification» and «realization» to model components with distinct specification and realization definitions, where one specification may have multiple realizations (see the UML Standard Elements Appendix).

Notation

A component is shown as a Classifier rectangle with the keyword «component». Optionally, in the right hand corner a component icon can be displayed. This is a classifier rectangle with two smaller rectangles protruding from its left hand side.

Figure 8.5 - A Component with one provided interface

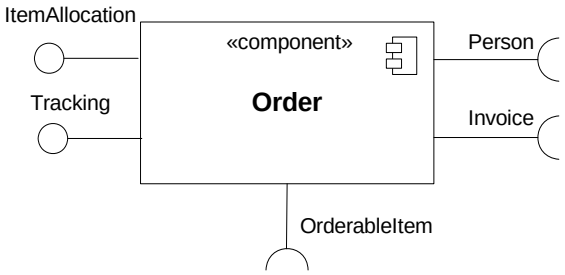


Figure 8.6 - A Component with two provided and three required interfaces

An external view of a Component is by means of Interface symbols sticking out of the Component box (external, or black-box view). Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box (for scalability, tools may offer way of listing and abbreviating component properties and behavior).

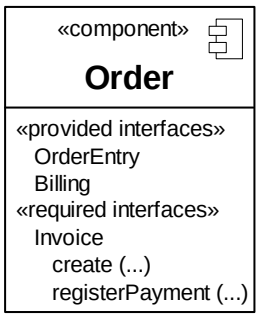


Figure 8.7 - Black box notation showing a listing of the properties of a component

For displaying the full signature of an interface of a component, the interfaces can also be displayed as typical classifier rectangles that can be expanded to show details of operations and events.

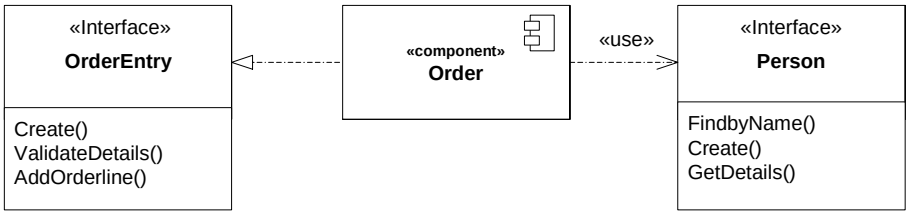


Figure 8.8 - Explicit representation of the provided and required interfaces, allowing interface details such as operation to be displayed (when desired).

An internal, or white box view of a Component is where the realizing classifiers are listed in an additional compartment. Compartments may also be used to display a listing of any parts and connectors, or any implementing artifacts.

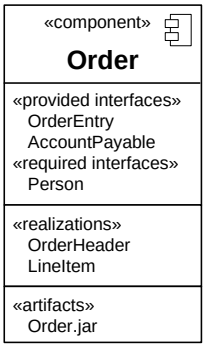


Figure 8.9 - A white-box representation of a component

The internal classifiers that realize the behavior of a component may be displayed by means of general dependencies. Alternatively, they may be nested within the component shape.

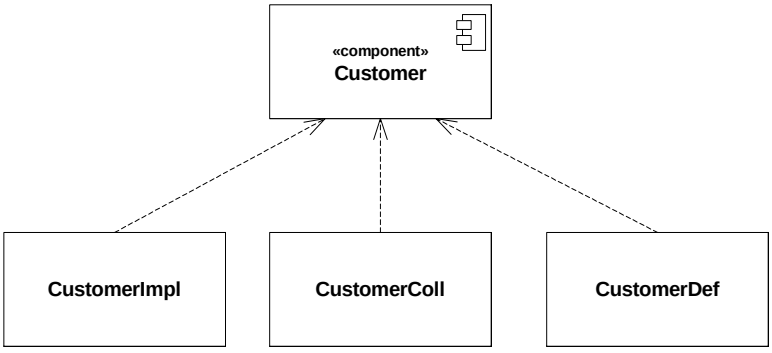


Figure 8.10 - A representation of the realization of a complex component

Alternatively, the internal classifiers that realize the behavior of a component may be displayed nested within the component shape.

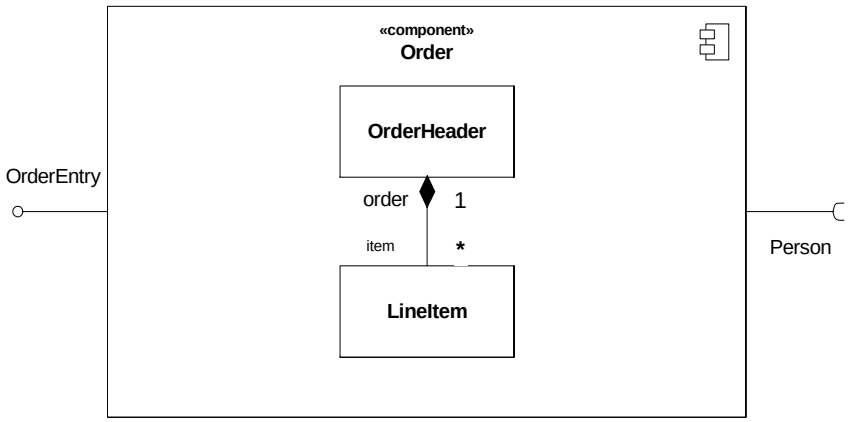


Figure 8.11 - An alternative nested representation of a complex component

If more detail is required of the role or instance level containment of a component, then an internal structure consisting of parts and connectors can be defined for that component. This allows, for example, explicit part names or connector names to be shown in situations where the same Classifier (Association) is the type of more than one Part (Connector). That is, the Classifier is instantiated more than once inside the component, playing different roles in its realization. Optionally, specific instances (InstanceSpecifications) can also be referred to as in this notation.

Interfaces that are exposed by a Component and notated on a diagram, either directly or though a port definition, may be inherited from a supertype component. These interfaces are indicated on the diagram by preceding the name of the interface by a forward slash. An example of this can be found in Figure 8.14, where "/orderedItem" is an interface that is implemented by a supertype of the Product component.

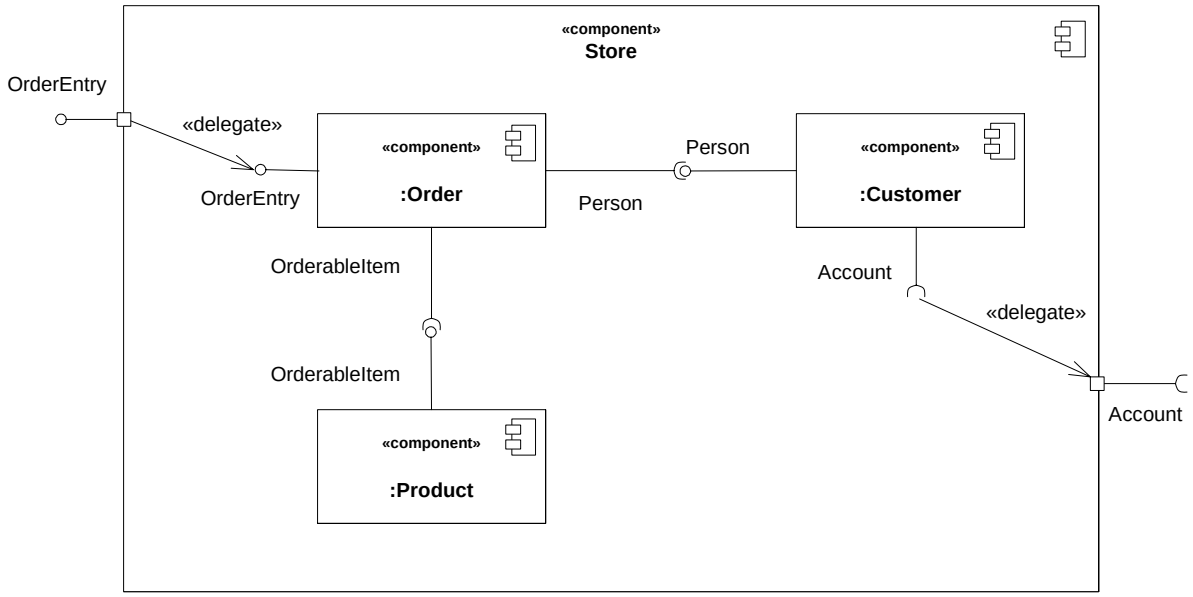


Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.

Artifacts that implement components can be connected to them by physical containment or by an «implement» relationship, which is an instance of the meta association between Component and Artifact.

Examples

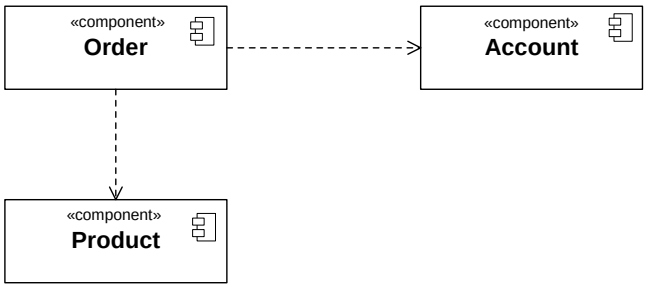


Figure 8.13 - Example of an overview diagram showing components and their general dependencies

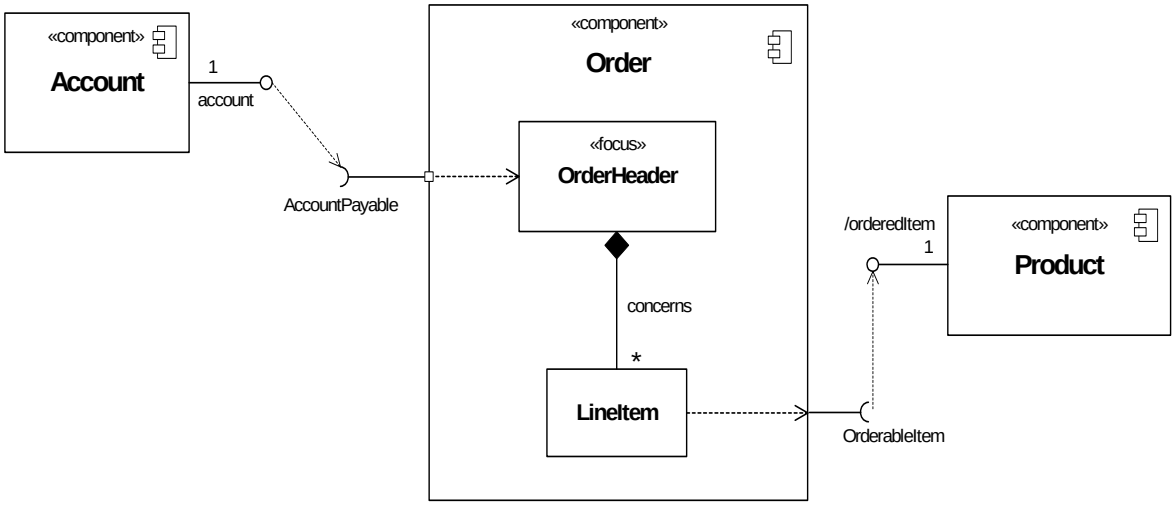


Figure 8.14 - Example of a platform independent model of a component, its provided and required interfaces, and wiring through dependencies on a structure diagram.

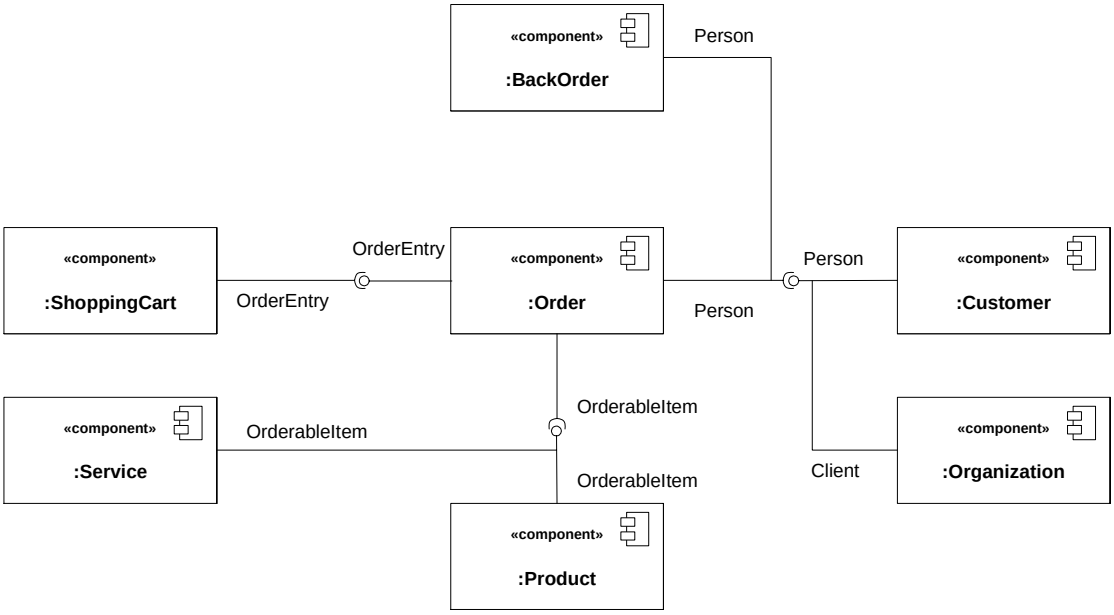


Figure 8.15 - Example of a composite structure of components, with connector wiring between provided and required interfaces of parts (Note: "Client" interface is a subtype of "Person").

The wiring of components can be represented on structure diagrams by means of classifiers and dependencies between them (Note: the ball-and-socket notation from Figure 8.15 may be used as a notation option for dependency based wiring). On composite structure diagrams, detailed wiring can be performed at the role or instance level by defining parts and connectors.

Changes from previous UML

The following changes from UML 1.x have been made.

The component model has made a number of implicit concepts from the UML 1.x model explicit, and made the concept more applicable throughout the modeling life cycle (rather than the implementation focus of UML 1.x). In particular, the "resides" relationship from 1.x relied on namespace aspects to define both namespace aspects as well as 'residence' aspects. These two aspects have been separately modeled in the UML metamodel in 2.0. The basic residence relationship in 1.x maps to the realizingClassifiers relationship in 2.0. The namespace aspects are defined through the basic namespace aspects of Classifiers in UML 2.0, and extended in the PackagingComponents metamodel for optional namespace relationships to elements other than classifiers.

In addition, the Component construct gains the capabilities from the general improvements in CompositeStructures (around Parts, Ports, and Connectors).

In UML 2.0, a Component is notated by a classifier symbol that no longer has two protruding rectangles. These were cumbersome to draw and did not scale well in all circumstances. Also, they interfered with any interface symbols on the edge of the Component. Instead, a «component» keyword notation is used in UML 2.0. Optionally, a component icon that is similar to the UML 1.4 icon can still be used in the upper right-hand corner of the component symbol. For backward compatibility reasons, the UML 1.4 notation with protruding rectangles can still be used.

8.3.2 Connector (from BasicComponents)

The connector concept is extended in the Components package to include interface based constraints and notation.

A *delegation* connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling.

An *assembly* connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Generalizations

- "Connector (from InternalStructures)" on page 170 (*merge increment*)

Description

In the metamodel, a connector kind attribute is added to the Connector metaclass. Its value is an enumeration type with valid values "assembly" or "delegation."

Attributes

BasicComponents

- kind : ConnectorKind Indicates the kind of connector.

Associations

No additional associations

Constraints

[1] A delegation connector must only be defined between used Interfaces or Ports of the same kind (e.g., between two provided Ports or between two required Ports).

[2] If a delegation connector is defined between a used Interface or Port and an internal Part Classifier, then that Classifier must have an "implements" relationship to the Interface type of that Port.

[3] If a delegation connector is defined between a source Interface or Port and a target Interface or Port, then the target Interface must support a signature compatible subset of Operations of the source Interface or Port.

[4] In a complete model, if a source Port has delegation connectors to a set of delegated target Ports, then the union of the Interfaces of these target Ports must be signature compatible with the Interface that types the source Port.

[5] An assembly connector must only be defined from a required Interface or Ports to a provided Interface or Port.

Semantics

A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by another instance that has "compatible" capabilities. This may be another Component or a (simple) Class. The latter situation is modeled through a delegation connector from a Component Interface or Port to a contained Class that functions as a Part. In that case, the Class must have an implements relationship to the Interface of the Port.

Delegation connectors are used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only).

A port may delegate to a set of ports on subordinate components. In that case, these subordinate ports must collectively offer the delegated functionality of the delegating port. At execution time, signals will be delivered to the appropriate port. In the cases where multiple target ports support the handling of the same signal, the signal will be delivered to all these subordinate ports.

The execution time semantics for an assembly connector are that signals travel along an instance of a connector, originating in a required port and delivered to a provided port. Multiple connectors directed from a single required interface or port to provided interfaces on different components indicates that the instance that will handle the signal will be determined at execution time. Similarly, multiple required ports that are connected to a single provided port indicates that the request may originate from instances of different component types.

The interface compatibility between provided and required ports that are connected enables an existing component in a system to be replaced by one that (minimally) offers the same set of services. Also, in contexts where components are used to extend a system by offering existing services, but also adding new functionality, assembly connectors can be used to link in the new component definition. That is, by adding the new component type that offers the same set of services as existing types, and defining new assembly connectors to link up its provided and required ports to existing ports in an assembly.

Notation

A delegation connector is notated as a Connector from the delegating source Port to the handling target Part, and vice versa for required Interfaces or Ports.

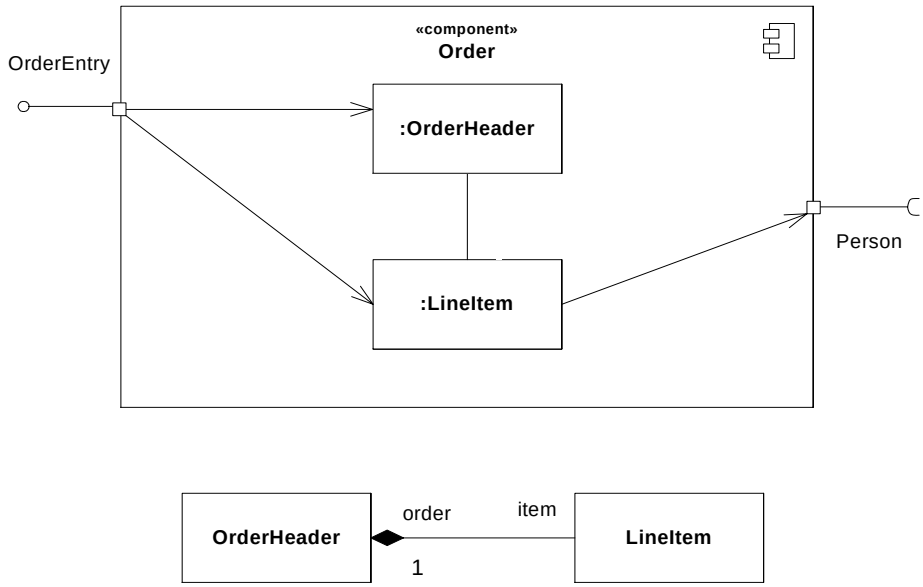


Figure 8.16 - Delegation connectors connect the externally provided interfaces of a component to the parts that realize or require them.

An assembly connector is notated by a “ball-and-socket” connection between a provided interface and a required interface. This notation allows for succinct graphical wiring of components, a requirement for scaling in complex systems.

When this notation is used to connect “complex” ports that are typed by multiple provided and/or required interfaces, the various interfaces are listed as an ordered set, designated with {provided} or {required} if needed.

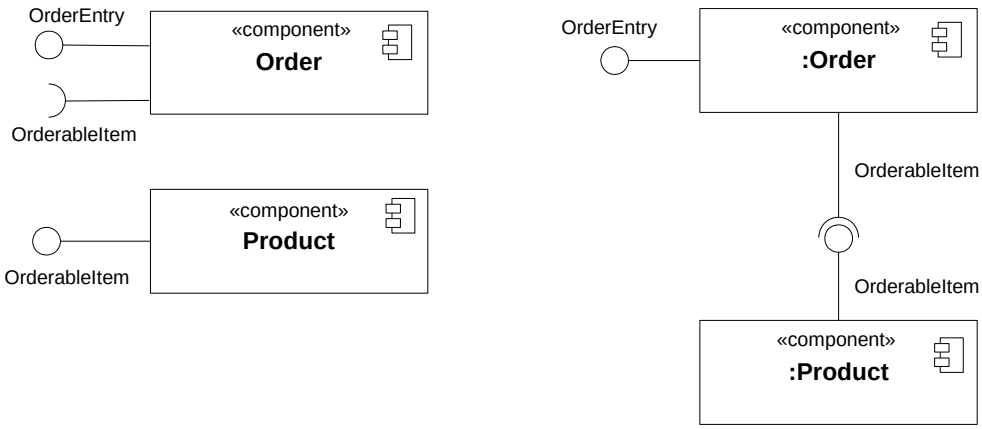


Figure 8.17 - An assembly connector maps a required interface of a component to a provided interface of another component in a certain context (definition of components, e.g., in a library on the left, an assembly of those components on the right).

In a system context where there are multiple components that provide or require a particular interface, a notation abstraction can be used that combines by joining the multiple connectors. This abstraction is similar to the one defined for aggregation and subtyping relationships.

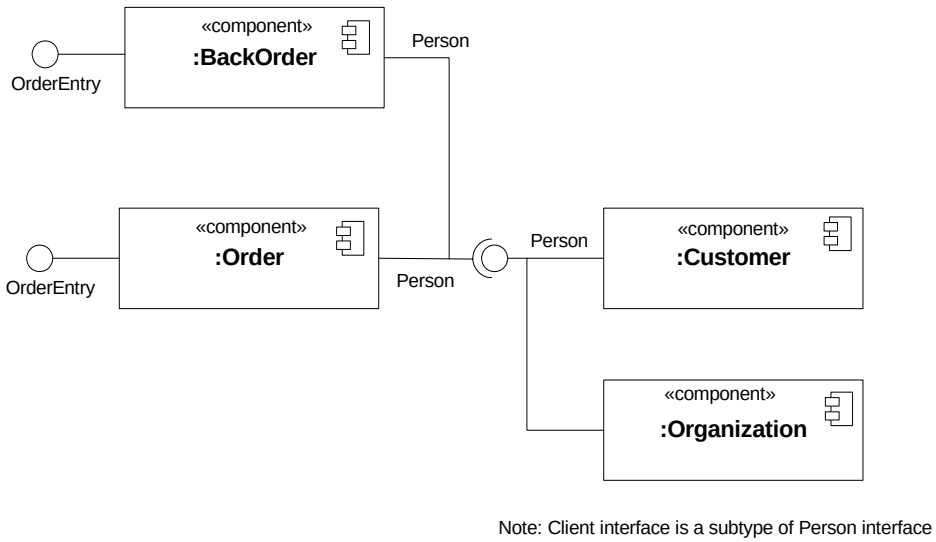


Figure 8.18 - As a notation abstraction, multiple wiring relationships can be visually grouped together in a component assembly.

Changes from previous UML

The following changes from UML 1.x have been made — Connector is not defined in UML 1.4.

8.3.3 ConnectorKind (from BasicComponents)

Generalizations

None

Description

ConnectorKind is an enumeration of the following literal values:

- assembly Indicates that the connector is an assembly connector.
- delegation Indicates that the connector is a delegation connector.

8.3.4 Realization (from BasicComponents)

The Realization concept is specialized in the Components package to (optionally) define the Classifiers that realize the contract offered by a component in terms of its provided and required interfaces. The component forms an abstraction from these various Classifiers.

Generalizations

- “Realization (from Dependencies)” on page 124 *(merge increment)*

Description

In the metamodel, a Realization is a subtype of Dependencies::Realization.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A component’s behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

It should be noted that for the purpose of applications that require multiple different sets of realizations for a single component specification, a set of standard stereotypes are defined in the UML Standard Profile. In particular, «specification» and «realization» are defined there for this purpose.

Notation

A component realization is notated in the same way as the realization dependency (i.e., as a general dashed line with an open arrow-head).

Changes from previous UML

The following changes from UML 1.x have been made: Realization is defined in UML 1.4 as a ‘free standing’ general dependency - it is not extended to cover component realization specifically. These semantics have been made explicit in UML 2.0.

8.4 Diagrams

Structure diagram

Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 8.1.

Table 8.1 - Graphic nodes included in structure diagrams

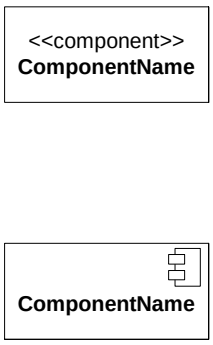
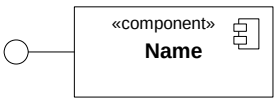
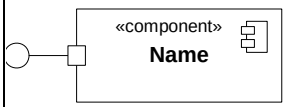
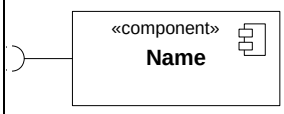
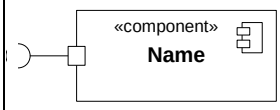
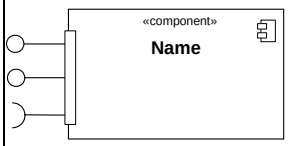
NODE TYPE	*NOTATION*	*REFERENCE*
Component		See “Component”
Component implements Interface		See “Interface”
Component has provided Port (typed by Interface)		See “Port”

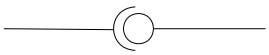
Table 8.1 - Graphic nodes included in structure diagrams

NODE TYPE	*NOTATION*	*REFERENCE*
Component uses Interface		See “Interface”
Component has required Port (typed by Interface)		See “Port”
Component has complex Port (typed by provided and required Interfaces)		See “Port”

Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 8.2.

Table 8.2 - Graphic nodes included in structure diagrams

PATH TYPE	*NOTATION*	*REFERENCE*
Assembly connector		See “assembly connector.” Also used as notation option for wiring between interfaces using Dependencies.

Variations

Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, showing instance specifications, or relationships between classes. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

Component diagram

The following nodes and edges are typically drawn in a component diagram:

- Component
- Interface

- Realization, Interface Realization, Usage Dependencies
- Class
- Artifact
- Port