A03. *Architecture Description in UML 2.0*

2014

Sungwon Kang

TOC

1. Architectural Constructs of UML 2.0

1.1 Structured Class

1) Classifier
2) Structured Classifier
3) Structured Class

1) Classifier

ClassifierName
...

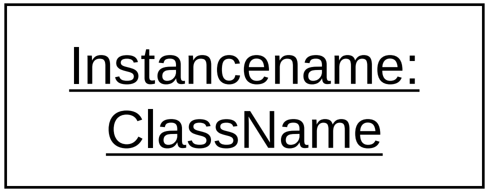
- "A *classifier* is a classification of instances — it describes a set of instances that have features in common."
- A classifier defines a type.
  - The type can be shown in << ... >> above the name.
- All instances of a classifier have values corresponding to the classifier's attributes.
- Notation
  - Classifier is an abstract model element, and so properly speaking has no notation.
  - Default: a solid-outline rectangle containing the classifier's name, and optionally with compartments containing features or other members
  - Some specializations of classifier have their own distinct notations.

1) Classifier - UML Models and What They Model

- A model contains three major categories of elements:
  1. A classifier describes a set of objects
  2. An event describes a set of possible occurrences
  3. A behavior describes a set of possible executions
- Each (circled concept) models individuals in an incarnation of the system being modeled.
- A model
  - *represents* sets of (but does not contain) objects, occurrences, and executions with similar properties.

=> Class, Actor, Interface, Data Type, Information Item, Signal, Association, Artifacts are all Classifiers.

1) Classifier - InstanceSpecification



- Instance name
- Name of an anonymous instance of an unnamed classifier is an underlined colon (‘:’).

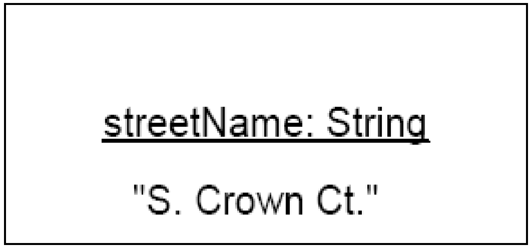


Figure 7.52 - Specification of an instance of String

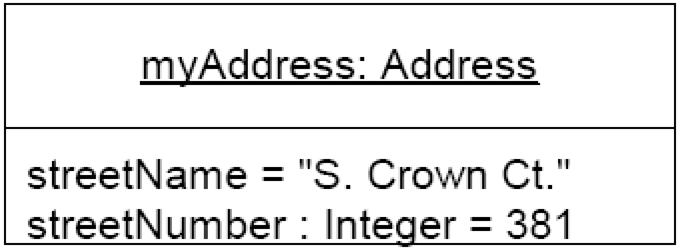


Figure 7.53 - Slots with values

2) Structured Classifier (1/3)

- Class-like entity that contains internal collaboration of its parts
  - A “part” is an instance
  - So it lives and dies as part of the lifetime of an object of the containing class.
- Parts are connected with a “connector”
- Not a simple runtime container of its parts.
  - Statically, a structured class owns its parts by composition relation and owns the connectors of the parts
  - parts and (assembly) connectors are created by the component
- May coordinate the activities and collaboration of the various part objects
  - In this case, a statechart is normally created for the structured class to control and mediate the interaction of the parts
- Used to represent “subsystem” and “component”
  - components usually contain further elements but classes cannot

2) Structured Classifier (2/3)

- A benefit of using a structured classifier, as opposed to a composition relation, is that constraints on the parts of a structured classifier only apply to instances of the parts

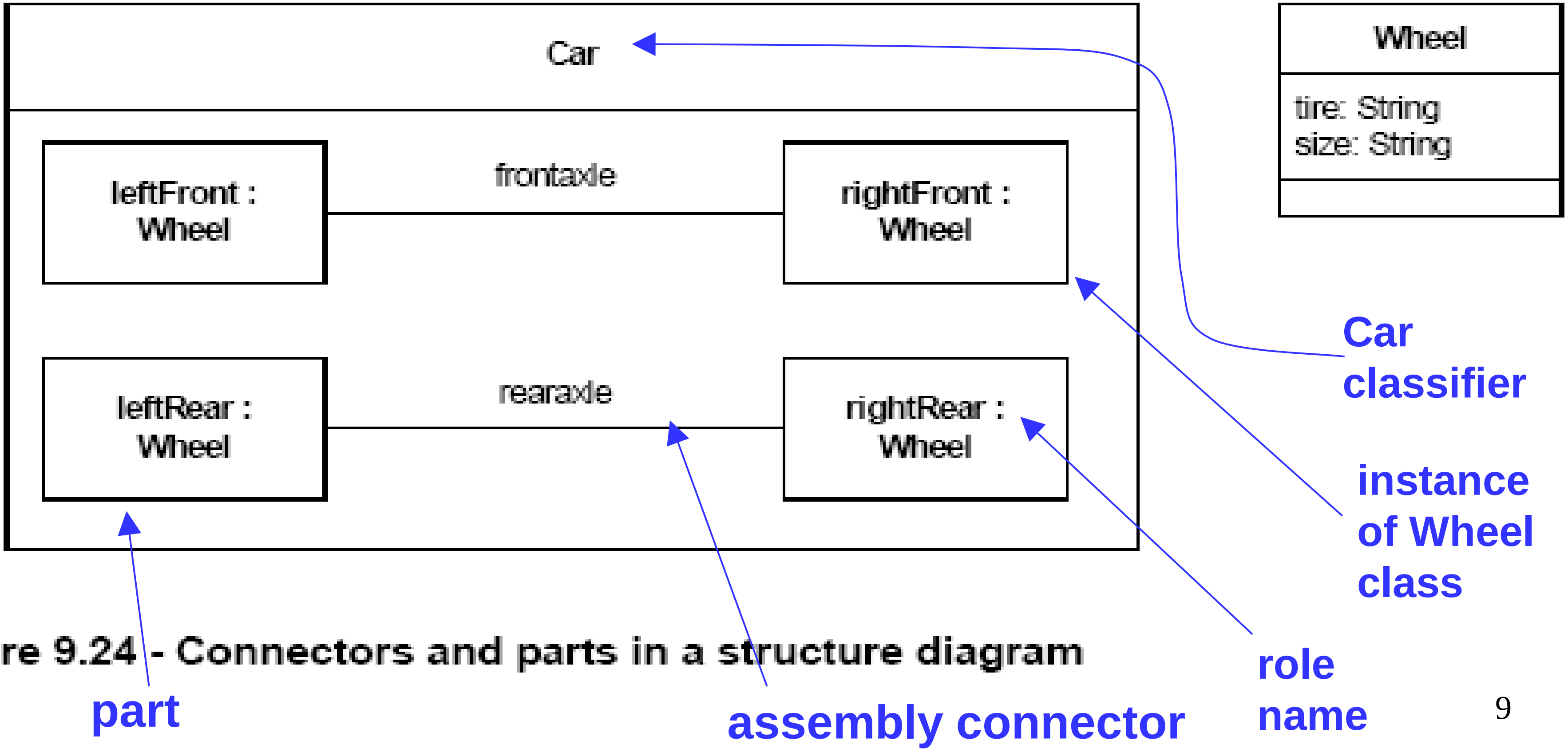


Figure 9.24 - Connectors and parts in a structure diagram

2) Structured Classifier (3/3)

- A system equivalent to Figure 9.24
- But relies on multiplicities to show the replication of the wheel and axle arrangement.

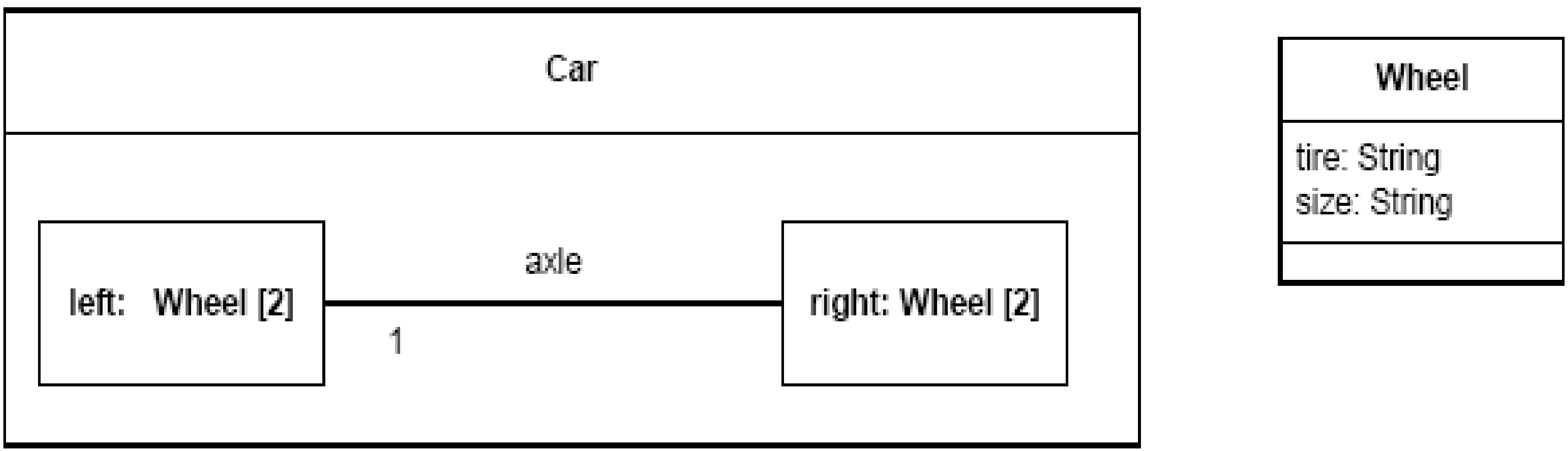


Figure 9.25 - Connectors and parts in a structure diagram using multiplicities

2) Structured Classifier – An Instance

- Not a structured Classifier !!!

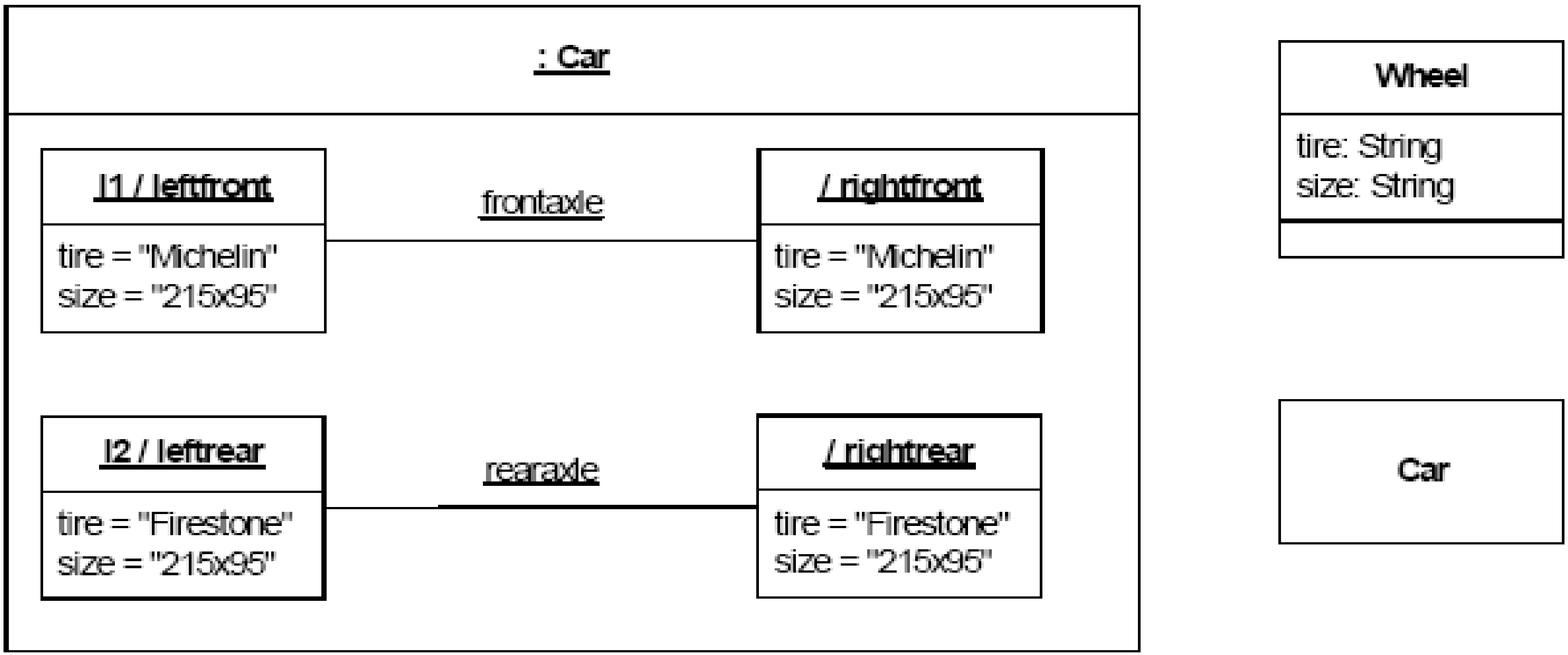


Figure 9.26 - A instance of the Car class

2) Structured Classifier (*)

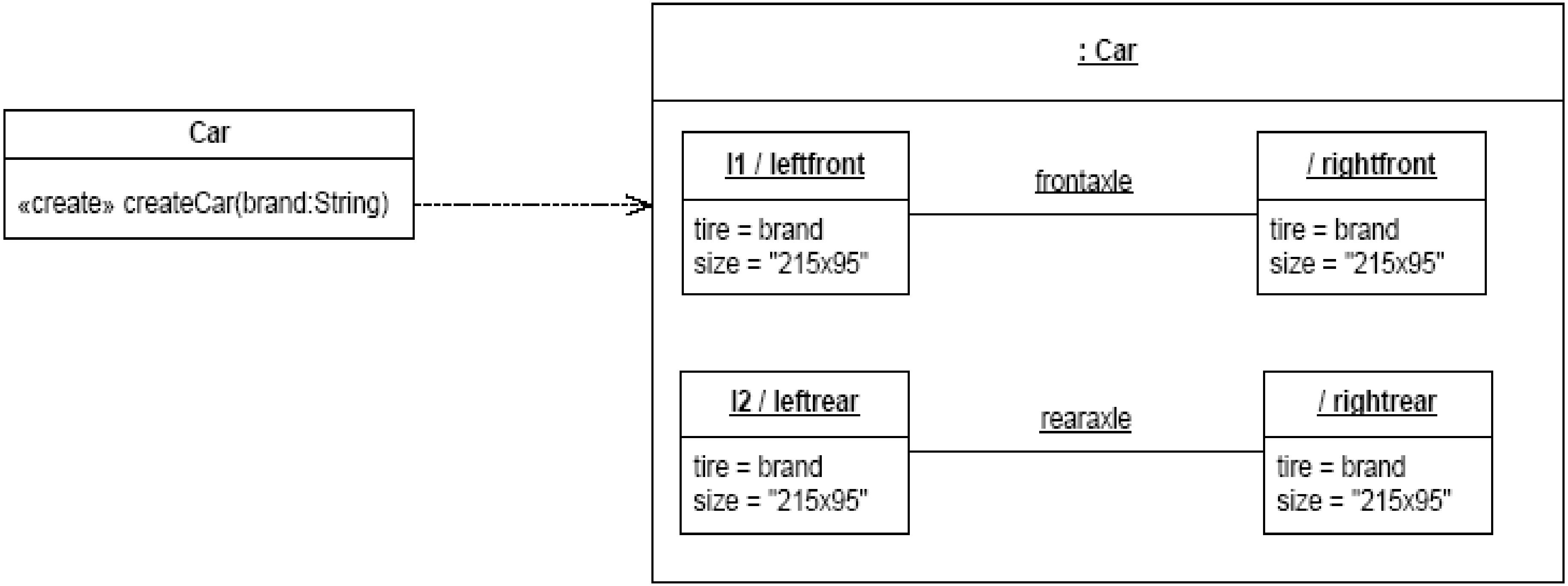


Figure 9.27 - A constructor for the Car class

3) Structured Class

- **Class that have ports or internal structure**
- Models containment hierarchies
- Is composed of parts with an explicit “nested” notation

- We first need some related concepts:

A) Class
B) Property
C) Part

These are all classifiers, i.e. types.

- Classifier concept is a generalization of class concept
- Class is a classifier that has attributes and operations

A) Class

- “... a kind of classifier whose features are attributes and operations."

- A class is shown using the classifier symbol.

- The most widely used classifier

  => The keyword “class” need not be shown in << ... >> above the name.

B) Property

- “Represents *a set of instances* that are ***owned*** by a containing classifier instance.” (UML 9.3.13)

Examples

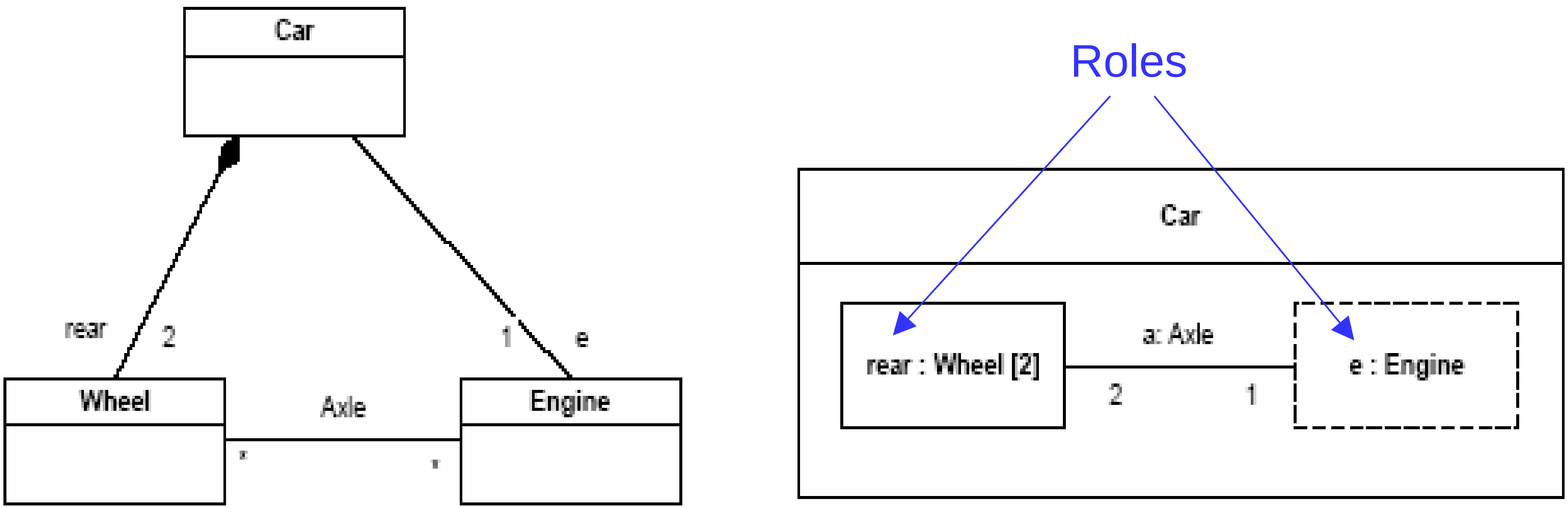
The containing instance will own four instances of the *Wheel* class by composition

The containing instance will **reference** one or two instances of the *Engine* class

Figure 9.21 - Property examples

B) Property - Example

- In (i), *Car* has a composition association with a class *Wheel* and an association to a class *Engine*.
- In (ii), the same is specified. However, in addition, *rear* and *e* belong to the internal structure of the class *Car*



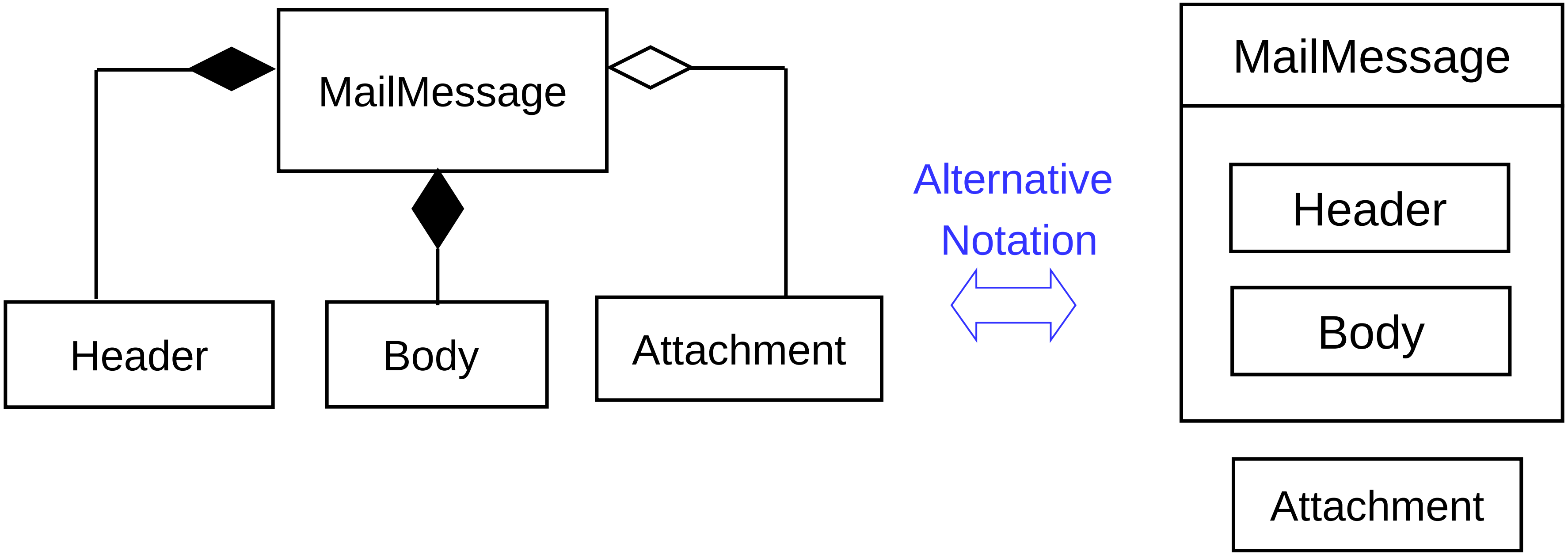
(i) (ii)

Figure 9.20 - Properties

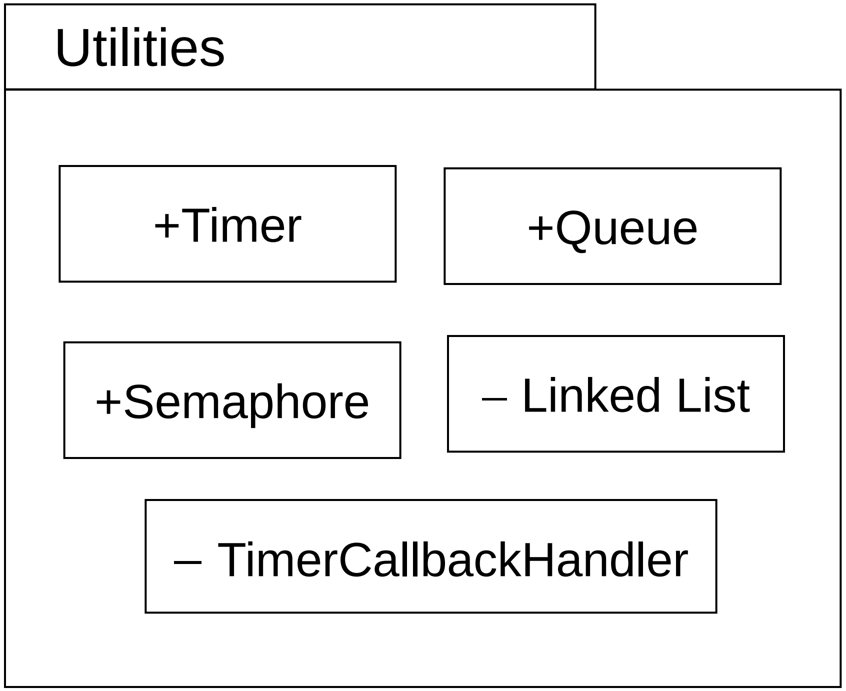
B) Property - Containment and Composition

- Composition is a stronger form of aggregation where the *part* object cannot exist without the *whole* object.

=> It is represented as an association between classes but it is really about relations between their objects.



B) Property - Containment and Package



B) Property - One Way to Interpret Multiplicities

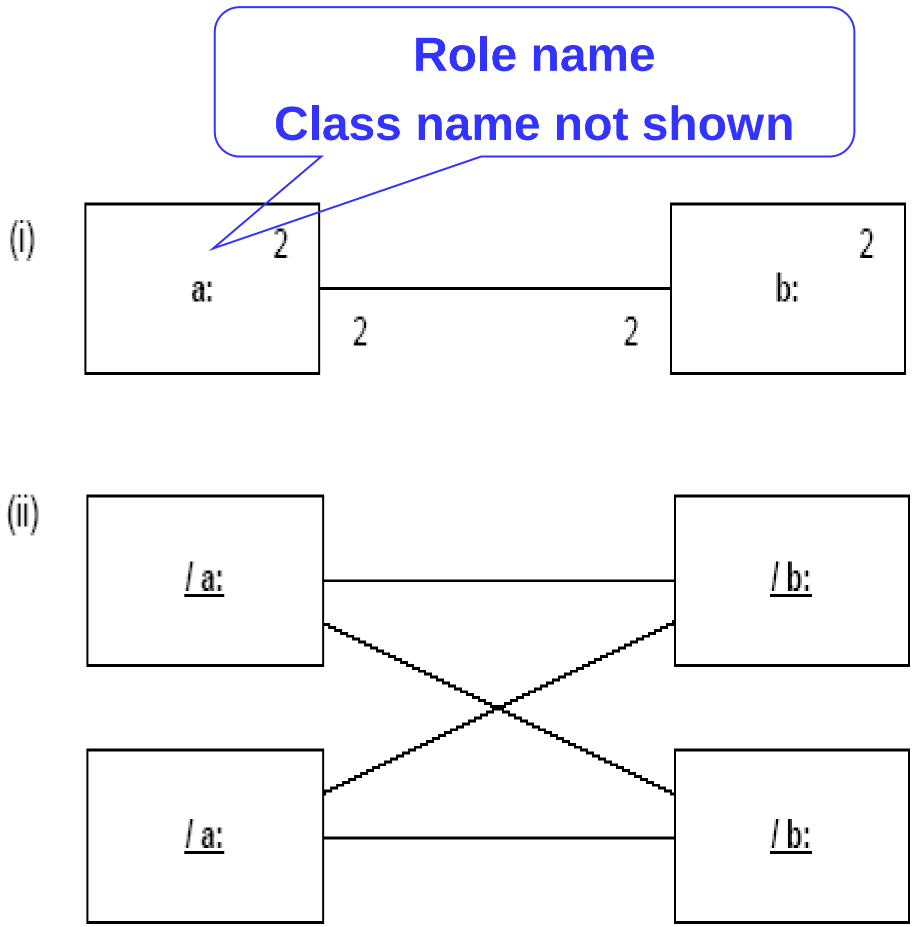


Figure 9.22 - “Star” connector pattern

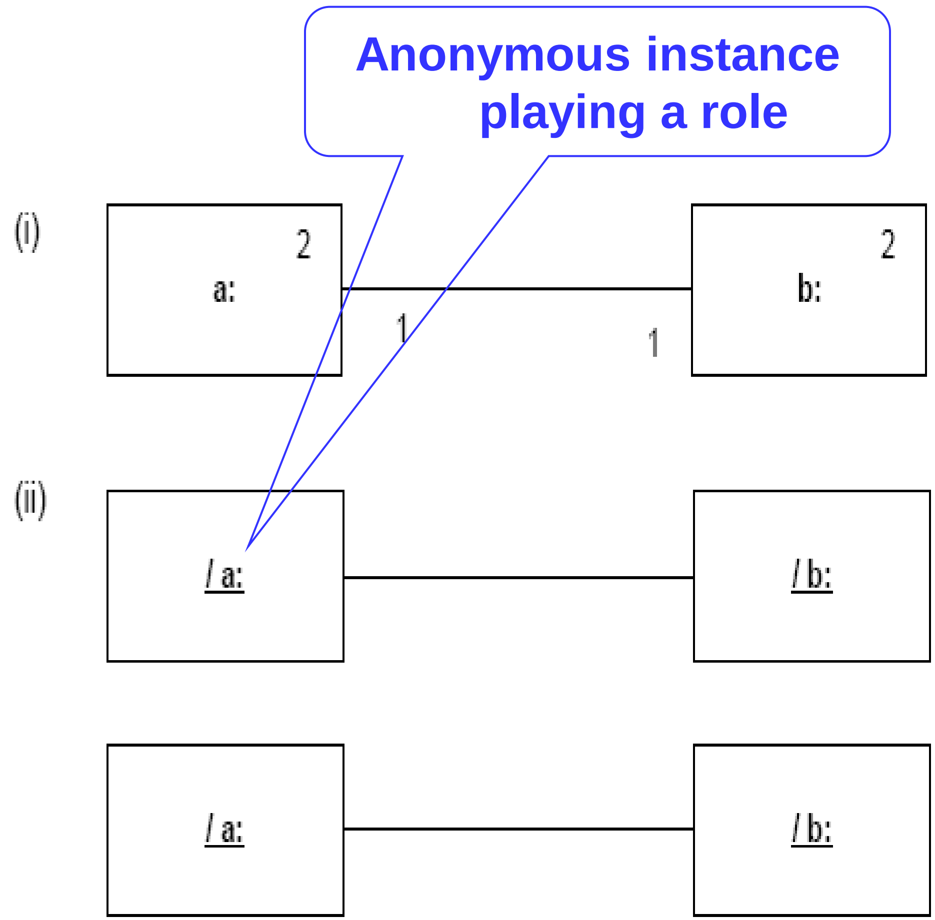


Figure 9.23 - “Array” connector pattern

B) Property – Writing name

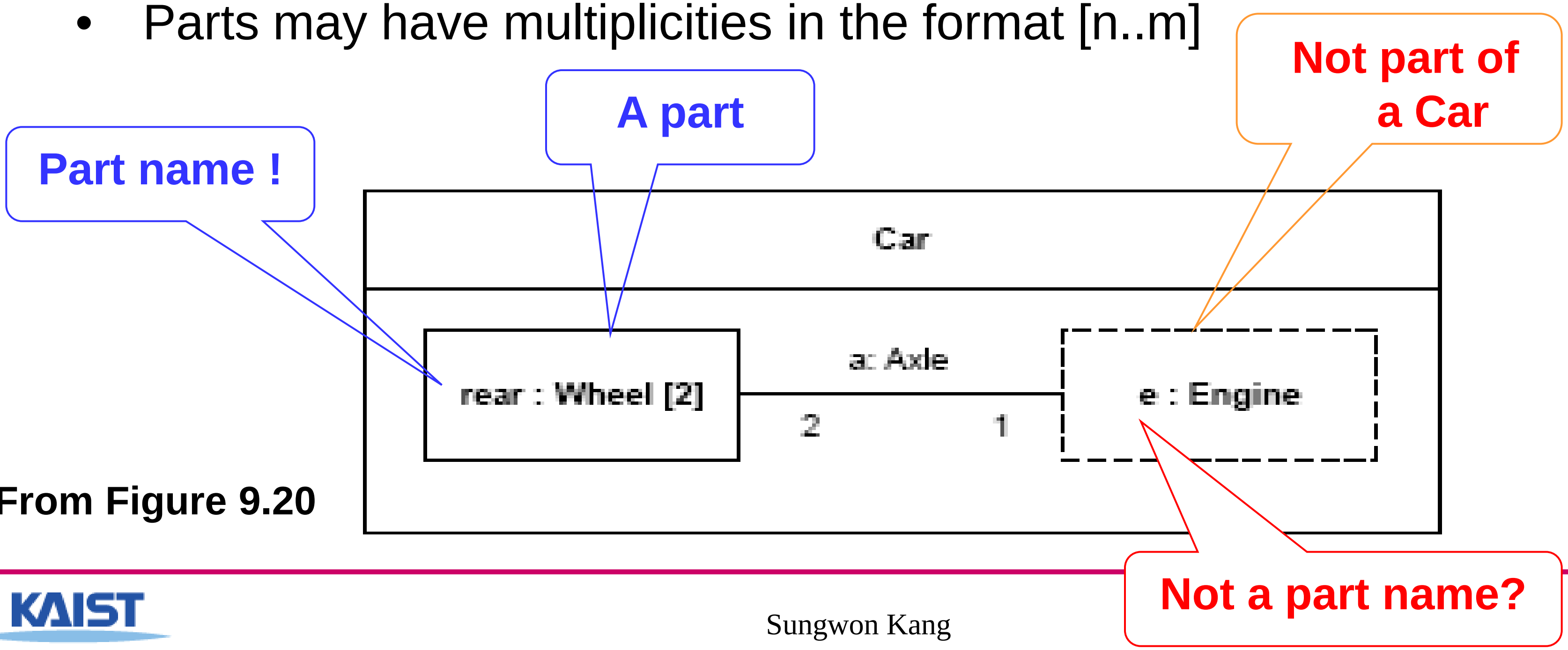
- The namestring of a role in an instance specification:

```
{<name> ['/'<rolename>] |'/'<rolename>}
       [':' <classifiername> [','<classifiername>]*]
```

C) Part (1/2)

- "A *part* declares that an instance of this classifier may contain a set of instances by composition." (UML 9.3.13)

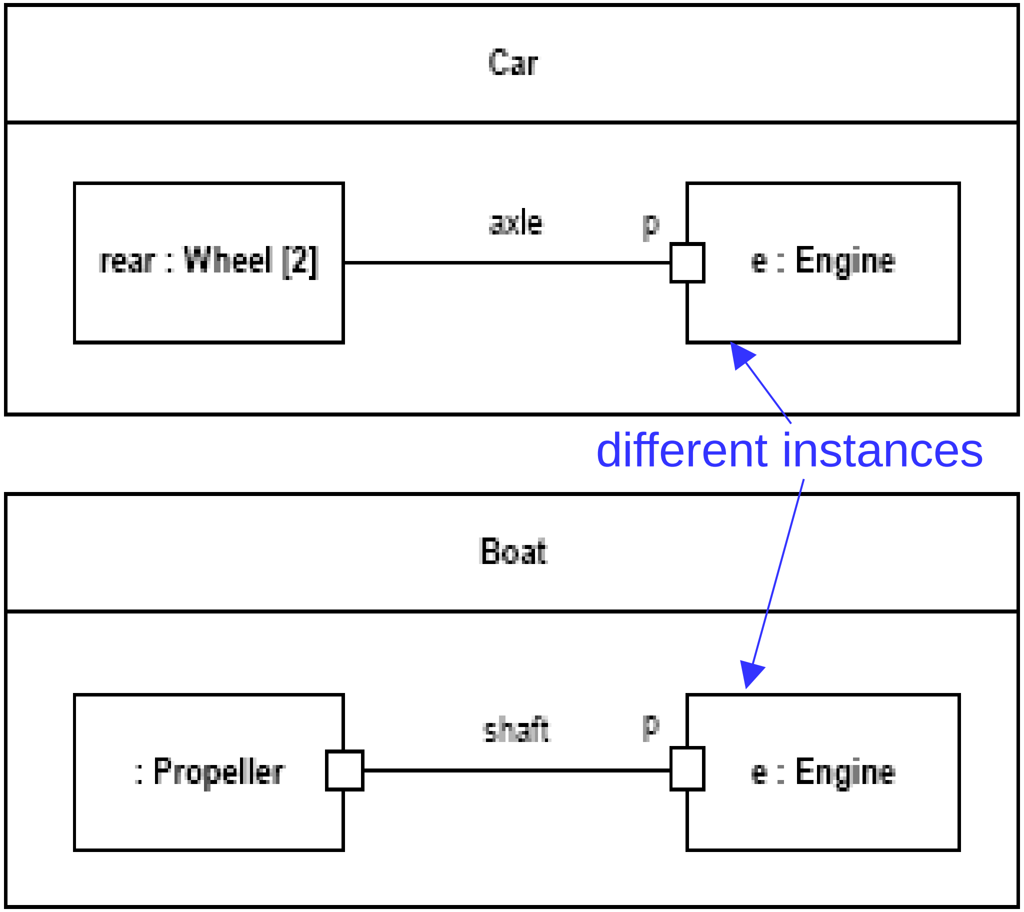
  => Lives and dies as part of the lifetime of an object of the containing class.
- Shown as an unadorned **rectangle**.
- **A part is a property**
- Parts may have multiplicities in the format [n..m]



From Figure 9.20

C) Part (2/2)

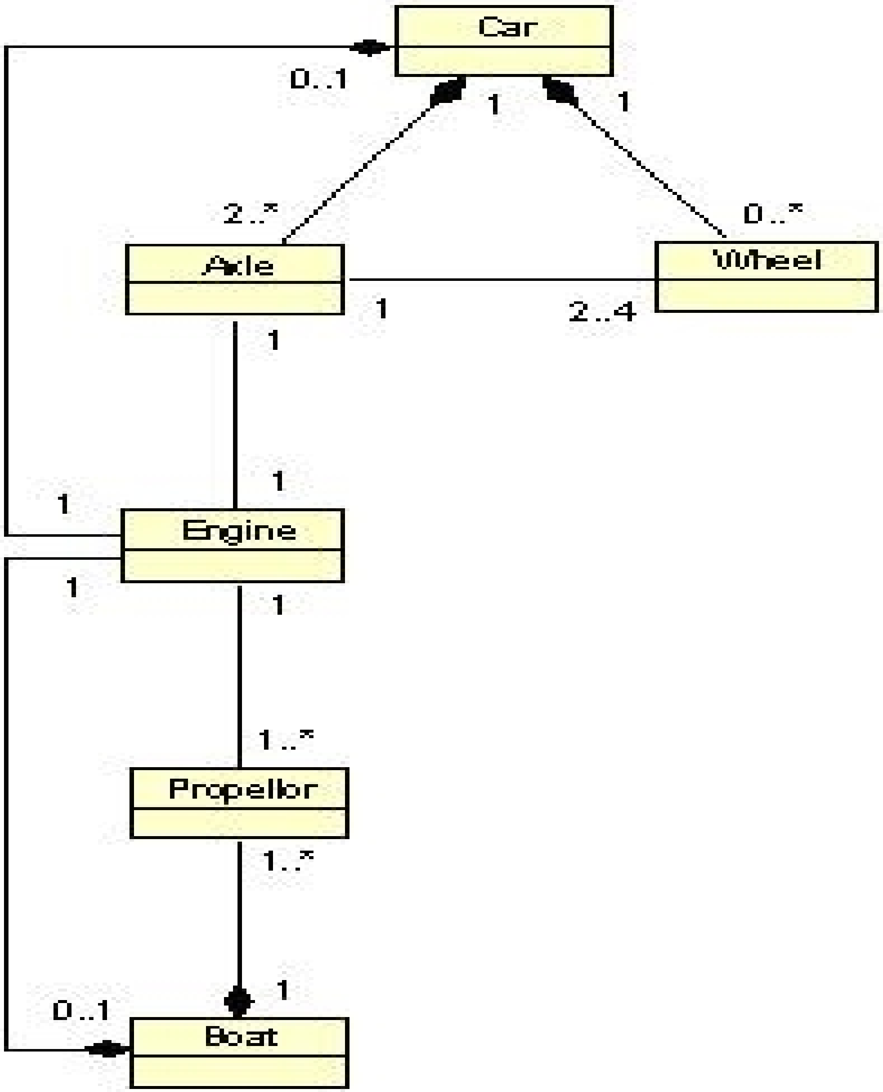
- The two parts with the same name *e:Engine* are different instances.



From Figure 9.19

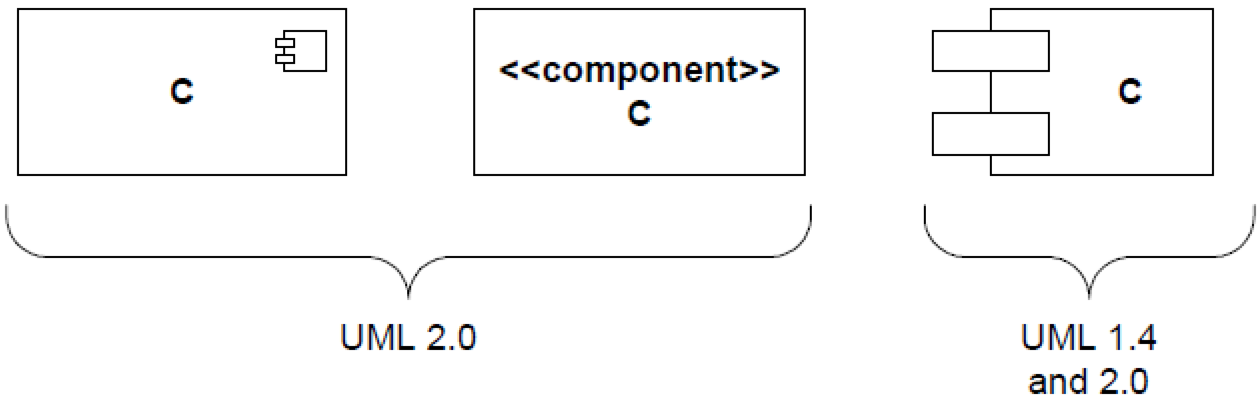
C) Part – Motivation for Part and Connector

- The composition association can also be used to describe composition.
- However, this model does not express completely that the same instance of Engine cannot be connected to an Axle in a Car and to Propeller in a boat at the same time.



1.2 Component (1/3)

- In UML 1.4, *component* had a strong implementation bias as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces" [OMG 01, p. 2-31].
- In UML 2.0, a *component* is "a **modular part** of a system that encapsulates its contents and whose manifestation is replaceable within its environment" [OMG 11, p. 149].



Three Ways to Represent Components

1.2 Component (2/3)

- UML 2.0에서는 개발주기과정에 걸쳐 더 의미가 있는 개념이 되도록 일반화 되었음
- UML 2.0 에서 component는 class의 일종
  - component type은 class
  - component instance는 object
  - structure classifier 개념의 도입으로 내부구조까지 기술가능
- provided interface 와 required interface를 통해 세분화 가능
- attribute, behavioral description 등을 활용하여 세부적인 정보기술 가능
- port를 사용하여 Interaction point를 기술
  - 같은 타입의 다수의 port 가능
  - The required and provided interfaces may be organized through ports.

1.2 Component (3/3)

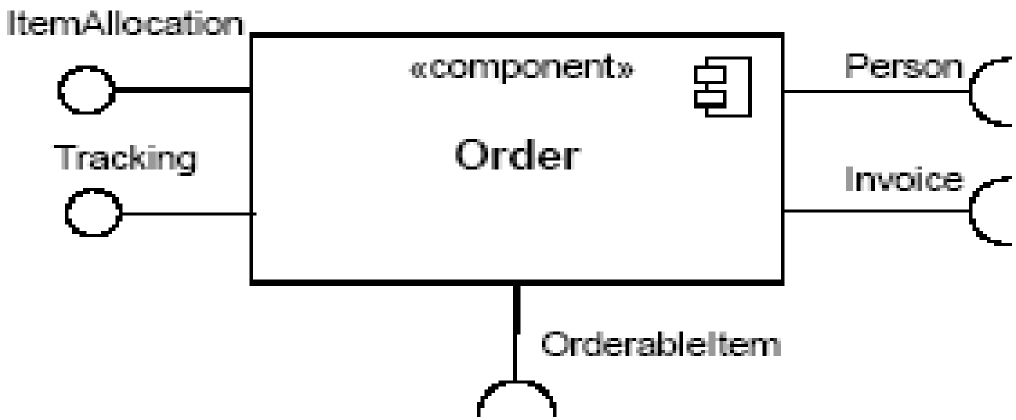


Figure 8.6 - A Component with two provided and three required interfaces

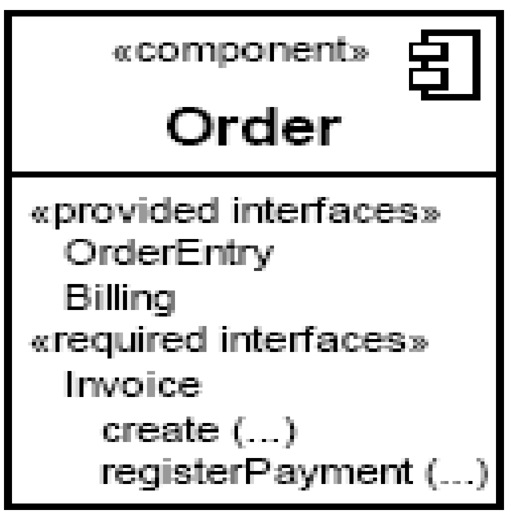


Figure 8.7 - Black box notation showing a listing of the properties of a component

1.3 Connector (1/4)

- "A connector specifies a link that enables communication between two or more instances."
- "This link may be an instance of an association, or may represent the possibility of the instances being able to communicate ... "
- In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify **links between instances playing the connected parts** only.

1.3 Connector (2/4)

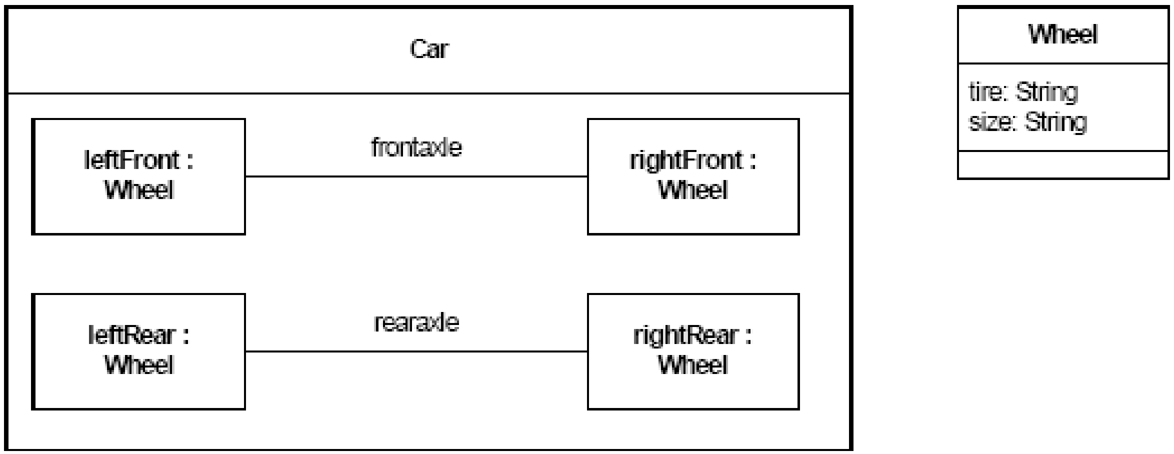


Figure 9.24 - Connectors and parts in a structure diagram

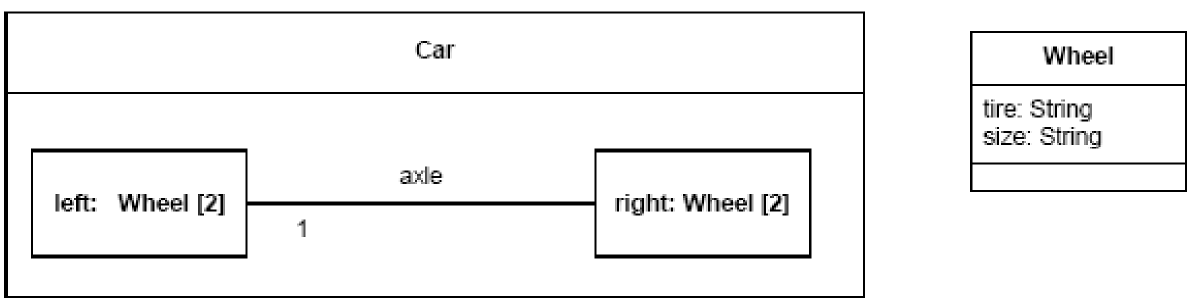


Figure 9.25 - Connectors and parts in a structure diagram using multiplicities

1.3 Connector (3/4)

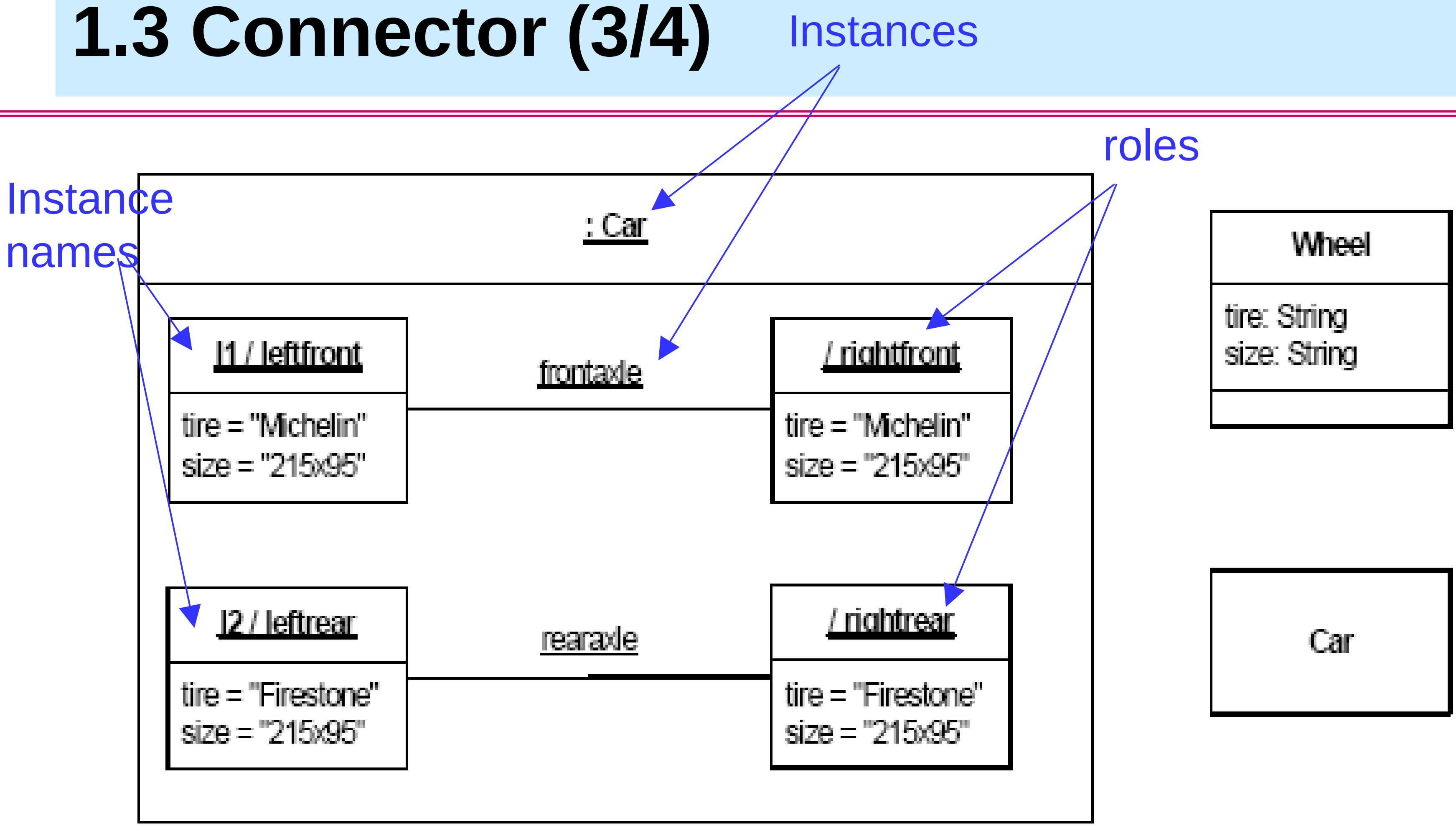


Figure 9.26 - A instance of the Car class

1.3 Connector (4/4)

- Two kinds:

  - Delegation Connectors: must be used between the same kinds of ports or interfaces (provided or required)

  - Assembly Connectors: can be defined only between a provided interface and a required interface that are compatible.

1.3 Connector - Assembly Connector (1/3)

- A connector between two components that
  - defines that one component provides the services that another component requires.
  - defined from a required interface or port to a provided interface or port.

1.3 Connector - Assembly Connector (2/3)

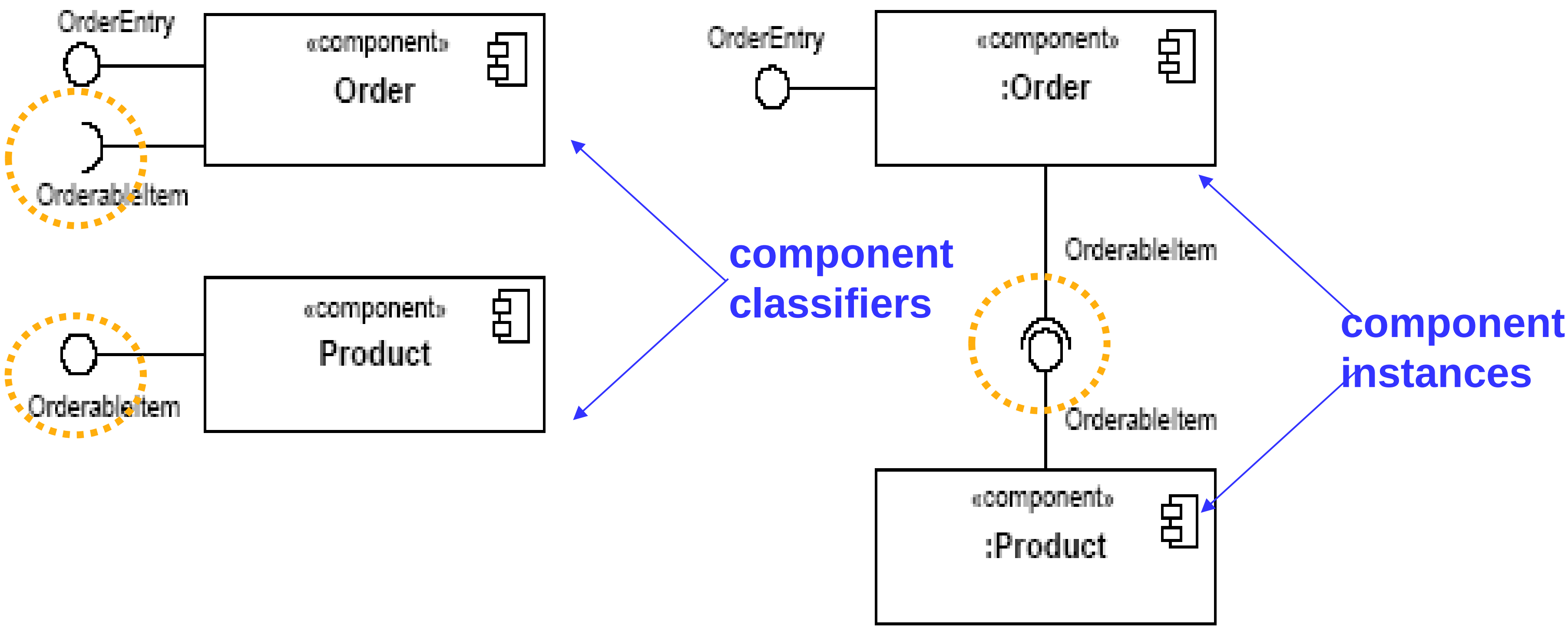


Figure 8.17 - An assembly connector maps a required interface of a component to a provided interface of another component in a certain context (definition of components, e.g., in a library on the left, an assembly of those components on the right).

1.3 Connector - Assembly Connector (3/3)

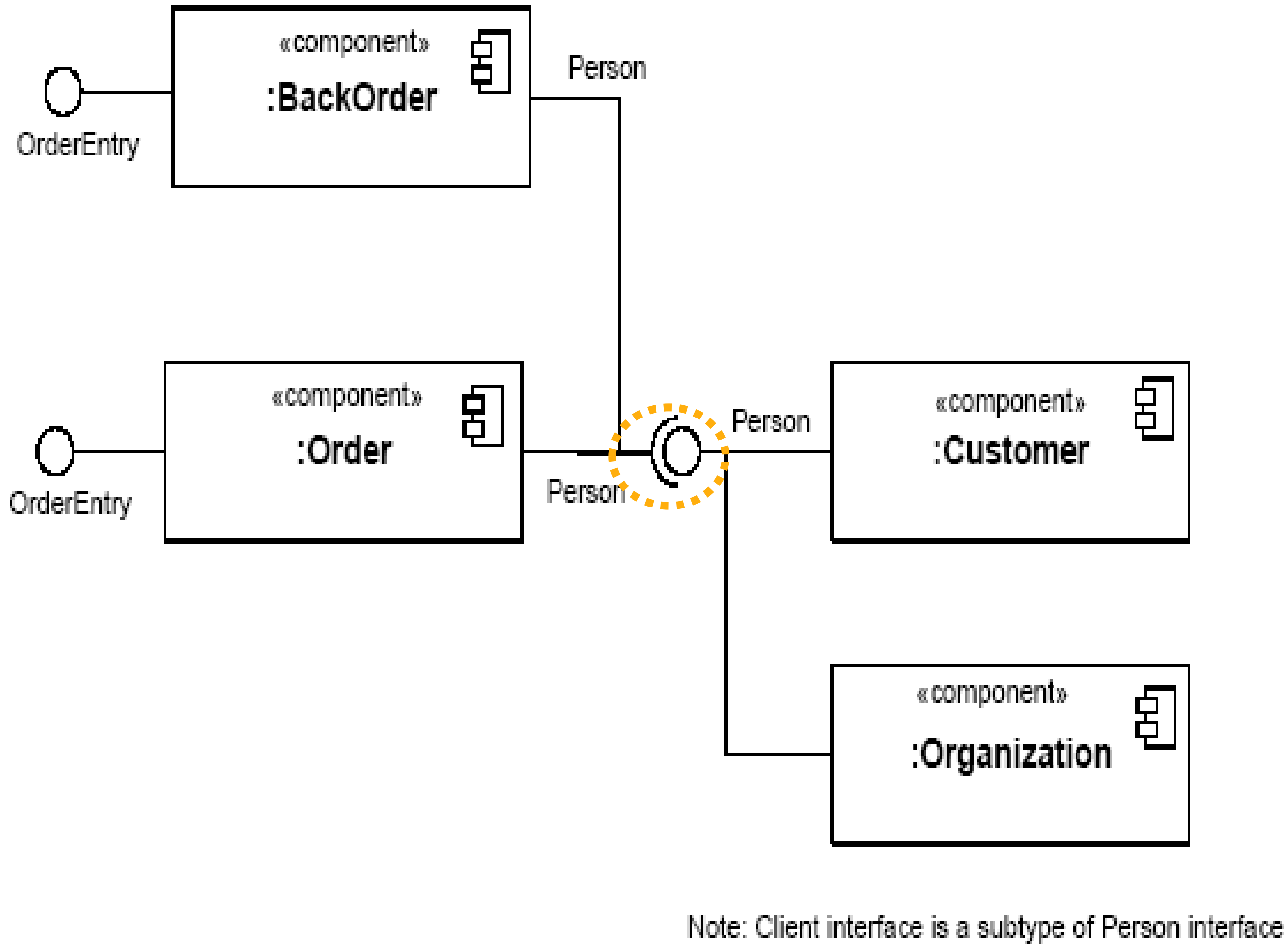


Figure 8.18 - As a notation abstraction, multiple wiring relationships can be visually grouped together in a component assembly.

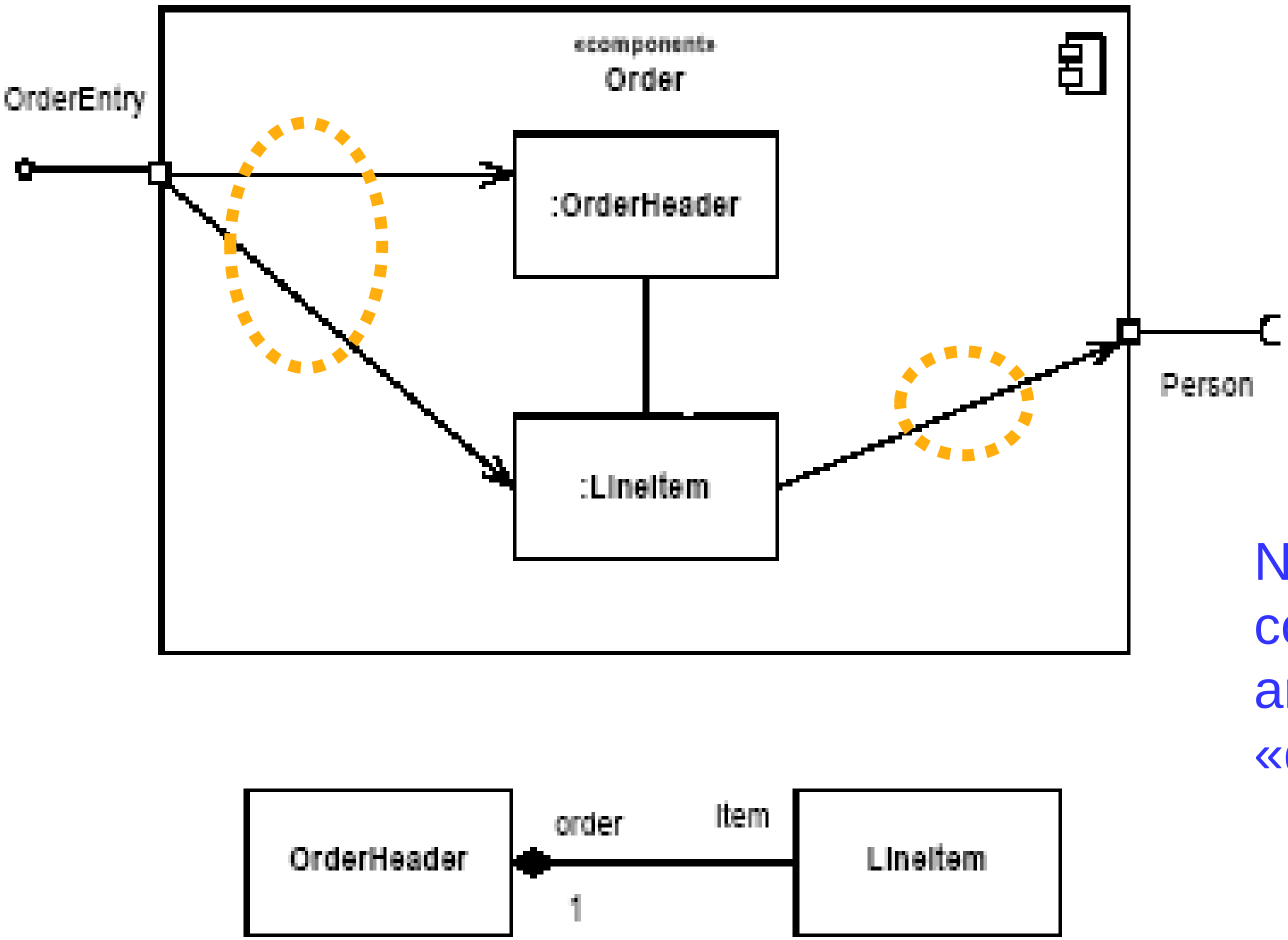
1.3 Connector - Delegation Connector (1/2)

- Define the internal workings of a component's external ports and interfaces.

- Connects an external contract of a component as shown by its ports to the internal realization of the behavior of the component's part.

- 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.

1.3 Connector - Delegation Connector (2/2)



Notation: A delegation connector is shown as an arrow with a «delegate» stereotype.

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

1.4 Interface

- Interfaces from UML 1.4 are provided interfaces
- UML 2.0 extends the interface concept to explicitly include *provided* and *required* interfaces.

Three Different Ways of Representing Interfaces

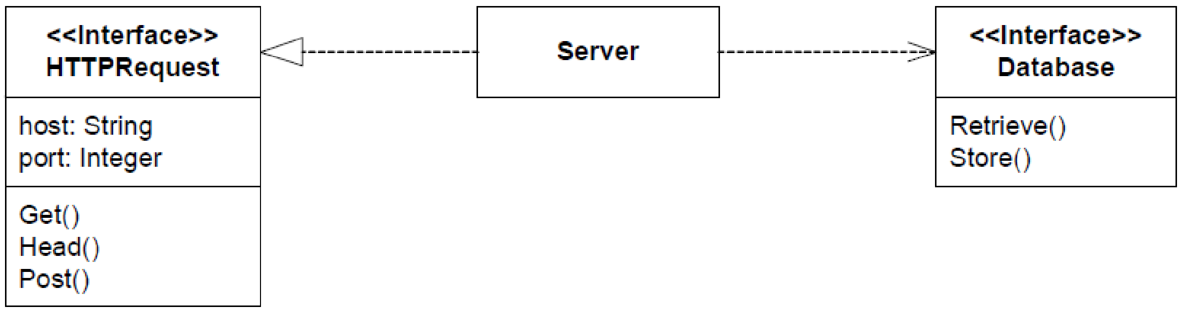


Figure 1: Interfaces as Stereotyped Classifiers

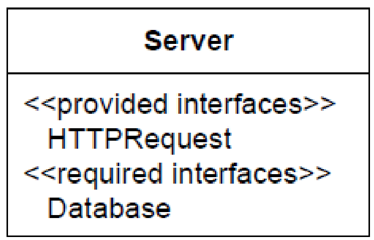


Figure 2: Interfaces in a Classifier Compartment

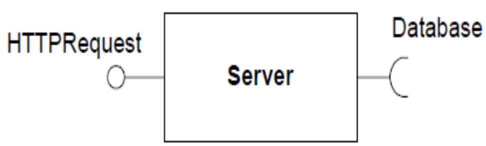


Figure 3: Interfaces Using Ball-and-Socket Notation

1.5 Port

- A typed element that represents an externally visible part of a containing classifier instance
- Specifies a distinct interaction point:
  – between a classifier and its environment
  – between the (behavior of the) classifier and its internal parts
- May specify the services provided or required by a classifier
  – how a service offered from an internal part is published across the boundary of its enclosing class
  – their behavior and sequencing constraints using state machines

1.5 Port- Notation

Powertrain interface is shown as the type of the port p.

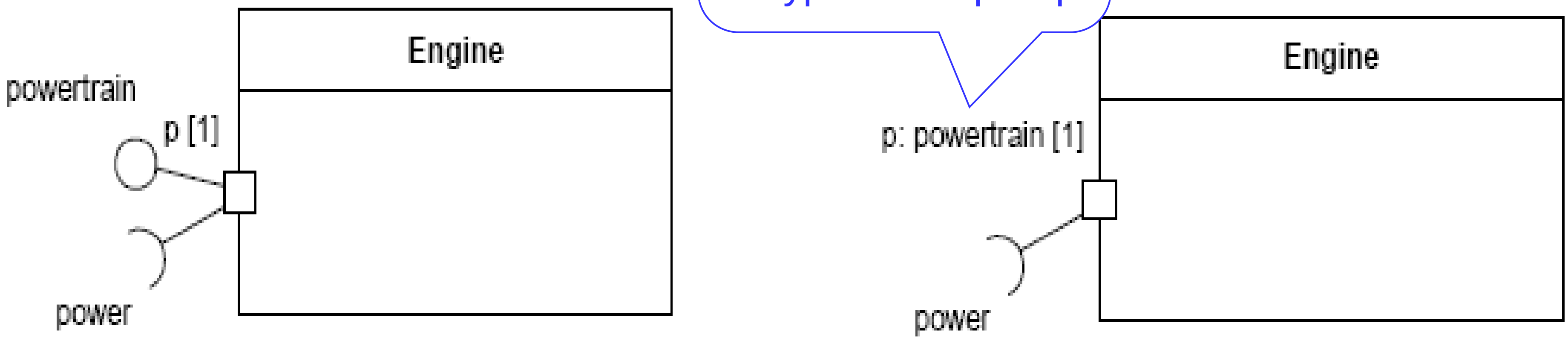


Figure 9.16 - Port notation

indicates the behavior of the containing classifier

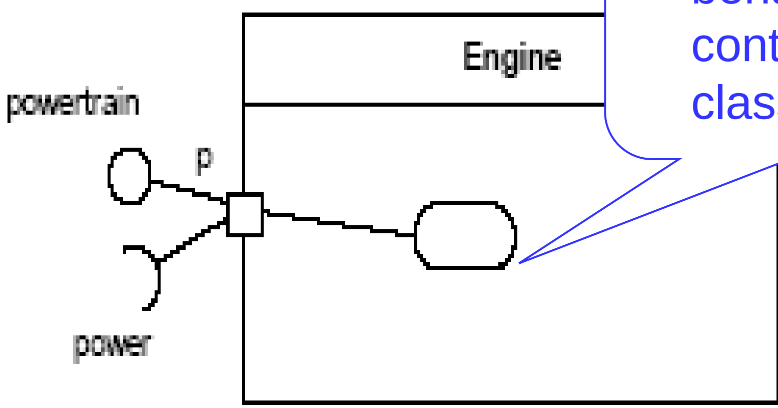


Figure 9.17 - Behavior port notation

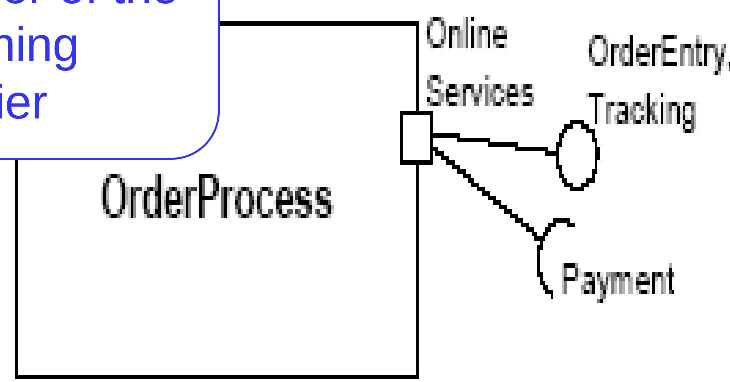
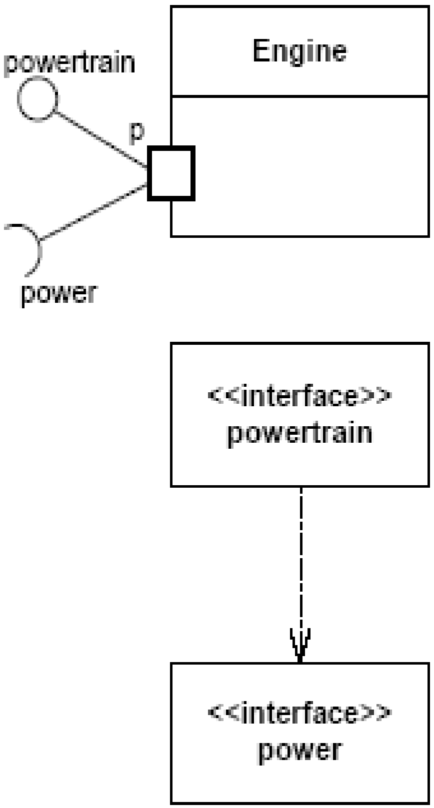


Figure 9.18 - Port notation showing multiple provided interfaces

1.5 Port- Examples

OK not to attach port



The above implies this but vice versa.

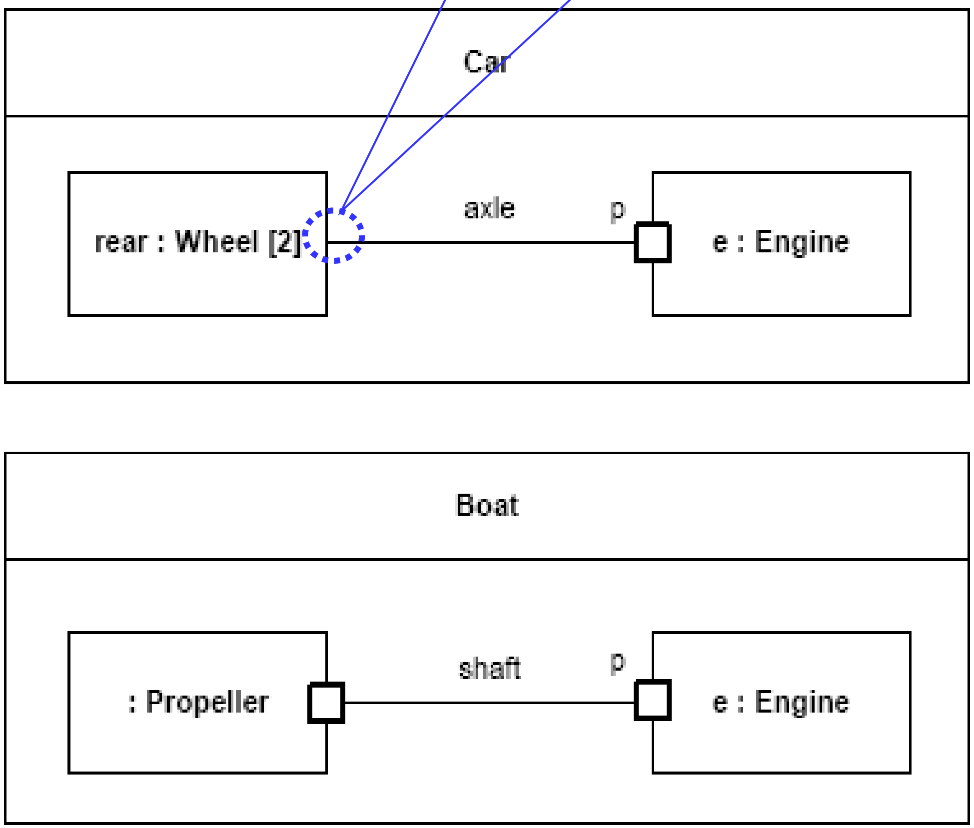


Figure 9.19 - Port examples

1.5 Port

- Port and Interface

	Port	Interface
Describes interaction	Yes	Yes
Is a distinct interaction point	Yes	No
Instantiable	Yes	No
Can have behavior	Yes	**Yes**

Example A predetermined sequence of operation invocations

- Ports are not required to build systems
  - using ports will usually introduce some runtime overhead and require additional memory

1.6 Composite Structure Diagram

- The term “structure” refers to “
  a composition of interconnected elements,
  representing run-time instances
  collaborating over communications links
  to achieve some common objectives.”

  - [UML Superstructure 9.1, p169]

- Shows the *internal structure* (including parts and connectors) of a structured classifier or *collaboration*.

  Composite structure “=” Internal structure

  – Shows the configuration and relationship of parts that together perform the behavior of the containing classifier.

  – Shows the interaction points to other parts of the system.

Ksw: Various notions of structure

- What are the meanings of “static” and “dynamic”?
  - Static: irrespective of time? or exists in time but does not involve change through time?
    1. code level?
    2. runtime but at the beginning?
    3. runtime but at one moment of execution?
  - Dynamic: involves time duration?
    4. runtime and involve change through time

- Static structure
  - Structure among modules?
  - Structure among computing entities?
    - At the beginning of execution (after initialization)
    - At a moment in a middle of execution

- Dynamic structure
  - Behavior of a system

After all, which are really important !!

- Runtime Architecture: Can it be static? Or is it dynamic?

1.6 Composite Structure Diagram

(1) **Internal Structure** : mechanisms for specifying structures of interconnected elements

(2) **Collaboration**: Objects in a system typically cooperate with each other to produce the behavior of a system. The behavior is the functionality that the system is required to implement.

(3) **Connector:** Specifies a link that enables communication between two or more instances

(4) **Port**: mechanisms for isolating a classifier from its environment.

(5) **Structured Class**: supports the representation of classes that may have ports as well as internal structure.

already discussed

1.6 Composite Structure Diagram – Graphic Elements

Part

Port

Collaboration

CollaborationUse

Connector

(1) Internal Structure

Does not mean instances will actually be exhibited (See Fig 9.20)

- *Internal structure* refers to the structures of interconnected elements that are created within an instance of a containing classifier and represents a decomposition of that classifier.

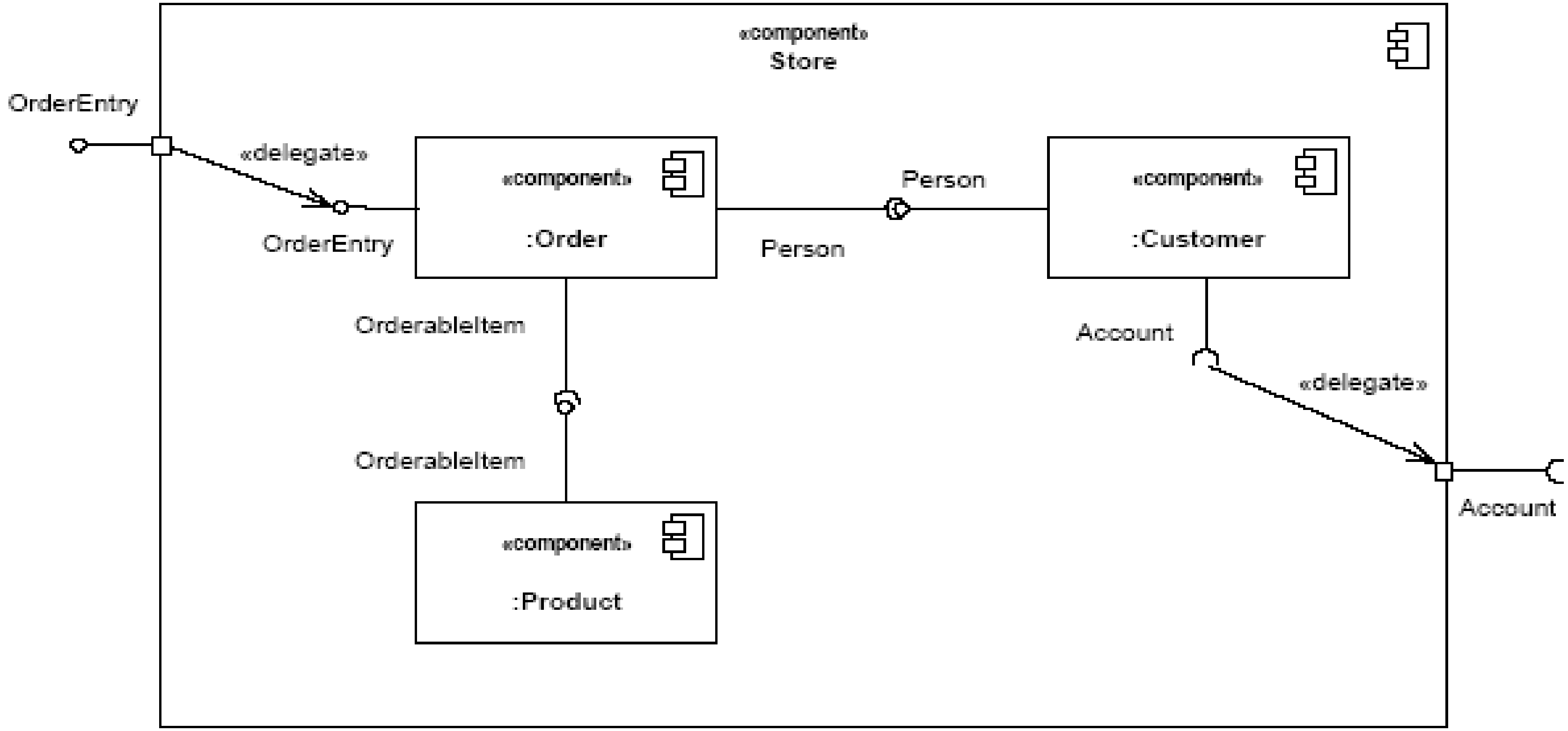


Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.

(2) Collaboration (1/2)

- Describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality.
- Its primary purpose is to explain how a system works

  => Typically only incorporates those aspects of reality that are relevant to the explanation suppressing the details such as the identity or precise class of the actual participating instances

- Often implements a pattern.

(2) Collaboration (2/2)

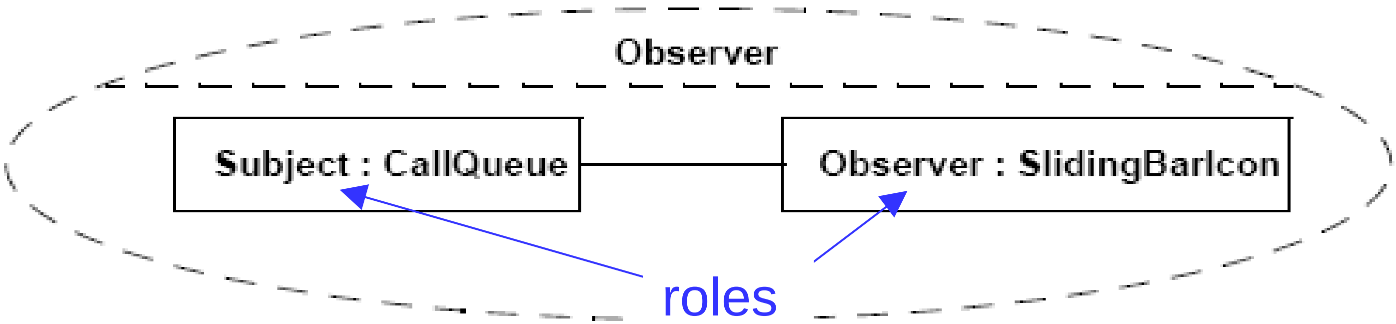


Figure 9.11 - The internal structure of the Observer collaboration shown inside the collaboration icon (a connection is shown between the Subject and the Observer role).

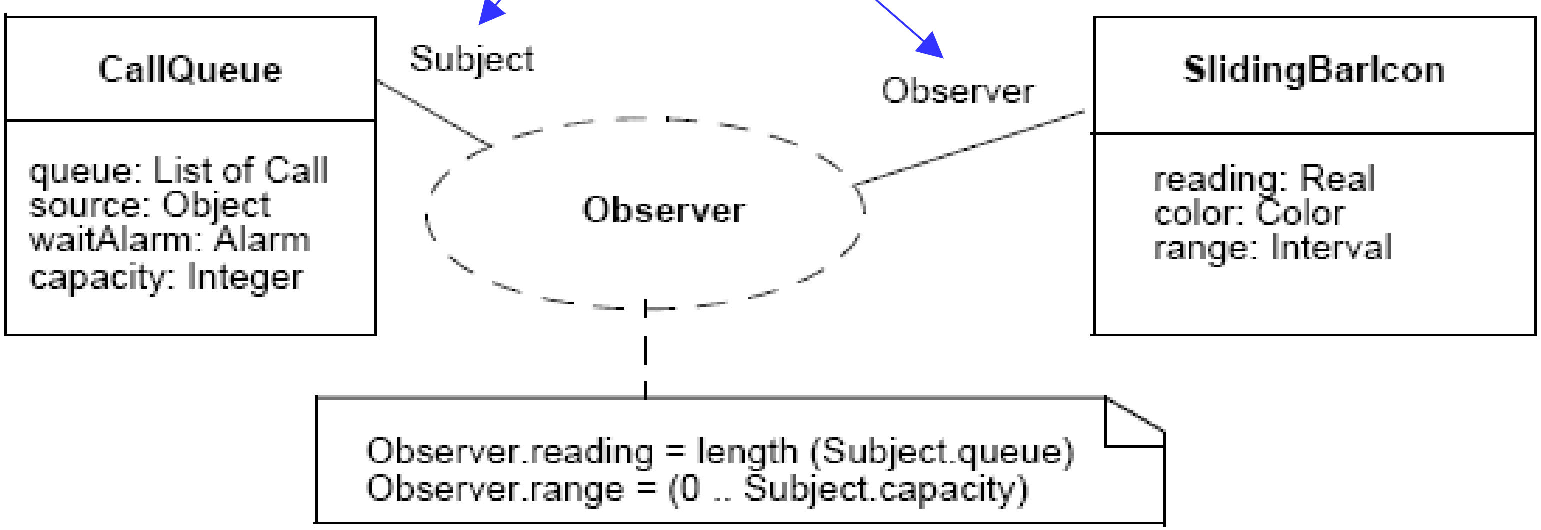


Figure 9.12 - In the Observer collaboration two roles, a Subject and an Observer, collaborate to produce the desired behavior. Any instance playing the Subject role must possess the properties specified by CallQueue, and similarly for the Observer role.

(2) Collaboration - CollaborationUse

- Represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration

- By binding specific **entities** to the roles of the collaboration, shows how the pattern described by a collaboration is applied in a given context.
  - Depending on the context, these entities could be
    - structural features of a classifier
    - instance specifications, or even
    - roles in some containing collaboration.

(2) Collaboration - CollaborationUse

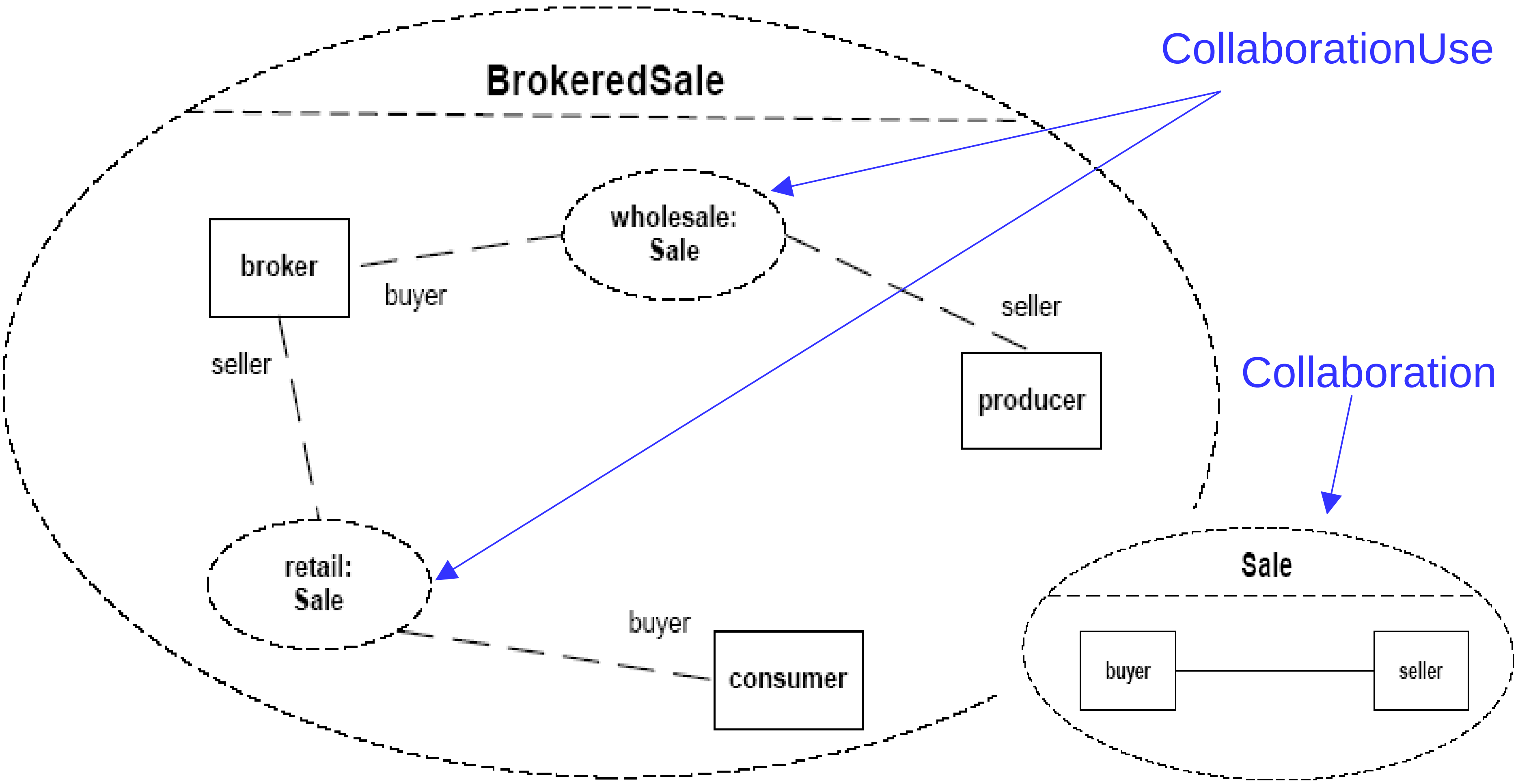


Figure 9.14 - The BrokeredSale collaboration

Figure 9.13 - The Sale collaboration

(2) Collaboration - CollaborationUse

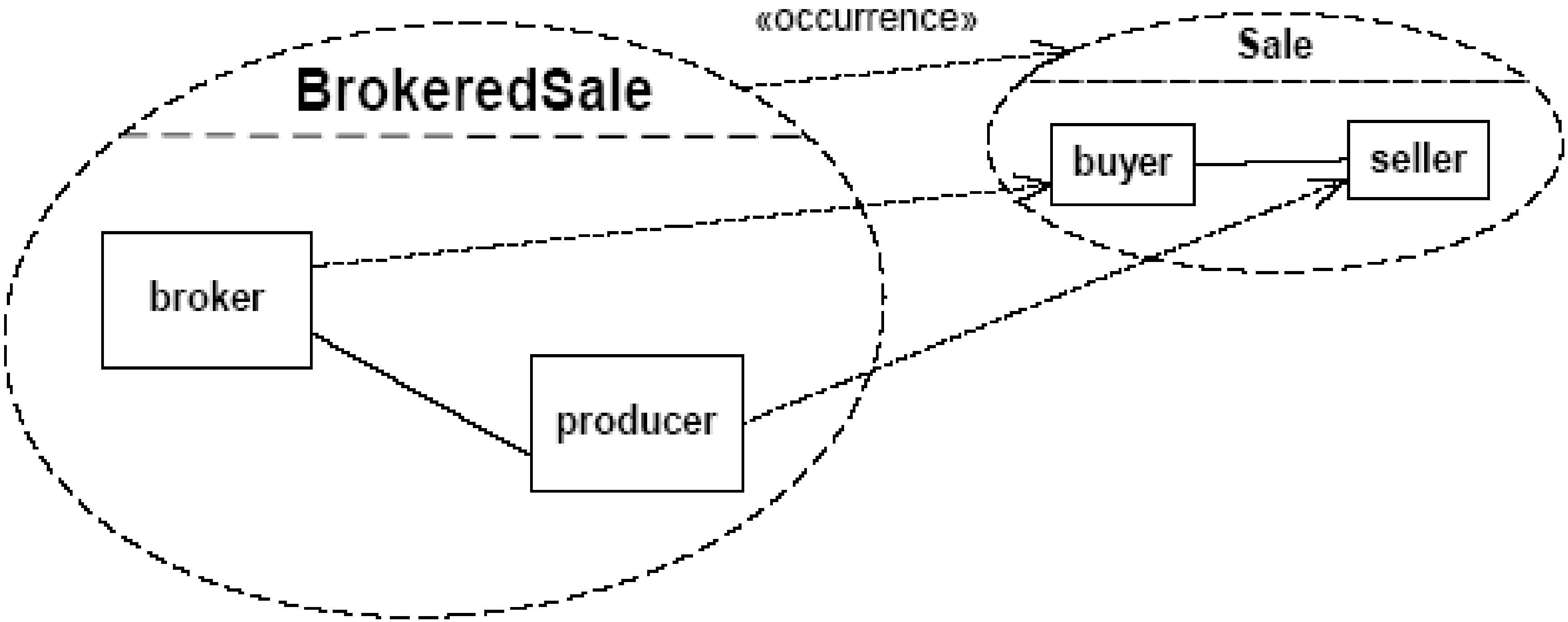
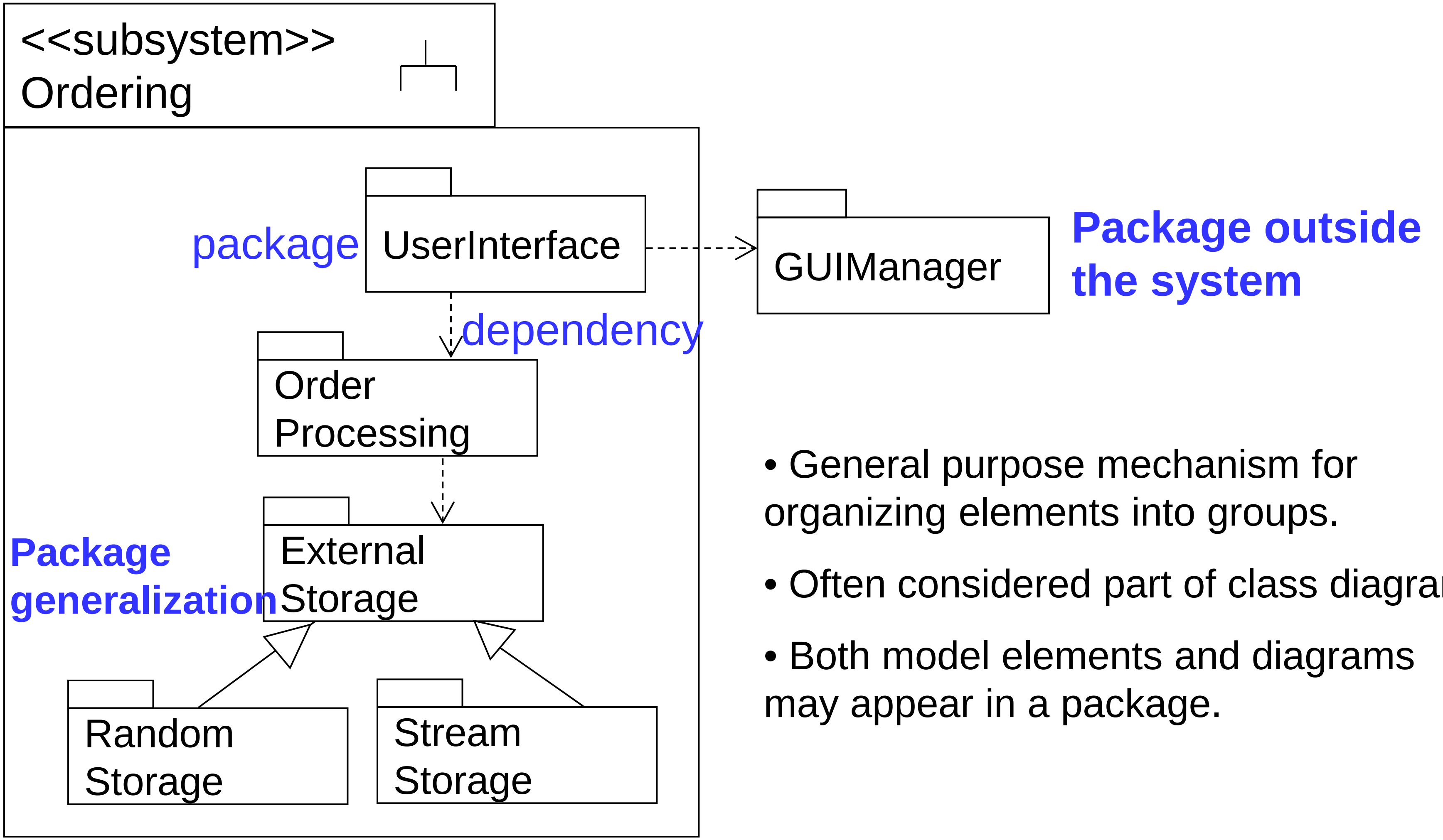


Figure 9.15 - A subset of the BrokeredSale collaboration

1.7 Package (1/2)

- Defines a namespace
- Used to represent “subsystem”
- Contain design elements only: classes, class diagrams, use cases etc.
- Subdivide models to permit teams of developers to manipulate and work effectively together

1.7 Package (2/2)



- General purpose mechanism for organizing elements into groups.
- Often considered part of class diagram
- Both model elements and diagrams may appear in a package.

2. Component Diagram

2.1 Component

2.2 External View

2.3 Internal View

2.1 Component

- "... represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment."
- Can be replaced at design time or run-time by a component that offers equivalent functionality.
- Specifies a formal contract of the services that it provides and that it requires from other components or services.
- A self contained unit that encapsulates the state and behavior of a number of classifiers.
- May implement a provided interface directly, or, its realizing classifiers may do so.
- External view vs. Internal view
- Subsystems :
  - A kind of component.
  - Usually "larger than" components and may contain other components.

2.2 External View

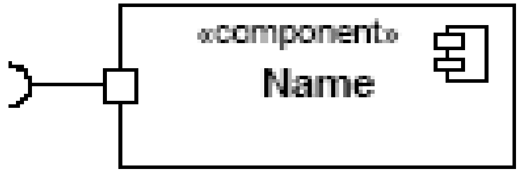
- "Black-box" view of component
  - By means of publicly visible properties and operations.

- A behavior such as a protocol state machine may be attached to an interface, port, and to the component itself
  - Defined 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., use case, activity, or interaction specifications).

2.2 External View – Graphic Nodes

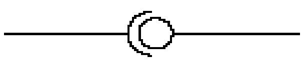
Component

Component has *required* Port



Component has *provided* Port

Assembly connector



Component *implements* Interface

Component *has* complex Port

Component *uses* Interface

2.2 External View

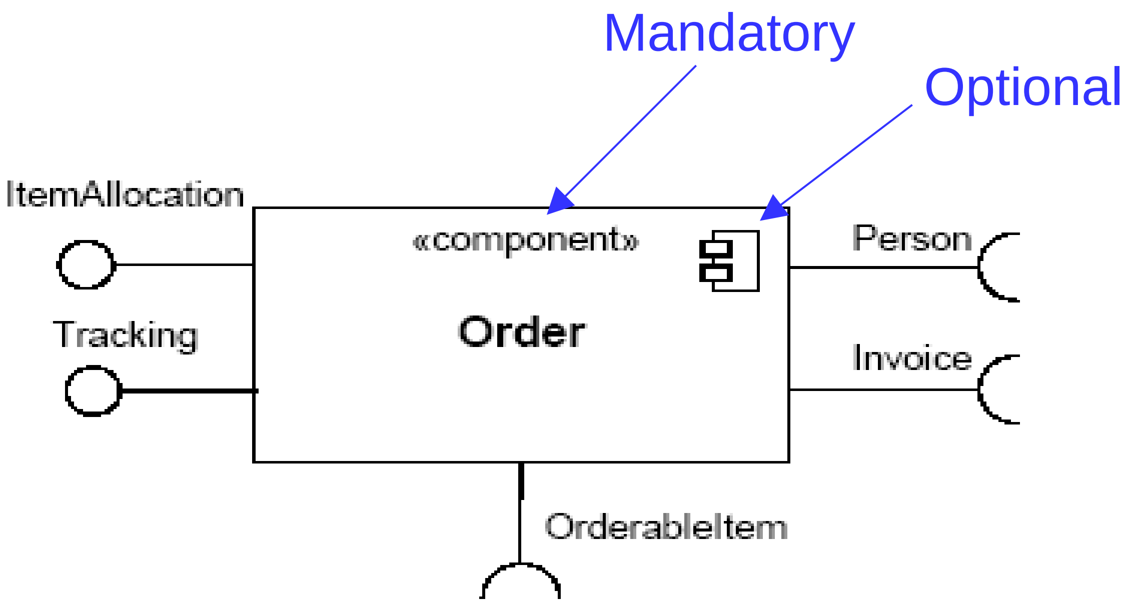


Figure 8.6 - A Component with two provided and three required interfaces

- A component is shown as a Classifier rectangle with the keyword «component».

2.2 External View

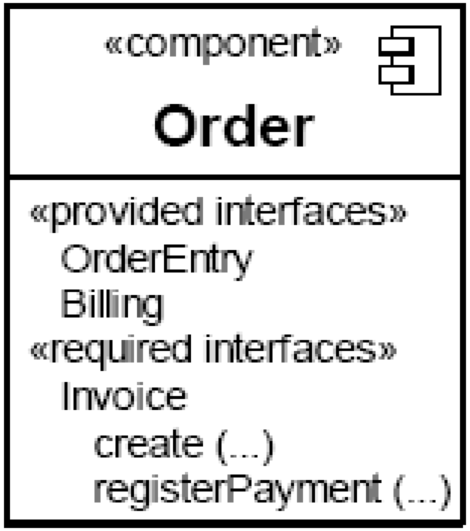


Figure 8.7 - Black box notation showing a listing of the properties of a component

- Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box

2.2 External View

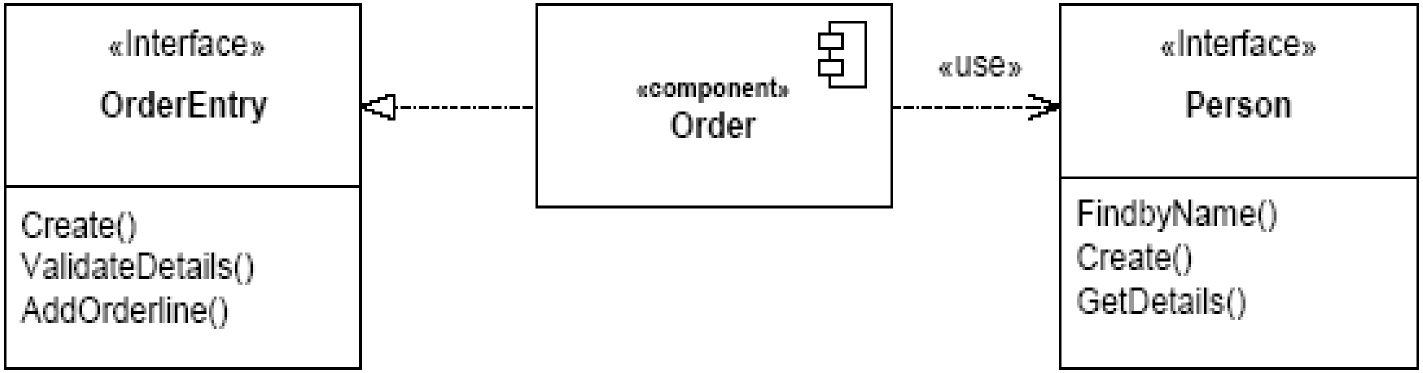
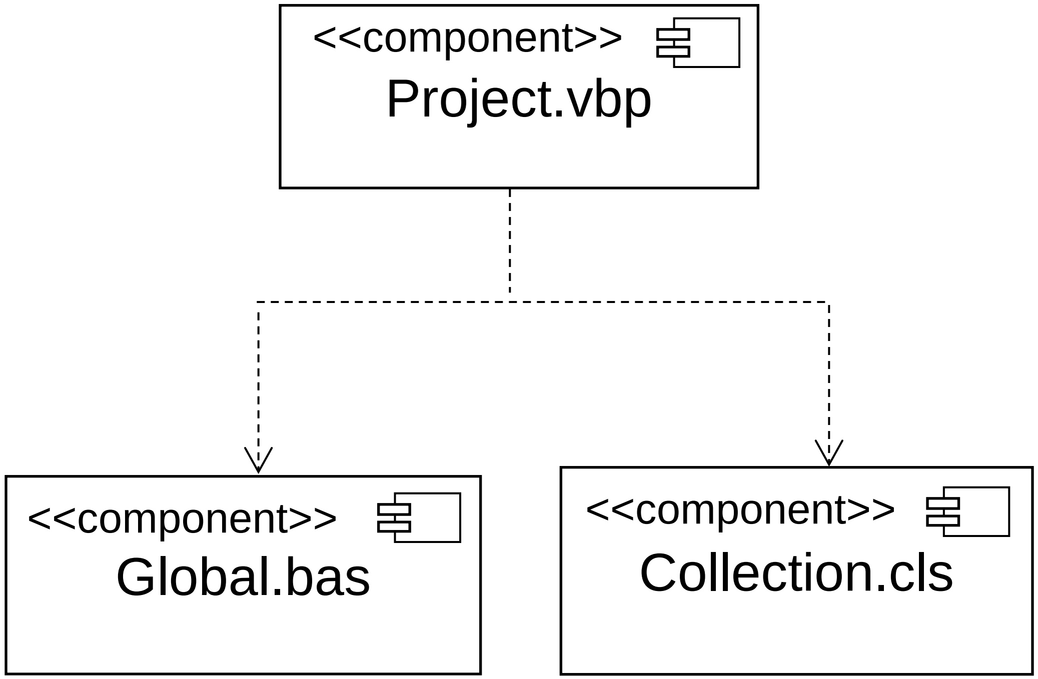


Figure 8.8 - Explicit representation of the provided and required interfaces, allowing interface details such as operation to be displayed (when desired).

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

2.2 External View

- Project file depends upon both the Global.bas and Collection.cls files.



However, later we will see that <<includes>> dependency represents internal view.

2.3 Internal view

- "White-box" view of component
- Shows how the external behavior is realized internally.
  - By means of its private properties and realizing classifiers.
- 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).
- More detailed behavior specifications such as interactions and activities may be used to detail the mapping from external to internal behavior.
- Standard stereotypes for component:

  «subsystem», «specification», «realization»

2.3 Internal View (1/3)

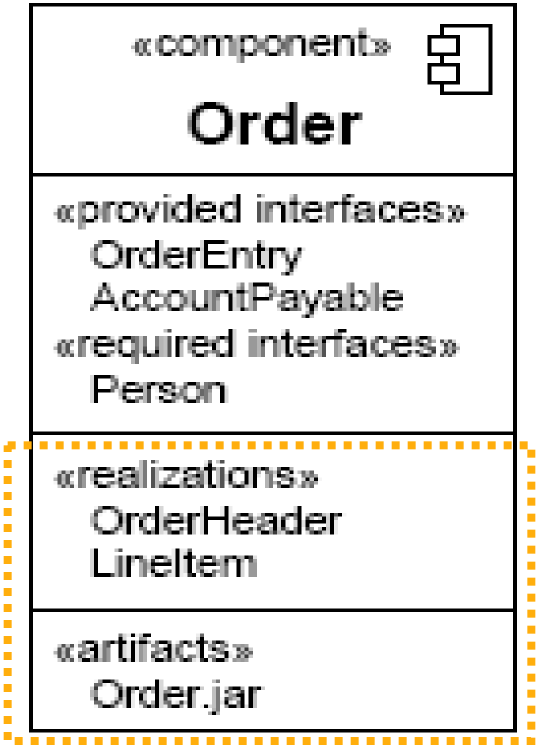
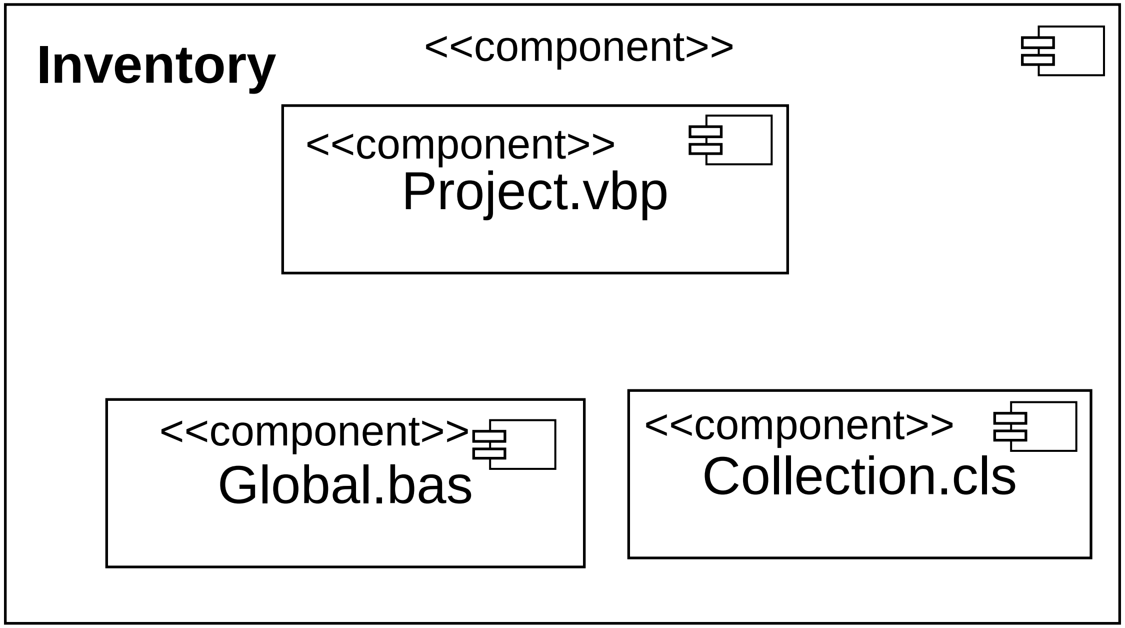


Figure 8.9 - A white-box representation of a component

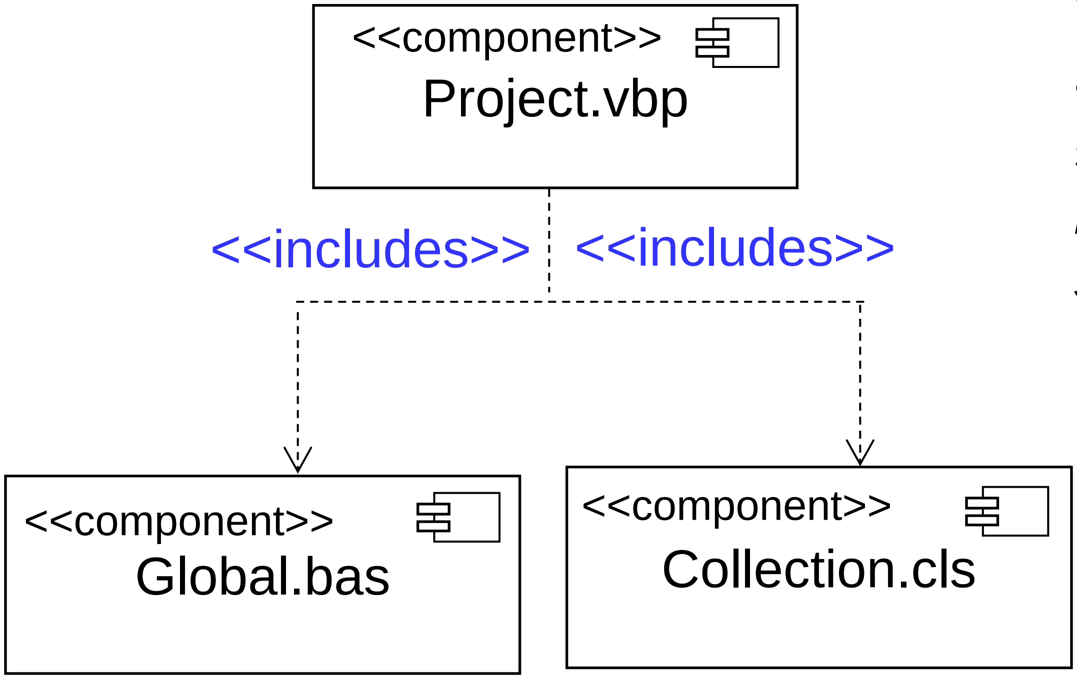
- Realizing classifiers are listed in an additional compartment

2.3 Internal View (2/3)

- Contained Components



2.3 Internal View (3/3)



- Stereotype can be used to explain further the dependency relation.
- Commonly used stereotypes are *includes*, *imports*, *contains*, *has*, *supports*.

2.3 Internal View (*)

- The internal classifiers that realize the behavior of a component may be displayed by means of general dependencies.

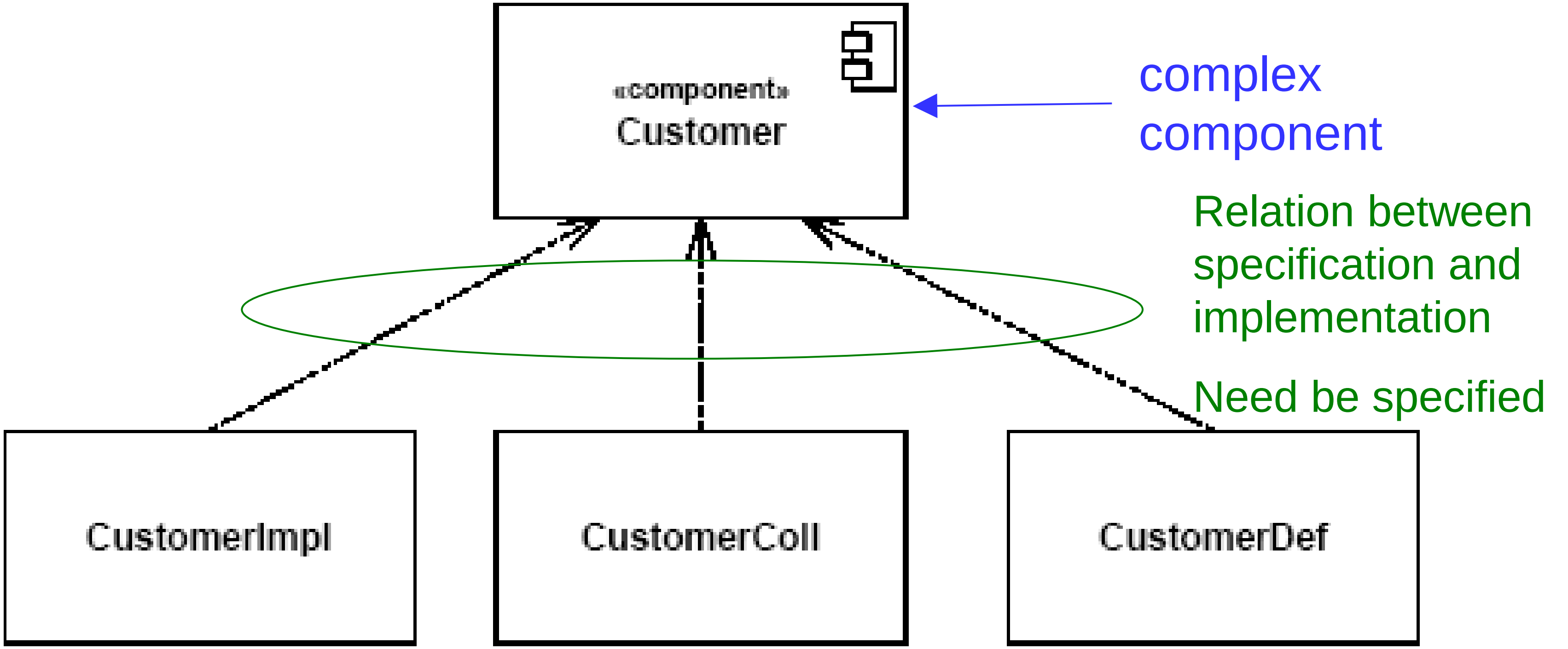


Figure 8.10 - A representation of the realization of a complex component

2.3 Internal View (*)

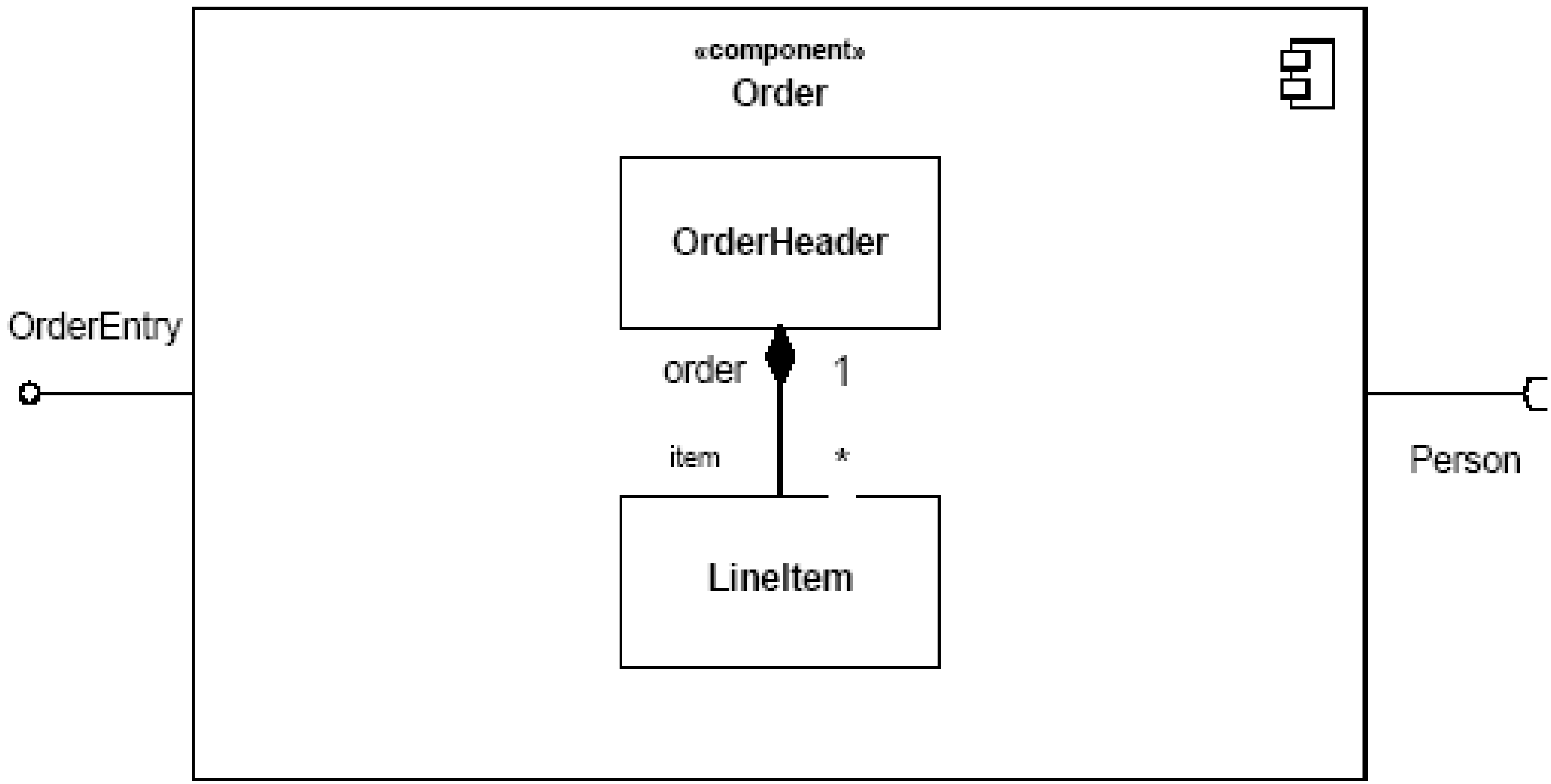


Figure 8.11 - An alternative nested representation of a complex component

2.3 Internal View (*)

- 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.

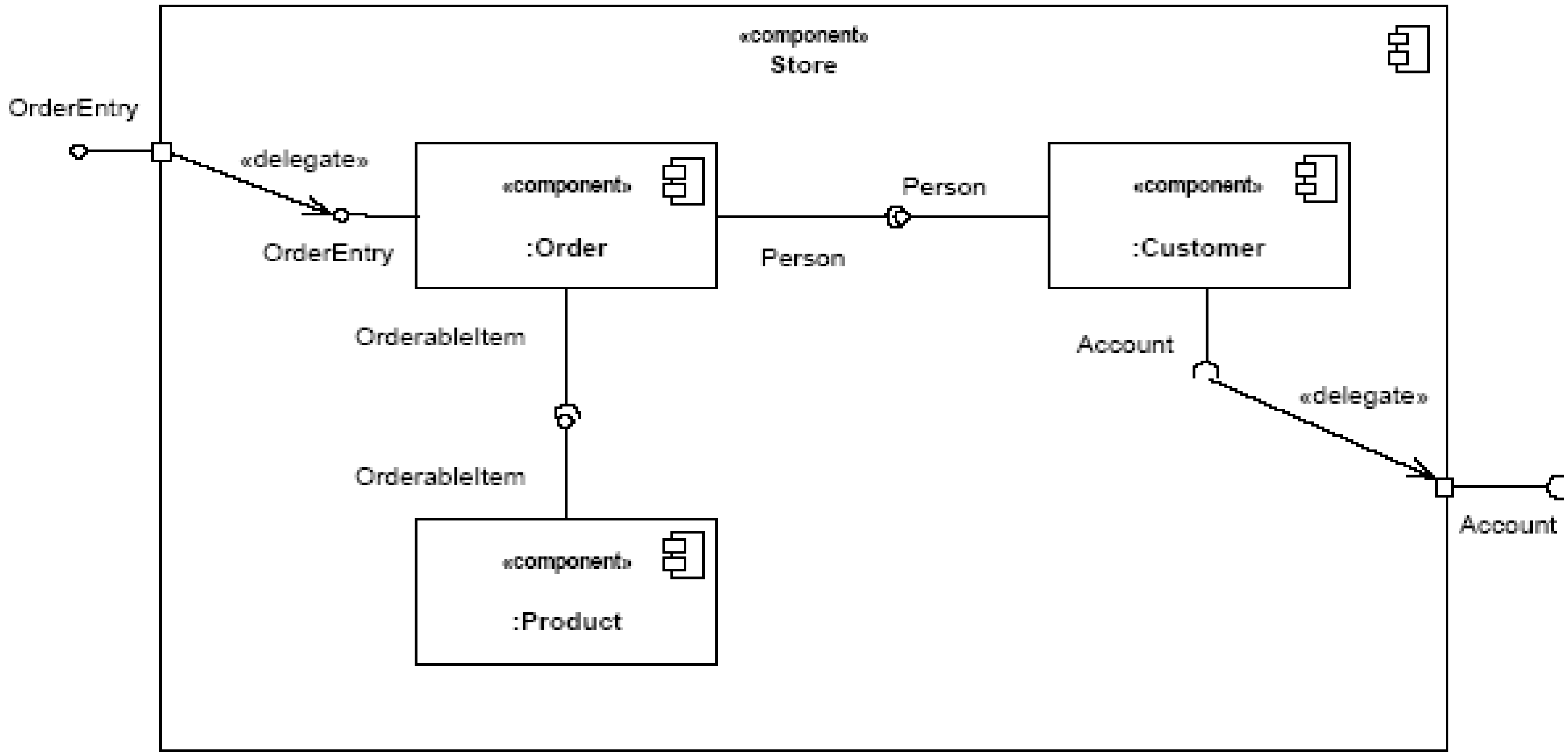


Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.

2.3 Internal View (*)

- Artifacts that implement components can be connected to them by physical containment or by an «implement» relationship.

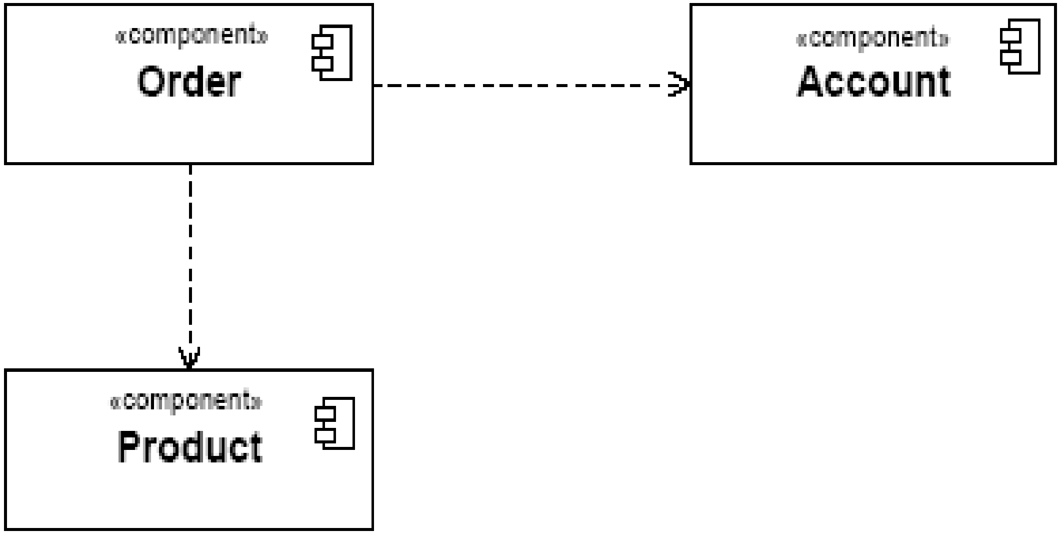
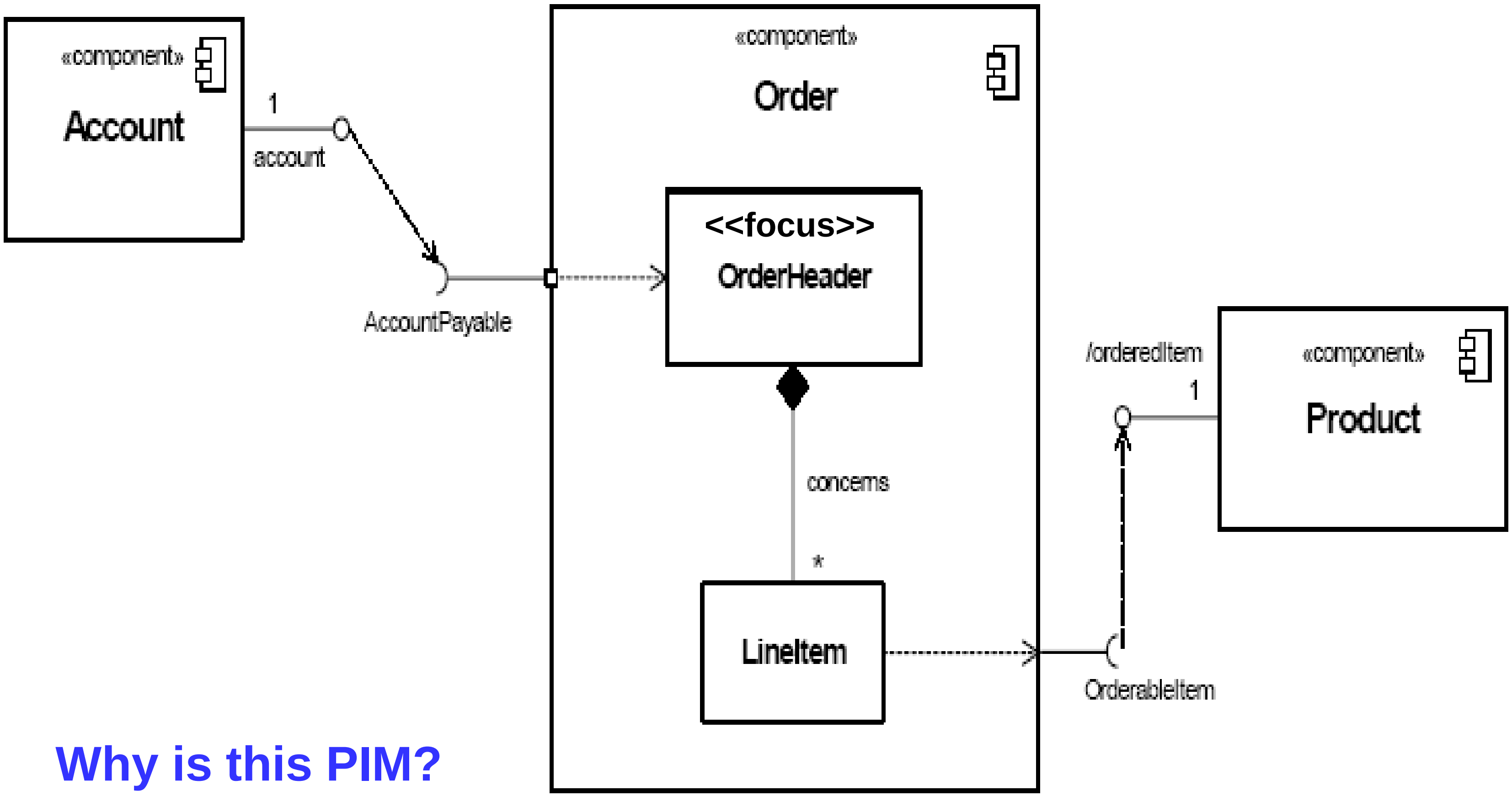


Figure 8.13 - Example of an overview diagram showing components and their general dependencies

2.3 Internal View (*)



Why is this PIM?

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.

2.3 Internal View (*)

On composite structure diagrams, detailed wiring can be performed at the **role level** or **instance level** by defining parts and connectors.

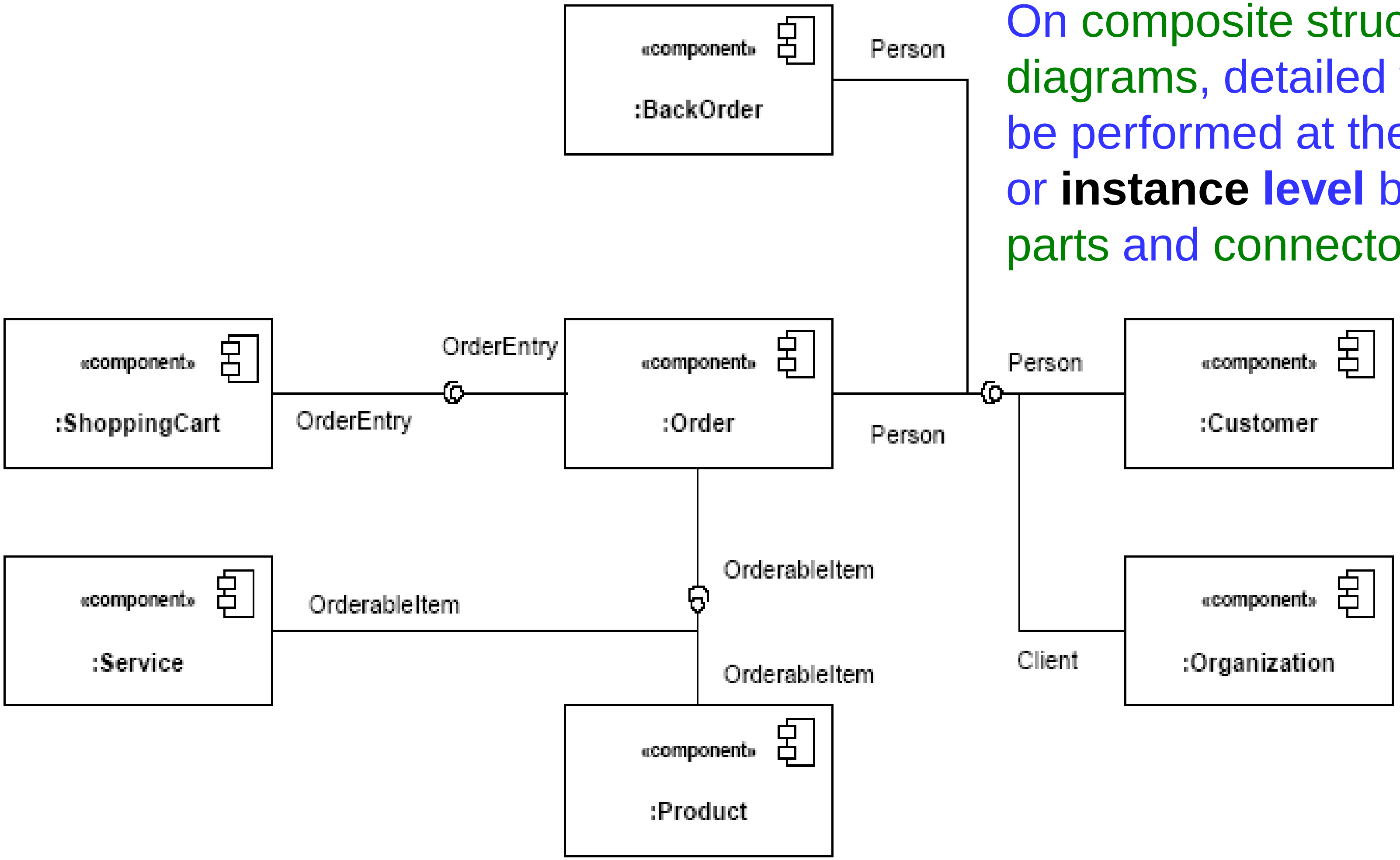


Figure 8.15 -Example of a composite structure of components, with connector wiring between provided and required interfaces of parts (Note: “Client” interface is a subtype of “Person”).

3. Deployment Diagram

3.1 Artifact (1/2)

- Specification of a physical piece of information that is
  - produced by a software development process and
  - used by deployment and operation of a system

    => Narrow sense of ‘Artifact’

Examples

- Model files
- Source files
- Scripts
- Binary executable files
- Table in a database system
- Development deliverable
- Word-processing document
- Mail message

3.1 Artifact (2/2)

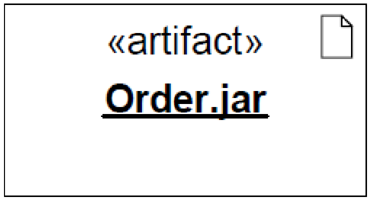
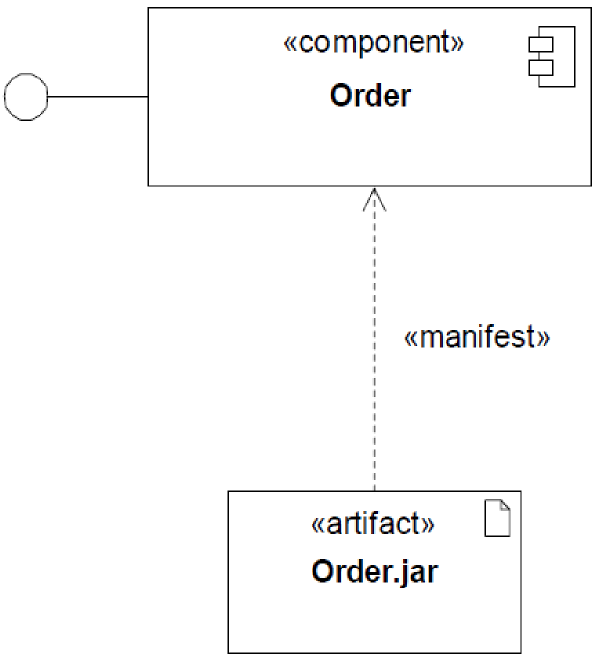


Figure 10.6 - An Artifact instance



- Presented using an ordinary class rectangle with the key-word «artifact».

  or by an icon. (ksw: make sense because it is narrow sense.)
- A manifestation is the concrete physical rendering of one or more model elements by an artifact.
- Optionally, **the underlining of the name of an artifact instance may be omitted**, as the context is assumed to be known to users.

  ☛ **Dangerous !**

Figure 10.7 - A visual representation of the manifestation relationship between artifacts and components

3.2 Deployment Diagram

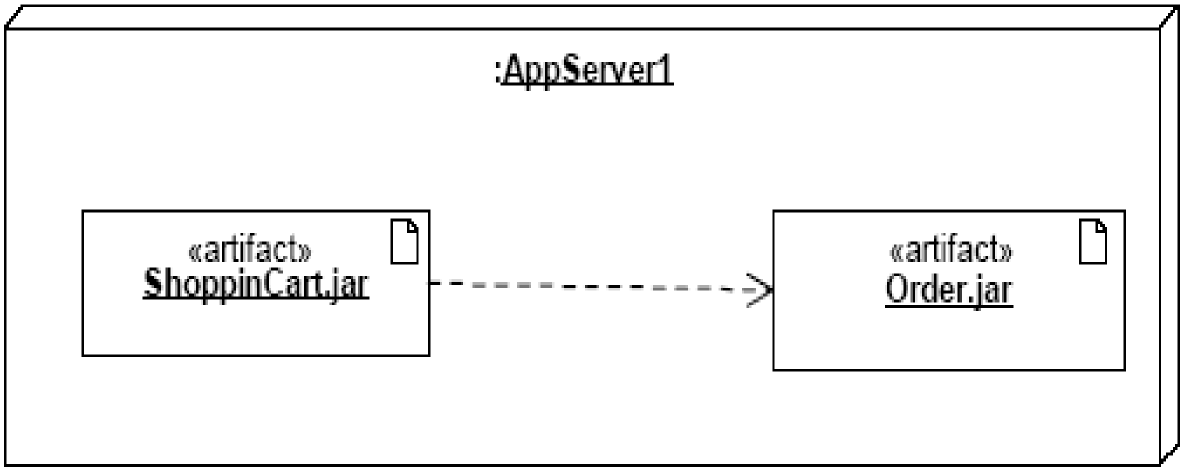


Figure 10.8 - A visual representation of the deployment location of artifacts (including a dependency between the artifacts).

- Shows the allocation of Artifacts to Nodes according to the Deployments defined between them.
- Artifacts may be contained within node instances.

3.2 Deployment Diagram - Alternative Notations

- An alternative notation is to use a dependency labeled «deploy».
- Dashed arrows with the keyword «deploy» show the capability of a node type to support a component type.

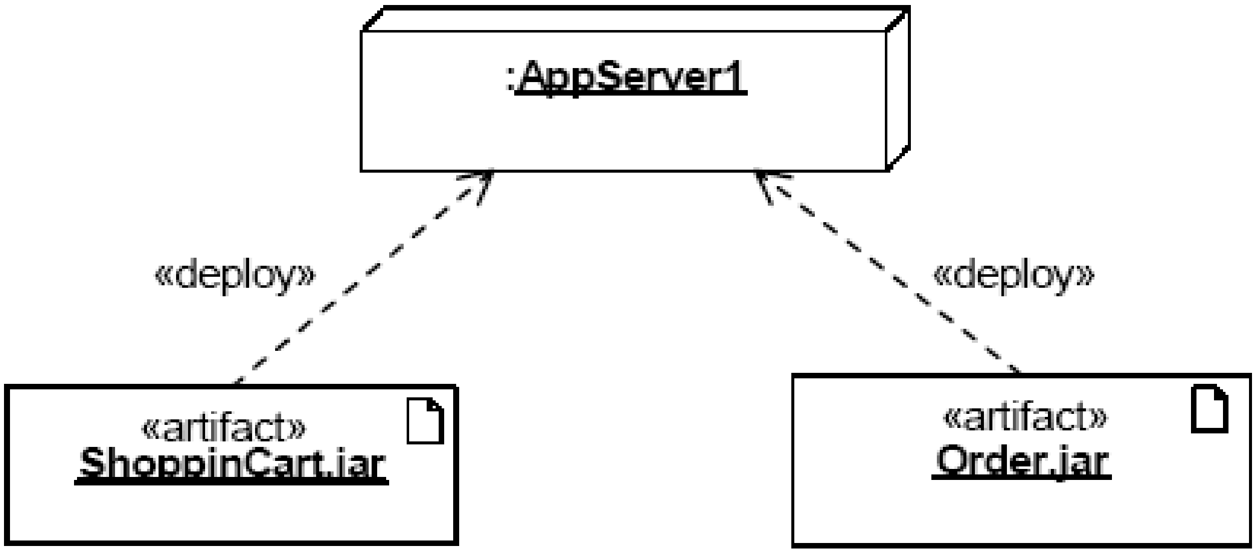


Figure 10.9 - Alternative deployment representation of using a dependency called «deploy»

3.2 Deployment Diagram - Alternative Notations

Figure 10.10 - Textual list based representation of the deployment location of artifacts

3.3 Deployment Specification

- A mechanism to parameterize a deployment relationship
- Its element is expected to be extended in specific component profiles.
- Stereotypes examples for deployment specification :

  «concurrencyMode» with tagged values {thread, process, none}

  «transactionMode» with tagged values {transaction, nestedTransaction, none}.

Figure 10.11 - DeploymentSpecification for an artifact (specification and instance levels)

3.3 Deployment Specification

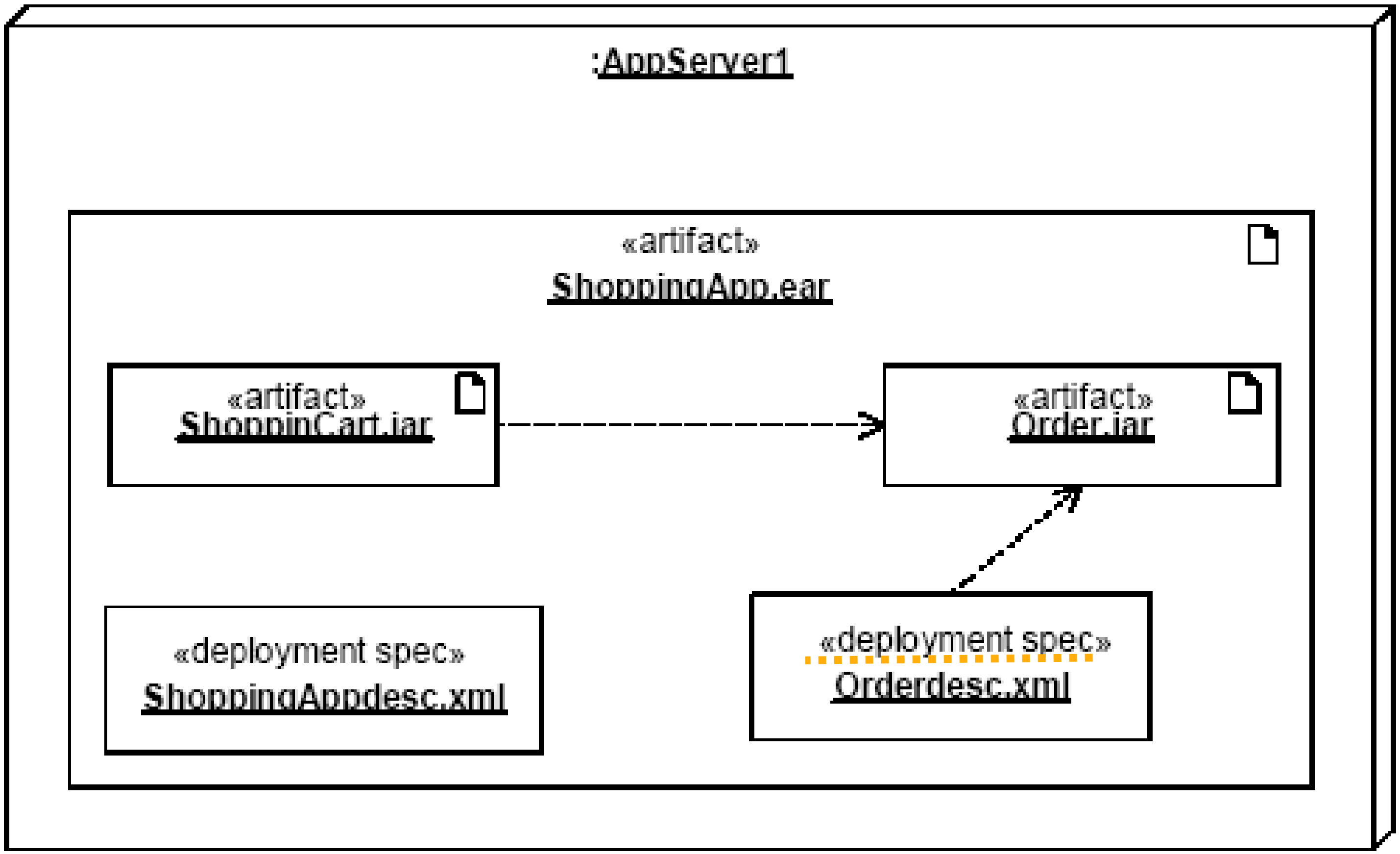


Figure 10.12 - DeploymentSpecifications related to the artifacts that they parameterize

3.3 Deployment Specification

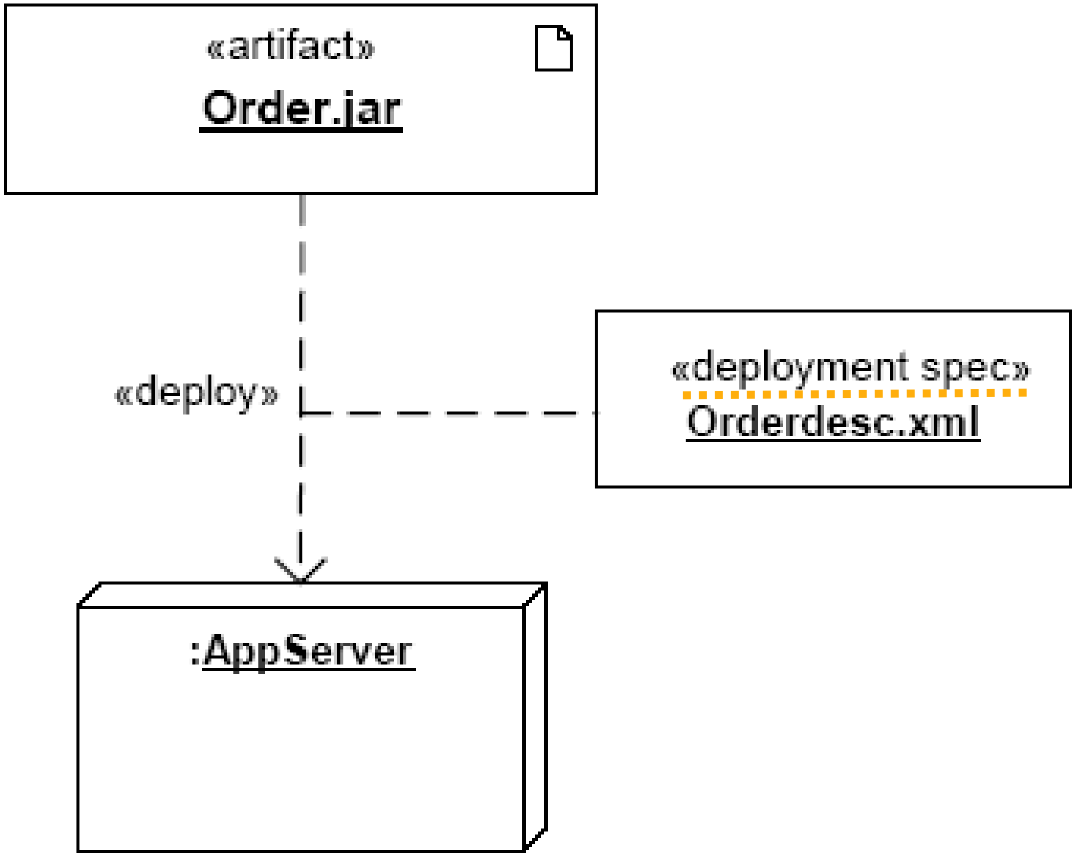


Figure 10.13 - A DeploymentSpecification for an artifact

3.4 Device

- A Device is a ***physical*** computational resource with processing capability.
- May consist of other devices.

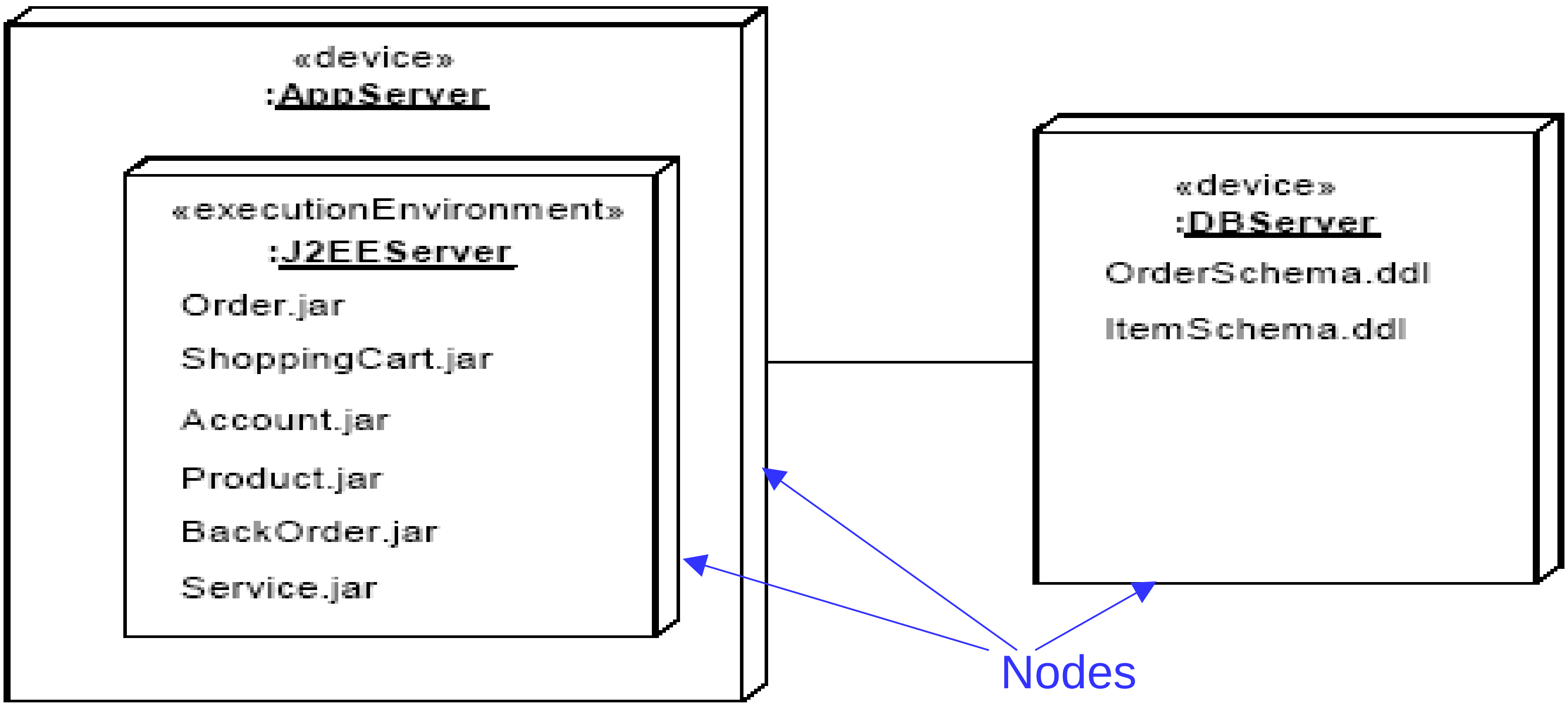


Figure 10.14 - Notation for a Device

3.5 ExecutionEnvironment

- An ExecutionEnvironment is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts.

Figure 10.15 - Notation for a ExecutionEnvironment (example of an instance of a J2EEServer ExecutionEnvironment)

3.6 Node

- A node is computational resource.

- Nodes can be interconnected through communication paths to define network structures.

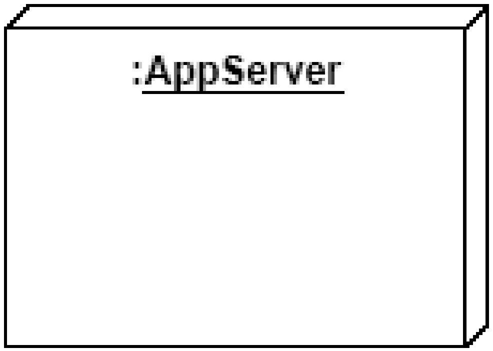


Figure 10.16 - An instance of a Node

3.7 Communication Path

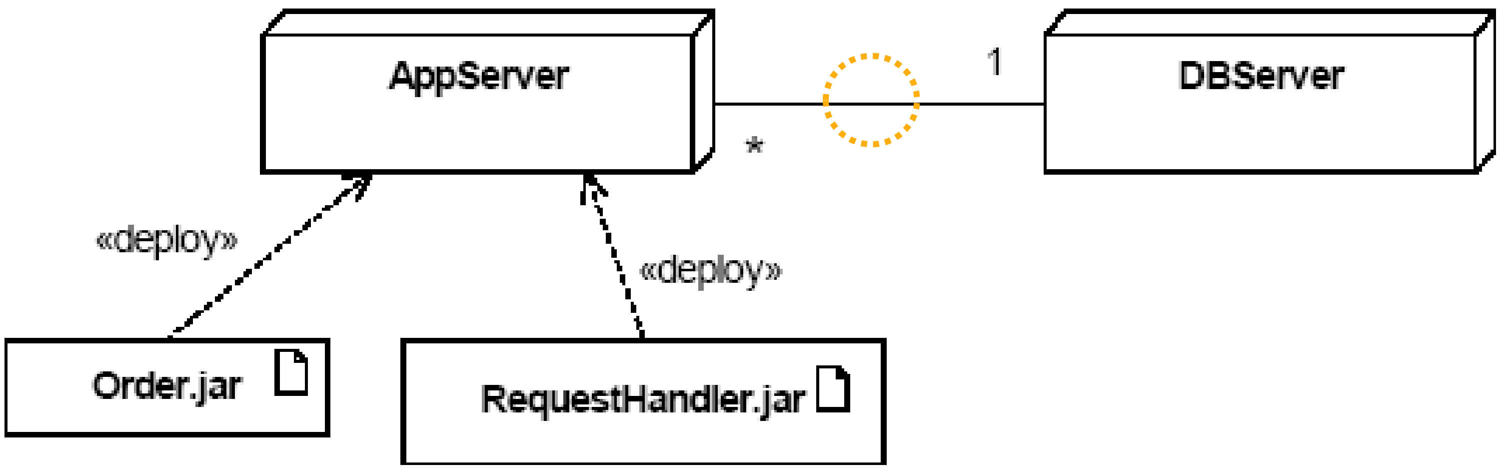


Figure 10.17 - Communication path between two Node types with deployed Artifacts

- Nodes may be connected by associations to other nodes.
- A link between node instances indicates a communication path.

3.7 Communication Path

Any problem in this diagram?

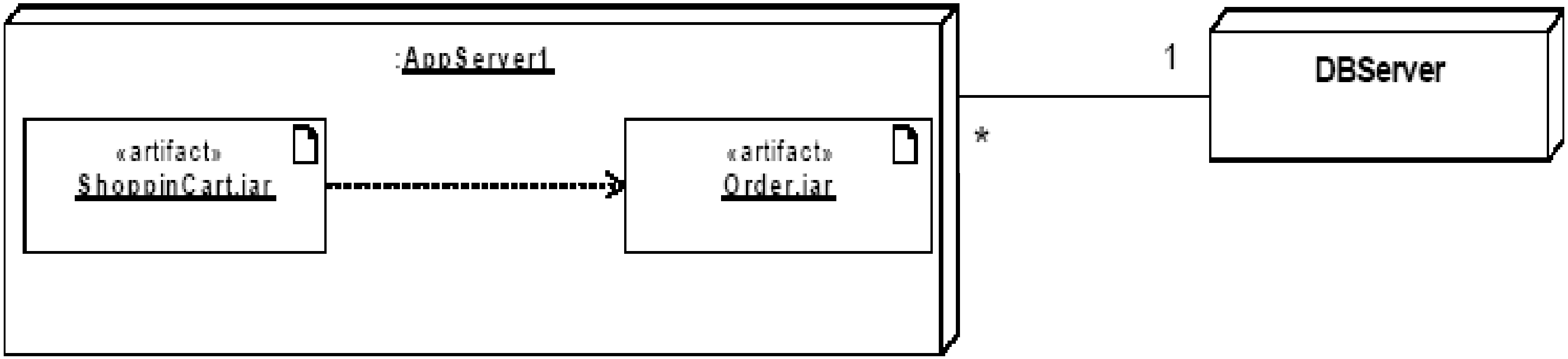


Figure 10.18 - A set of deployed component artifacts on a Node

3.7 Communication Path

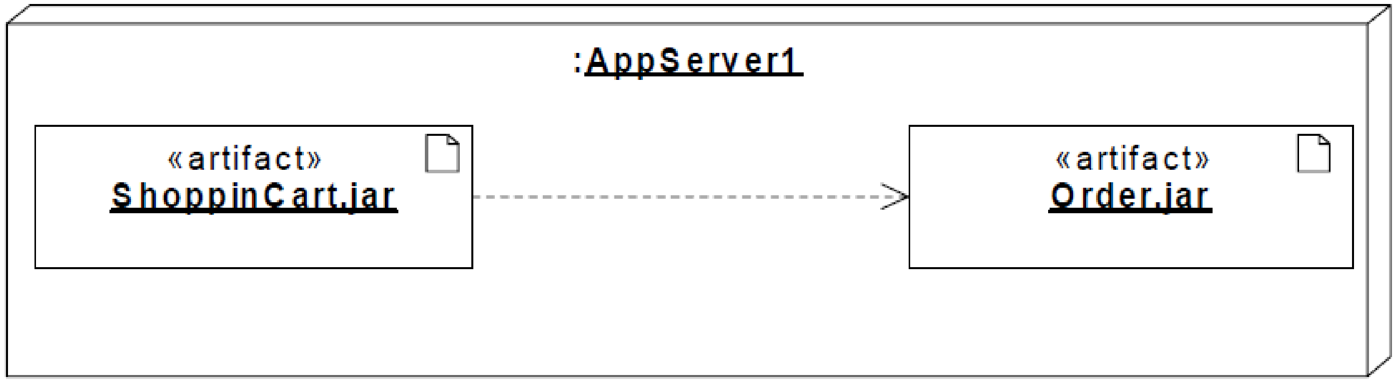


Figure 10.18 - A set of deployed component artifacts on a Node

4. Documenting C&C View using UML2.0

C&C view

- [Logical] C&C view is a technically important architecture view
  - It allows various reasoning about architecture because of its connector centric view presentation.

- ksw: C&C View - misleading name
  - [Clements 11] used the term "C&C style".

    It is better. But "C&C" is more pervasive than to be called just "a style"

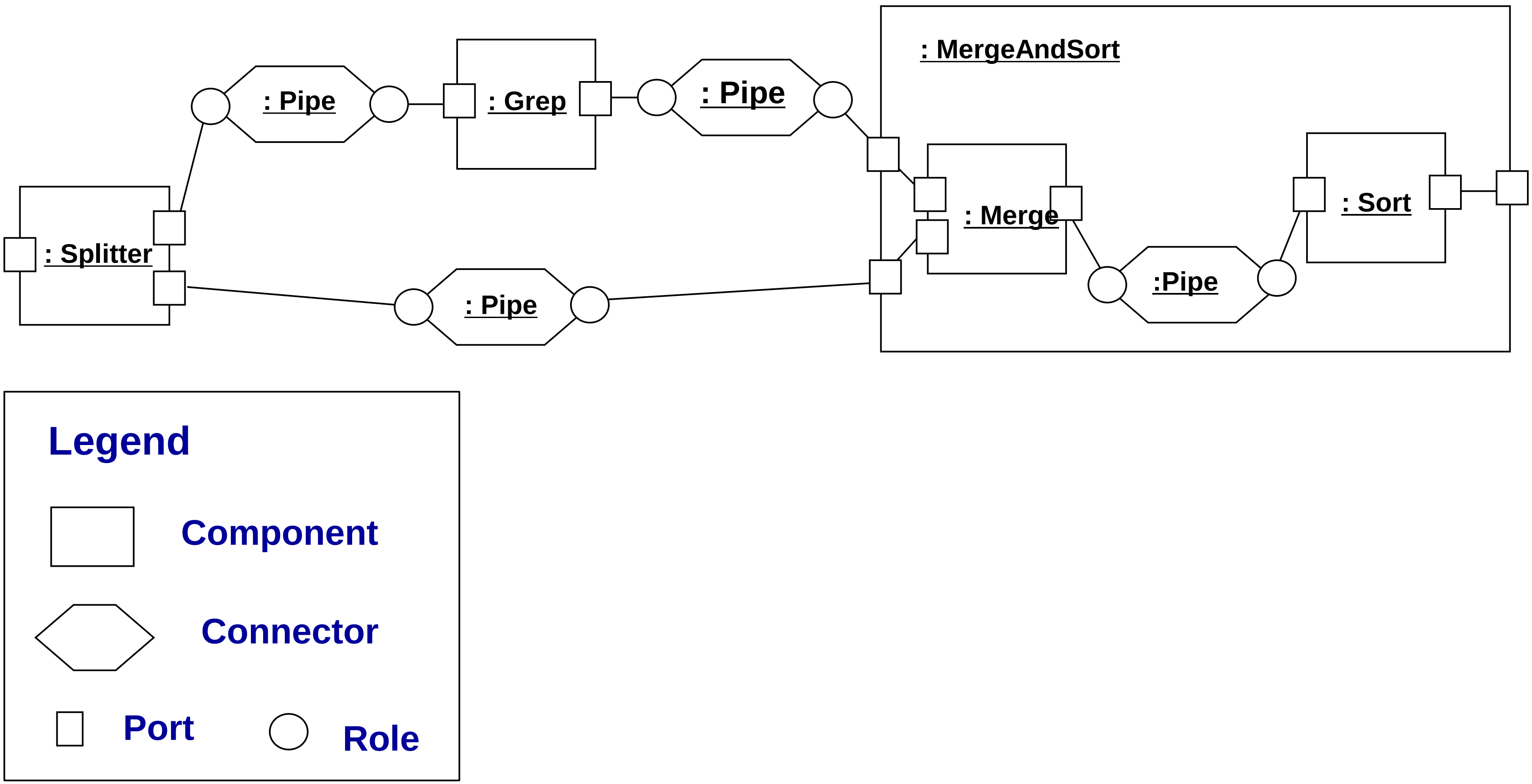
    E.g.) Siemens conceptual and execution views are both C&C style.

- Influence of C&C view on UML evolution

Elements of C&C view

- **component**:
  - represents system's computational elements or data repository
  - may contain interfacing elements called ports
- **connector** :
  - glues components
  - may contain interfacing elements called roles
- **port/role**: . . .
- **property**:
  - describe additional information about architectural elements that is beyond structure
- **system** : consists of components and connectors

C&C View Example – Acme Notation



Ksw - Classification of Software Concepts

	Requirements	Design	Coding	Deployment	Execution (= Runtime)
Language	Domain	UML	Java	System+ Env	
Concepts	Entity, People, **…** Protocol, Agreement Interaction	Class Object Interface	Class Object Interface	Application Class Interface Info	Process Thread
	… **…**	Component Connector	Class(es)	Component	Connectors
Relationship	Is related to, knows, provides service, sends info or data	Association	Object reference variable		
Variability	Commonality & Variability	Generalization	"extends"		

- Semantic mismatches
- Same terms for semantically different things in different stages of development

Ksw - Classification of Software Concepts

- Connectors
  - tight connection: transfer of control
  - loose connection: synchronization of applications, processes and threads or combinations of them
  - very loose connection: asynchronous communication
- Connector Design vs. Implementation
  - initially: two entities are "related"
  - intermediate stages: specific properties of "connection" are more and more exposed such as communication types (1-to-1, 1-to-many, BB, ...), its services (what interfaces it provides), its constraints (what services it relies on)
  - finally: concrete instance of communication mechanism

☛ Design can be refined more and more to get closer to implementation.

☛ Still runtime objects are entities different from design entities and implementation entities.

☛ When the "concepts" from different stages of development are mixed we should be clear about which concepts belong to which stage.

Shortcomings of Using UML 1.4

- Several natural modeling constructs for components and component types.
- However,
  - Connector in UML 1.4 is not a first-class concept:
    - Have to be encoded as associations or as components
  - Interfaces
    - Not allow the representation of multiple runtime points of interaction, (many might have the same “signature”).
  - Hierarchical decomposition of components
    - No way to provide a more detailed architectural model scoped to a single component
  - Internal structure
    - No way to indicate that, when a component instance is created, it must have the indicated substructure.

Documenting C&C Views with UML 2.0

- Criteria for choosing one way over other possibilities:
  1. Semantic match: constructs should map intuitively to architectural features
  2. Visual clarity: description should bring
     - conceptual clarity to a system design
     - avoid visual clutter
     - highlight key design details.
  3. Completeness (= Expressiveness): All relevant architectural features for the design should be represented
  4. Tool support: Not all uses of UML are supported equally by all tools (particularly when specializing UML)

Acknowledgement The contents of this section heavily relies on the following work

J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, J. Rodrigo Oviedo Silva, “Documenting Component and Connector Views with UML 2.0”CMU/SEI-2004-TR-008, *April 2004*.

4.1 Components

- Ports and structured classifiers in UML 2.0 clearly represents component ports (interfaces) and component decomposition.

Strategy 1: Using UML Classes

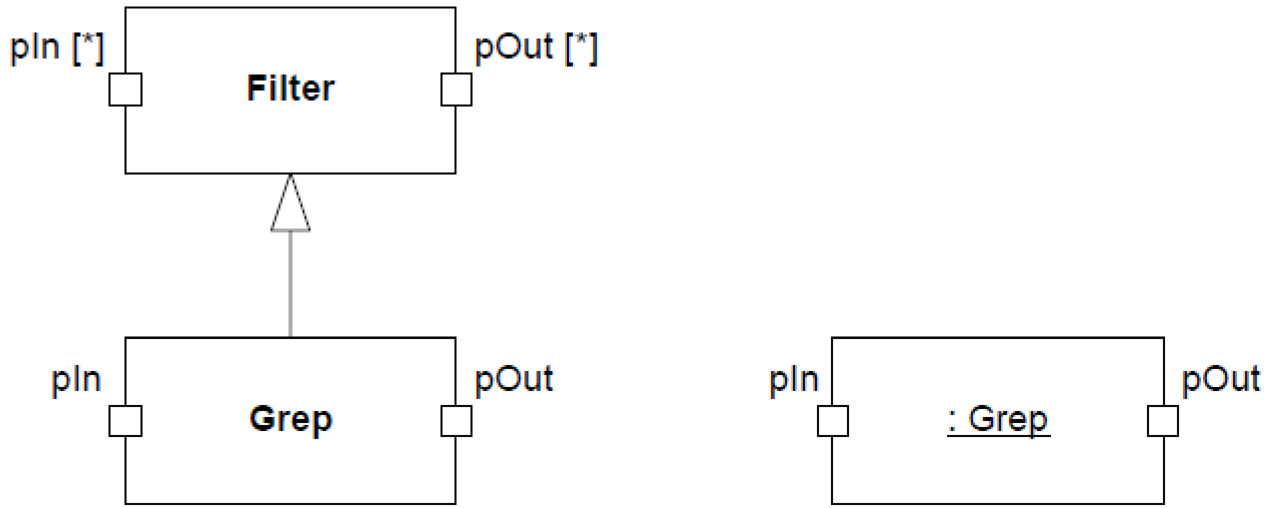


Figure 14: C&C Types as UML Classes and C&C Instances as UML Objects

Strategy 2: Using UML Components

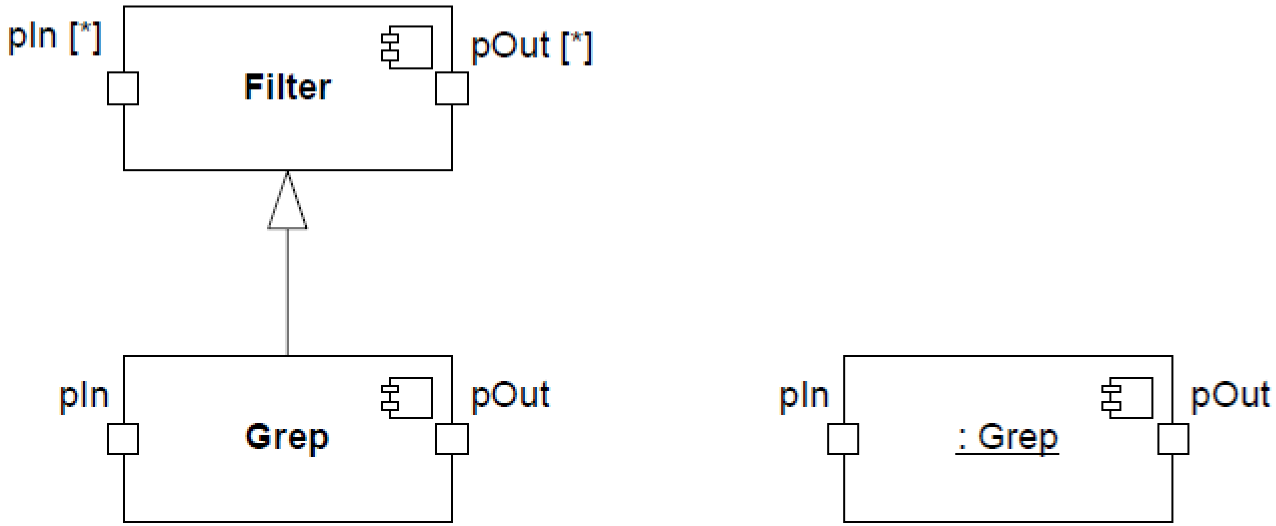


Figure 15: C&C Types as UML Component Types and C&C Instances as UML Component Instances

Choosing

- Either strategy is adequate since both have the same expressiveness (i.e., can represent all relevant C&C component features).

- The choice may rely more on the semantic match:
  - For readers familiar with of UML 1.4, an unintentional implementation bias may be difficult to overcome when using UML components to represent C&C components.
  - While the UML 2.0 enables the use of components throughout the development life cycle, there are potential problems with using the same construct for representing both a design abstraction and implementation, particularly when there is no one-to-one mapping between the two.

4.1 Component - Comparison

	Expressiveness	Visual clarity	Semantic match
1. UML Class	OK	C&C connector로 class를 쓸 경우 같이 쓰면 명료성이 떨어짐.	OK
2. UML Component	OK	OK (esp. with component icon)	OK UML 1.4에서의 component의 개념에 익숙한 사용자에게 혼란을 야기할 수 있음

4.2 Connectors

- While UML 2.0 improves its suitability for documenting components and ports, similar improvements supporting connectors are missing.

Strategy 1: Using UML Connector (1/2)

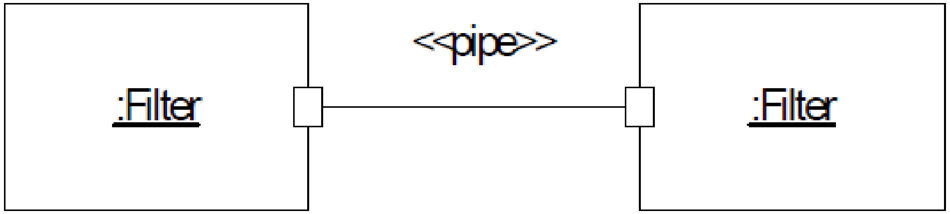
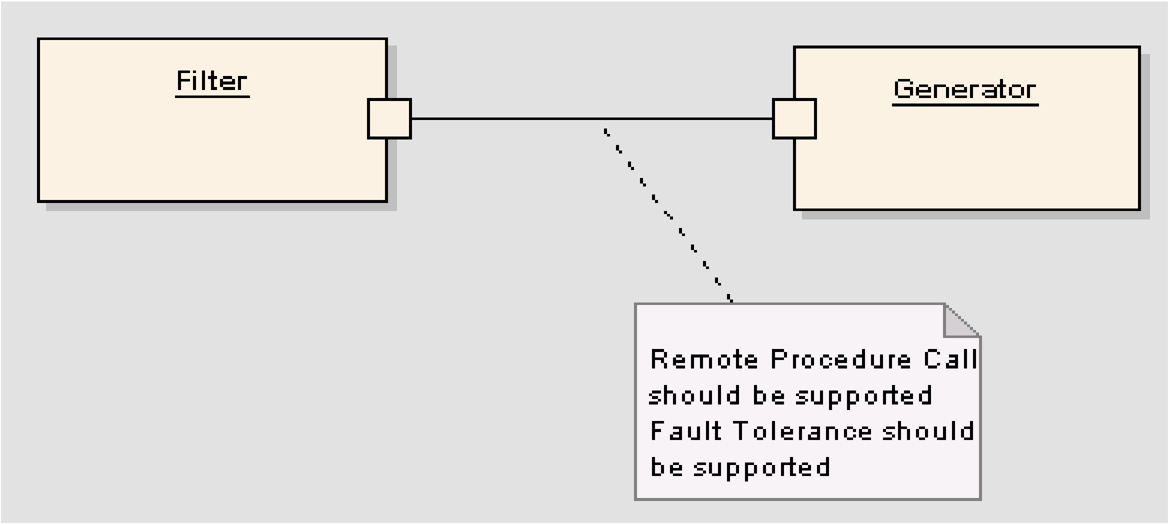


Figure 17: A C&C Connector as a UML Association (Link)

- Visually distinct from component
- Good to identify where different types of connectors are used in a system.
- Limitations:
  - Represent a potential for interaction between two classifiers but do not have any behavior of their own.
    - The roles (interfaces) of C&C connectors cannot be defined.
    - C&C connector semantics cannot be defined because associations cannot own attributes or have behavioral descriptions.

Strategy 1: Using UML Connector (2/2)



- Useful at the initial stage of development when detailed information about connector is not necessary

- "Note" can be used to record brief information

Strategy 2: Using UML Association Classes (1/2)

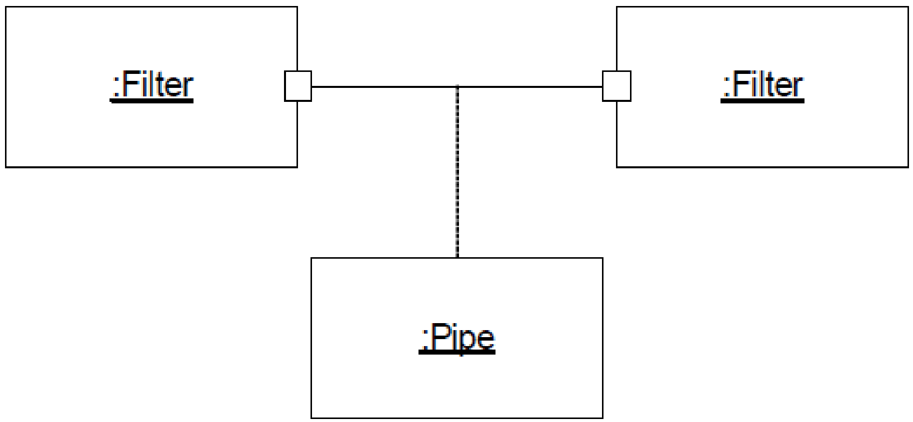


Figure 18: A C&C Connector as a UML Association Class (Link Object)

- Addresses many of the limitations of using association.
- Allows rich semantic descriptions, including attributes and behavior
- Can also have substructure if decomposition of the connector is useful (e.g., to show details of how the connector is to be implemented).
- Allows connector types to be defined independently of usage

Strategy 2: Using UML Association Classes (1/2)

- Also allows C&C connector roles to be represented using UML ports on the association class

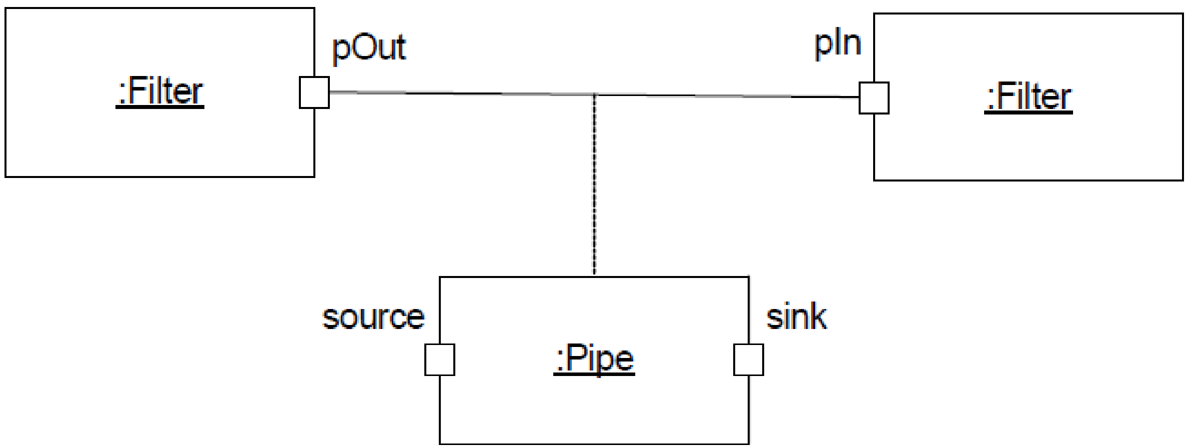
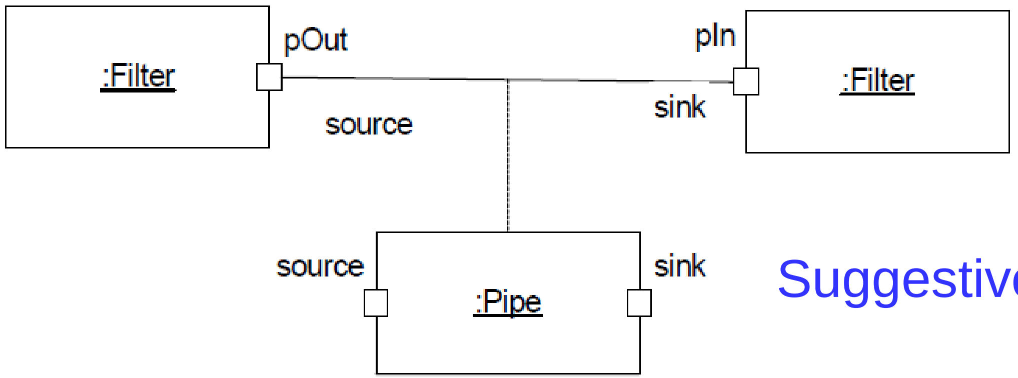


Figure 19: A C&C Connector as a UML Association Class with Ports

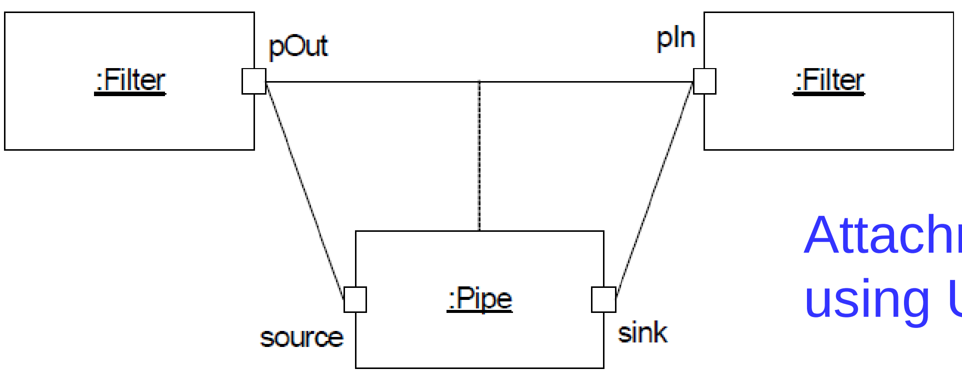
- Not clear whether the source C&C connector role is attached to pOut or pIn.

Resolving ambiguity



Suggestive rather than formal

Figure 20: Link Role Names Are Used to Represent Attachments



Attachment is shown explicitly using UML assembly connectors

Figure 21: Assembly Connectors Are Used to Represent Attachments

Strategy 3: Using UML Classes (1/2)

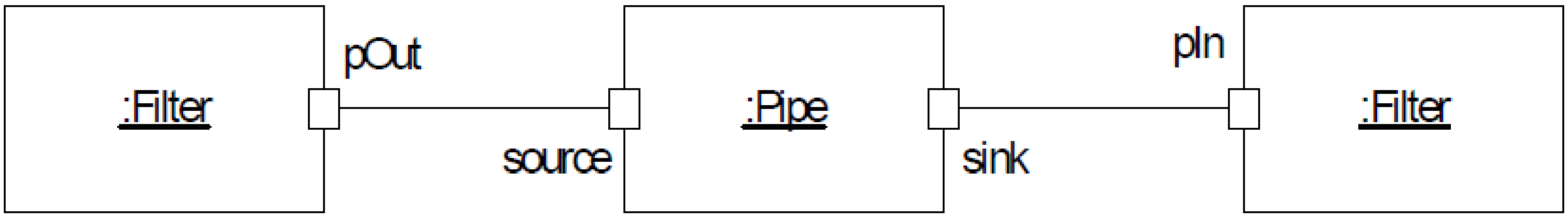


Figure 22: A C&C Connector as a UML Class (Object)

- UML classes offers essentially the same solution, but without the redundant association portion of the association class and its associated visual clutter.
- C&C attachments are always represented explicitly using UML assembly connectors.
- Resolves the component and connector attachment problems of the second strategy while retaining its expressiveness.

Strategy 3: Using UML Classes (2/2)

- But presents the poorest visual distinction between C&C components and connectors

  => Dilutes a benefit of a C&C view—the ability to quickly identify the principle computational elements and understand their interaction patterns.

  => Can be mitigated by using different UML concepts

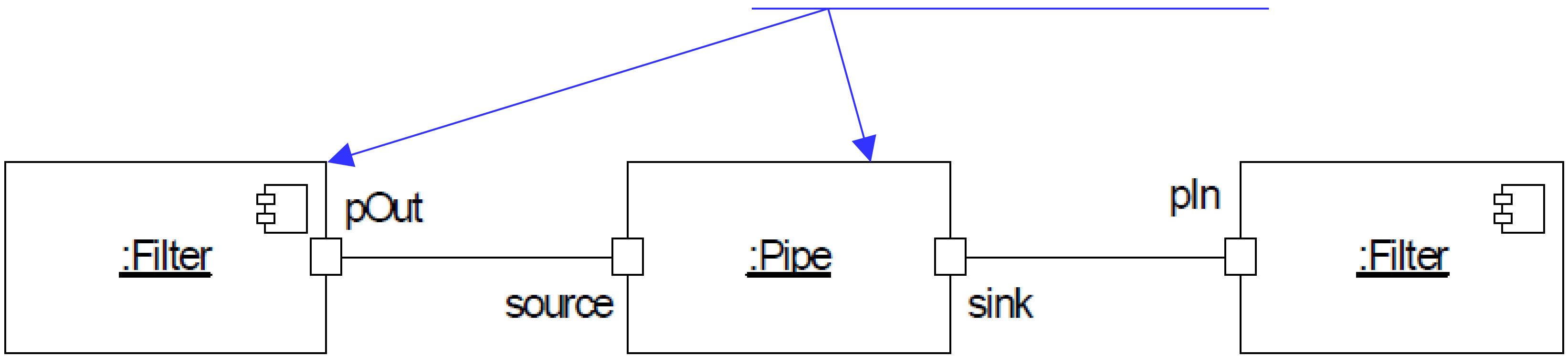


Figure 23: Using a UML Class for a C&C Connector and a UML Component for a C&C Component

The ideal approach [Ivers 04]

- Combine the expressiveness of this strategy with a different visualization (such as that from the first strategy) by using UML's stereotype, which permit the visualization of stereotyped elements to be customized.

Figure 24: A C&C Connector as a UML Stereotyped Class with Custom Visualization

- Requires graphical support not offered by most UML tools

Choosing

- Depends on various factors:
  - Are you identifying which types of connectors are used, describing what effect the connectors have on component interaction, and/or providing enough design information to guide the connector implementation?
  - Where are you in the system's development life cycle (i.e., which decisions have been made and which have been deferred)?
  - How are you representing components?

    Are their visualizations distinct from those of the connectors?
  - What tool support is available?

4.2 Connector - Comparison

	Expressiveness	Visual clarity	Semantic match
1. UML Connector	이름을 제외한 다른 의미 기술이 부족, 간략한 정보를 기술하기 위해서는 Note를 사용가능	OK	First class element인 C&C connector와 차이가 있음
2. UML Association Class	port 와 role의 연결을 제외하고 문제를 가지지 않음	association class의 추가로 인해 복잡도가 높아짐	connector와 component의 의미적 구별이 힘들 수 있음
3. UML Class	OK	Connector와 component 의 시각적 구분이 어려움	connector와 component의 의미적 구별이 힘들 수 있음

4.3 Ports/Roles

- UML 2.0 Port is very well suited for documenting C&C port:
  - Defines explicit interaction point of classifier (including Component and Class)
  - Can have multiple required and provided interfaces
  - Can define port type through class or interface

Figure 16: C&C Ports as UML Ports

4.4 Properties

- C&C properties are used to capture *semantic* information about a system and its elements that goes beyond structure
  - quality attributes such as
    - a connector's throughput
    - a component's response time, or
    - a component's mean time to failure.
  - Other properties such as
    - thread priority

- UML 2.0 property concept is not a good choice for C&C view's properties because it represents a structural feature
  - "Represents *a set of instances* that are *owned* by a containing classifier instance."

=> Need other strategies

Strategy 1: Using UML Tagged Values

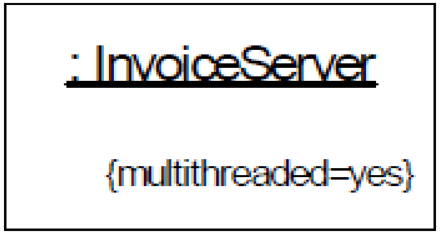


Figure 27: Architectural Properties as UML Tagged Values

- A *tagged value* is an explicit definition of a property as a name-value pair [OMG 03].
- Limitations:
  - There is no explicit documentation of the value's type.
  - Can be used for instance but not for type

Strategy 2: Using UML Attributes (1/2)

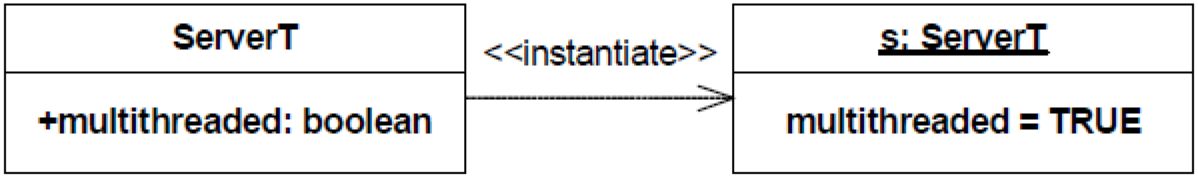


Figure 28: Architectural Properties as UML Attributes

- Easiest way

- Limitations:
  - C&C properties are not structural elements but rather semantic ones.
  - Can cause misinterpretation

    => Could result in code for them when should not

Strategy 2: Using UML Attributes (2/2)

- A variation that overcomes such misinterpretation is to use a stereotype denoting that an attribute is semantic, not structural.

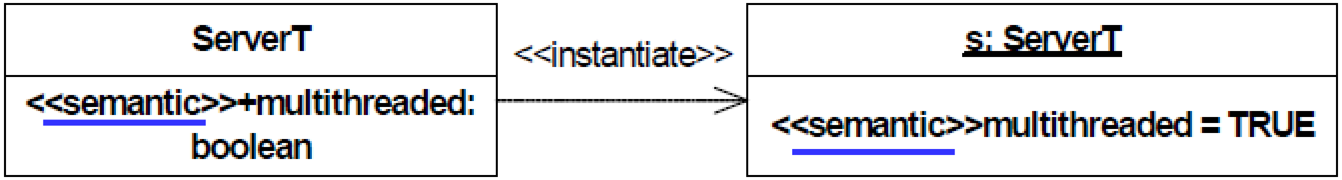


Figure 29: Architectural Properties as UML Stereotyped Attributes

- Limitations:
  - Tools that do not recognize the stereotype may produce inappropriate results, such as code for those attributes as if they were structural.

Strategy 3: Using UML Stereotypes (1/3)

- A more accurate, but more complex, way to represent semantic information is through stereotypes.

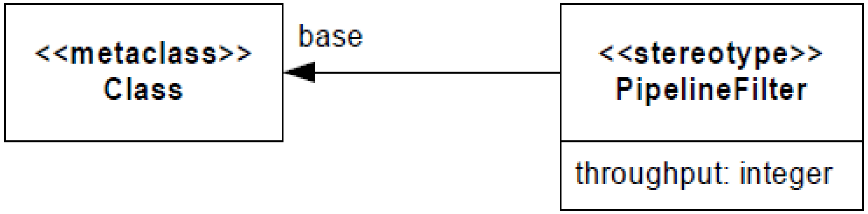


Figure 30: Architectural Property Captured in a UML Stereotype

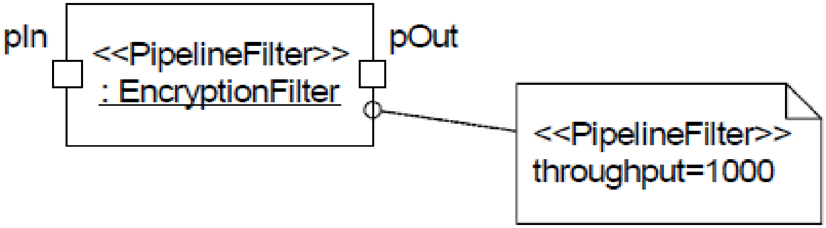
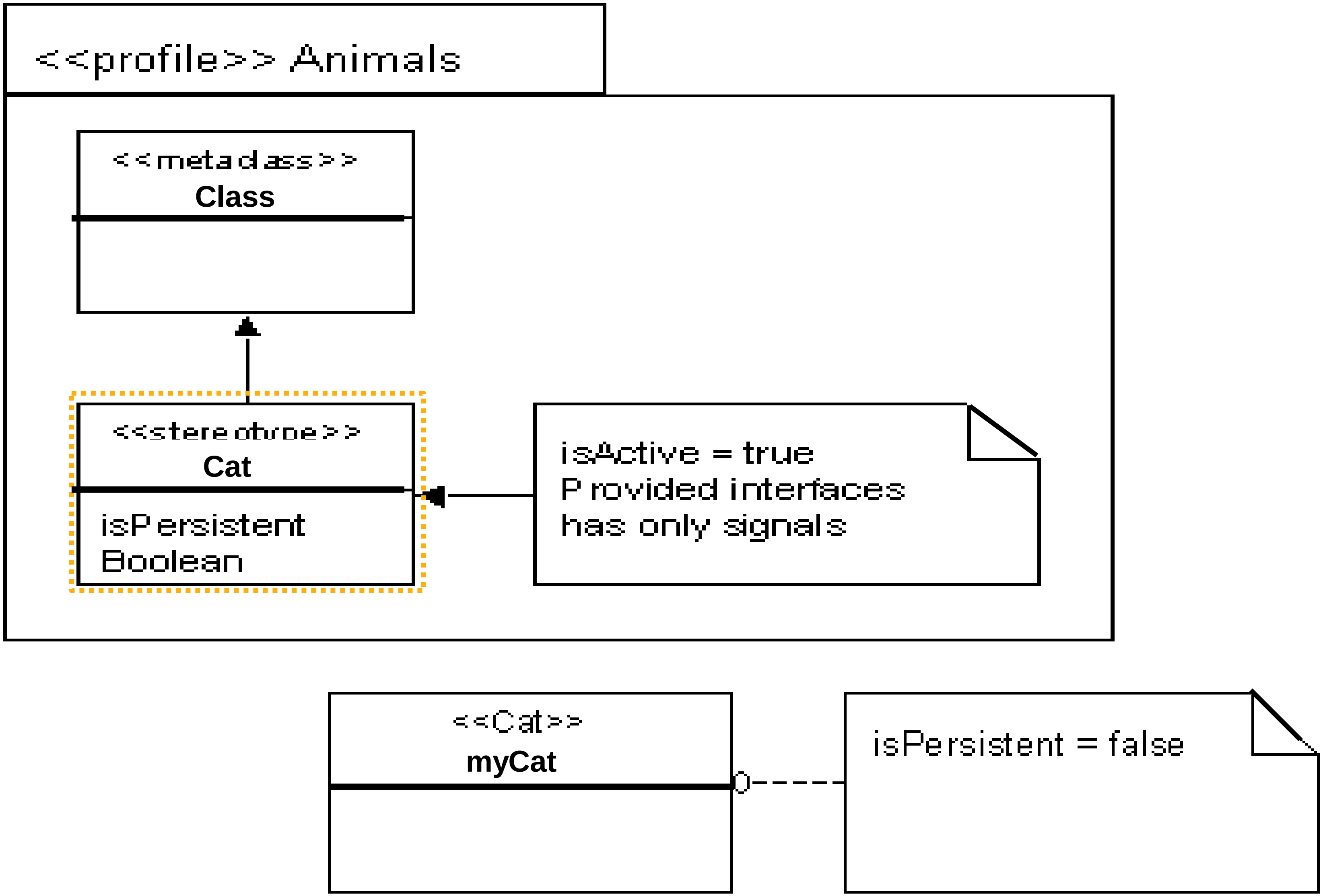


Figure 31: Stereotype Applied to an Instance

Strategy 3: Using UML Stereotypes (2/3)



Strategy 3: Using UML Stereotypes (3/3)

- Although the **`throughput`** tagged value is in the EncryptionFilter class definition, there is no requirement that a value must be assigned to it.

- Doing so would give semantic information that may not make sense at this level of abstraction because the stereotype may extend multiple metaclasses.

- Instead, we can add a new extension that allows the stereotype to extend not only the EncryptionFilter class but also its instances (where assigning value to the **`throughput`** tagged definition does make sense).

Choosing

- The first strategy
  - Adequate if analysis tools are not used and properties are not required for all instances of a class
  - Guide the design or convey information
- The second strategy
  - Good if the properties are mandatory to support analysis, but the implementation consequences are not terribly detrimental (e.g., there are no memory constraints and there is good documentation regarding the unnecessary program variables).
  - Ensures that all components consider the values
  - Provides explicit documentation.
- The third strategy
  - Provides explicit documentation
  - Lacks the semantic mismatch and potential implementation consequences of the second strategy.
  - Although does not require that values be supplied, provides a stronger hint than the first strategy by providing a placeholder and associated semantics about the type.

4.4 Properties - Comparison

	Expressiveness	Visual clarity	Semantic match
1. UML Tagged Value	value의 타입을 명시하지 못한다	OK	OK
2. UML Attribute	OK	OK	구조적 정보를 나타내는 attribute와 혼동이 있을 수 있음
3. UML Stereotype	OK	OK	OK

4.5 Systems

- C&C views of systems depend on two types of information:

  1) definitions of component and connector types

  2) topologies of instances of component and connector types forming the system

C&C View	UML 2.0
System	Composite structure diagram
Component	Component Instance or Object
Connector	Connector, Association Class or Class
Connector's Behavioral Constraint	<<protocol>> class
Port	Port
Role	Port
Property	Stereotype

1) Component and Connector Types

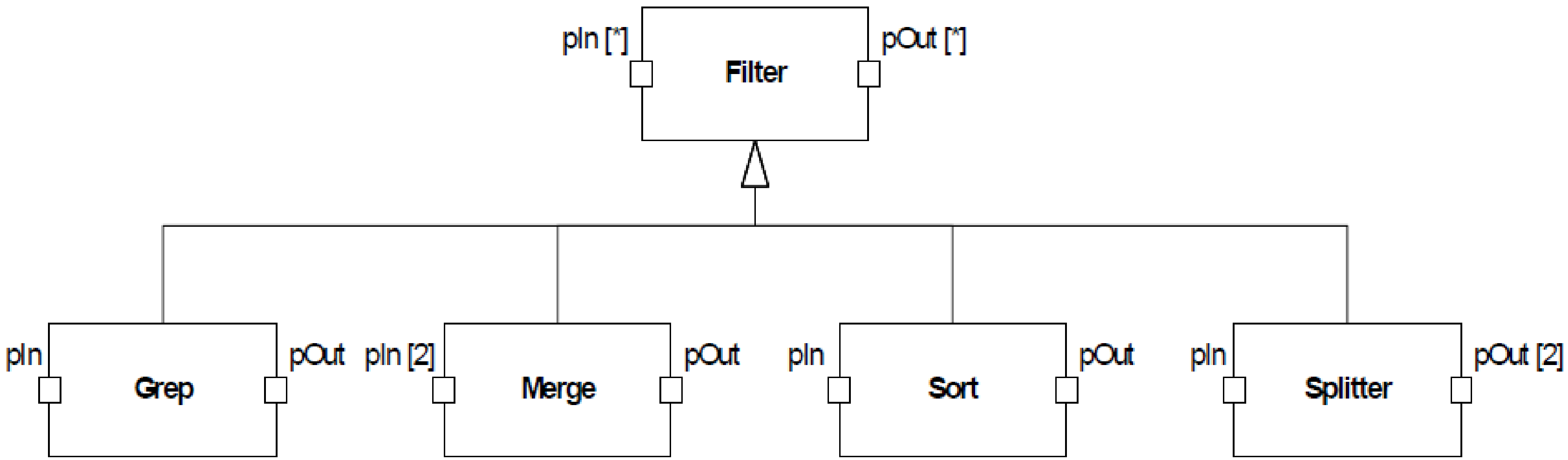
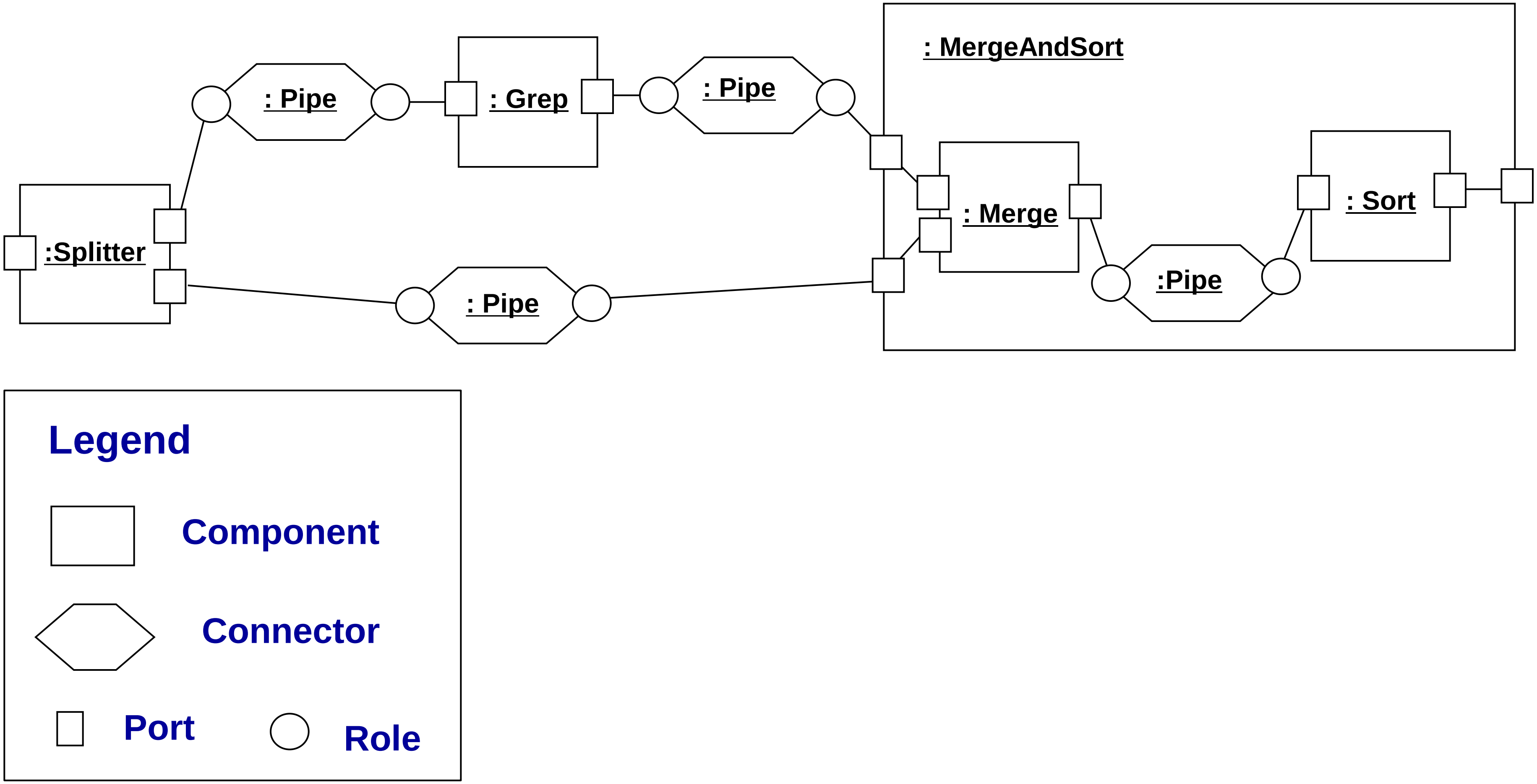


Figure 25: Documentation of Component Type Hierarchy

- For a complete C&C view, interfaces, attributes, and behavioral models should be fully defined for each type.
- Interfaces should be documented in terms of UML ports and interfaces

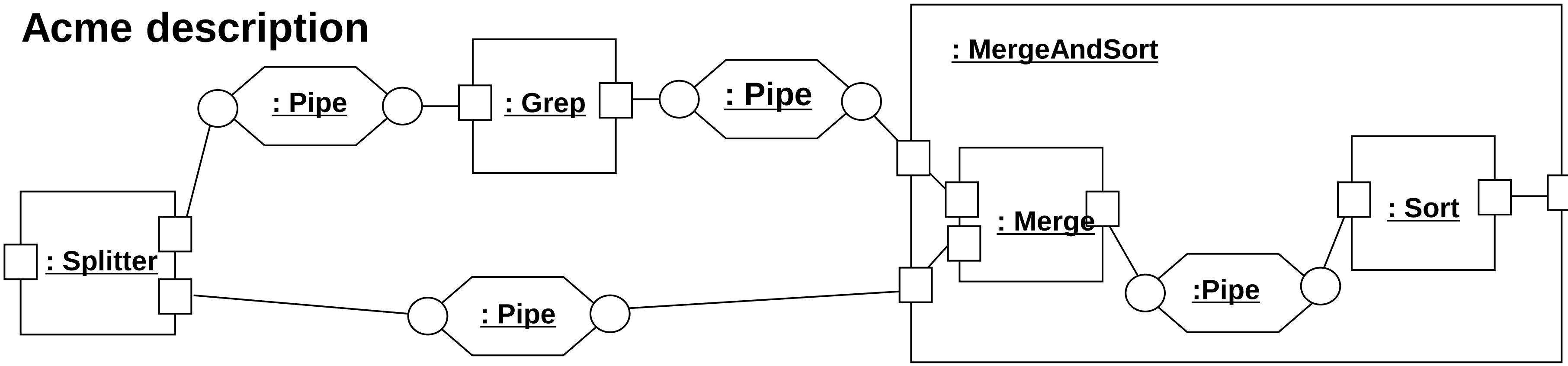
2) Topologies of Component and Connector Instances

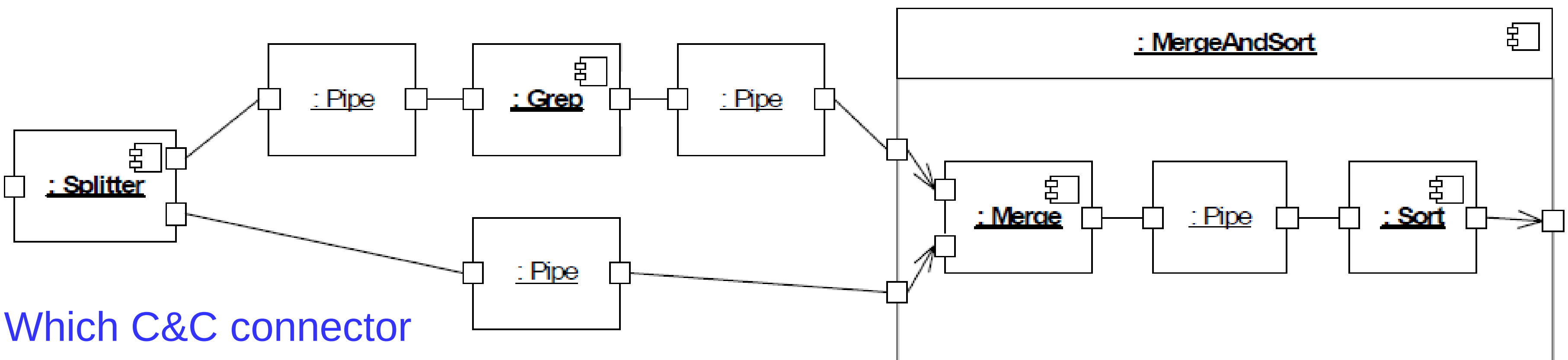
Acme description



2) Topologies of Component and Connector Instances

Acme description



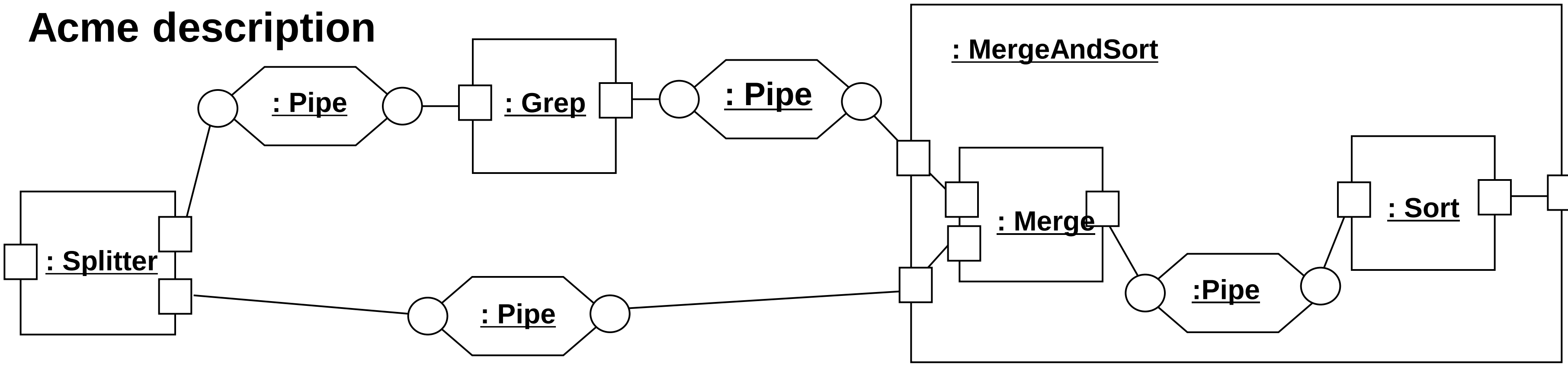


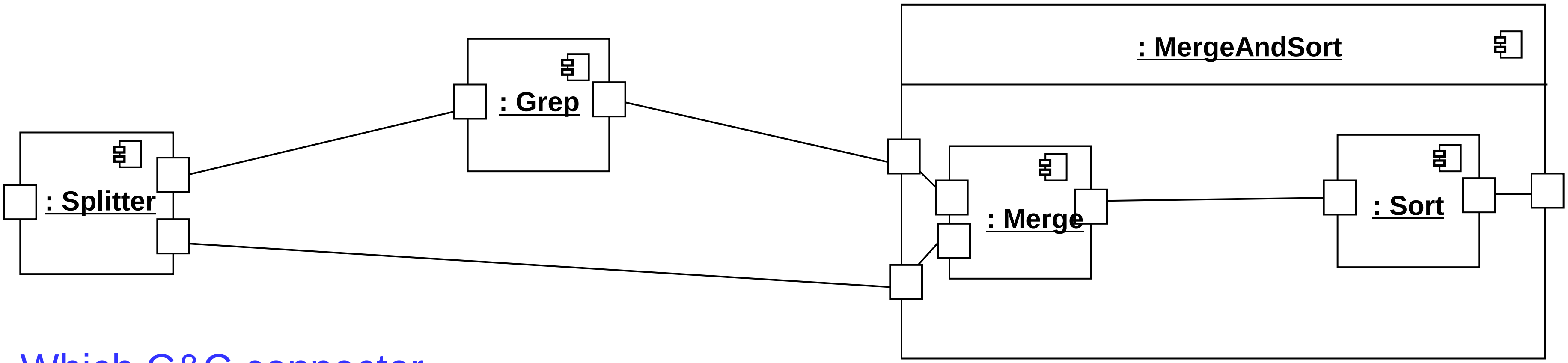
Which C&C connector solution was used here?

UML class

Figure 26: Documentation of a System

2) Topologies of Component and Connector Instances





Which C&C connector solution was used here?

UML connector

4.6 Conclusion

- UML 2.0 has improved C&C architecture view modeling of UML 1.4:
  - Several problems fixed
  - Other problems have been mitigated:
    - Structured classifiers for architectural hierarchy
    - Ports for runtime points of interaction

- However, some problems still remain
  - UML connectors are not first class
    - Difficult to associate detailed semantic descriptions or representations with them.

      => A poor choice for representing C&C connectors
  - Properties
    - Can be represented but not naturally.

5. UML for Multiple Views

Siemens Four Views Approach

- UML originally designed for detailed design at Object level.
- **Concern**: Blurring the distinction between software architecture and detailed design
- **Decision**: The benefits to be gained by a standardized well-understood notation outweighs the drawback

Conceptual View - Elements

Element	UML Element	New Stereotype	Notation	Attributes	Associated Behavior
CComponent	Active class	<<ccomponent>>		Resource budget	Component behavior
CPort	Class	<<cport>>	■	—	—
CConnector	Active class	<<cconnector>>		Resource budget	Connector behavior
CRole	Class	<<crole>>	●	—	—
Protocol	Class	<<protocol>>		—	Legal sequence of interactions

Why use active class for component and connector ?

Conceptual View - Relationships

Relation	UML Element	Notation	Description
composition	Composition	Nesting (or ↑)	A component or connector can be decomposed into a configuration of interconnected components and connectors.
cbinding	Association	————	A port can be bound to a port of the enclosing component. A role can be bound to a role of the enclosing connector.
cconnection	Association	————	A component's port can be connected to a connector's role when both are directly enclosed by the same element.
obeys	Association	obeys	A port or role obeys a protocol.
obeys conjugate	Association	obeys conjugate	A port or role obeys the conjugate of a protocol.

Conceptual View - Artifacts

Artifact	Representation
Conceptual configuration	UML Class Diagram
Port or role protocol	ROOM protocol declaration (uses UML Sequence or Statechart Diagram)
Component or connector behav-ior	Natural language description or UML Statechart Diagram
Interactions among components	UML Sequence Diagram

ROOM = real-time object-oriented modeling; UML = Unified Modeling Language.

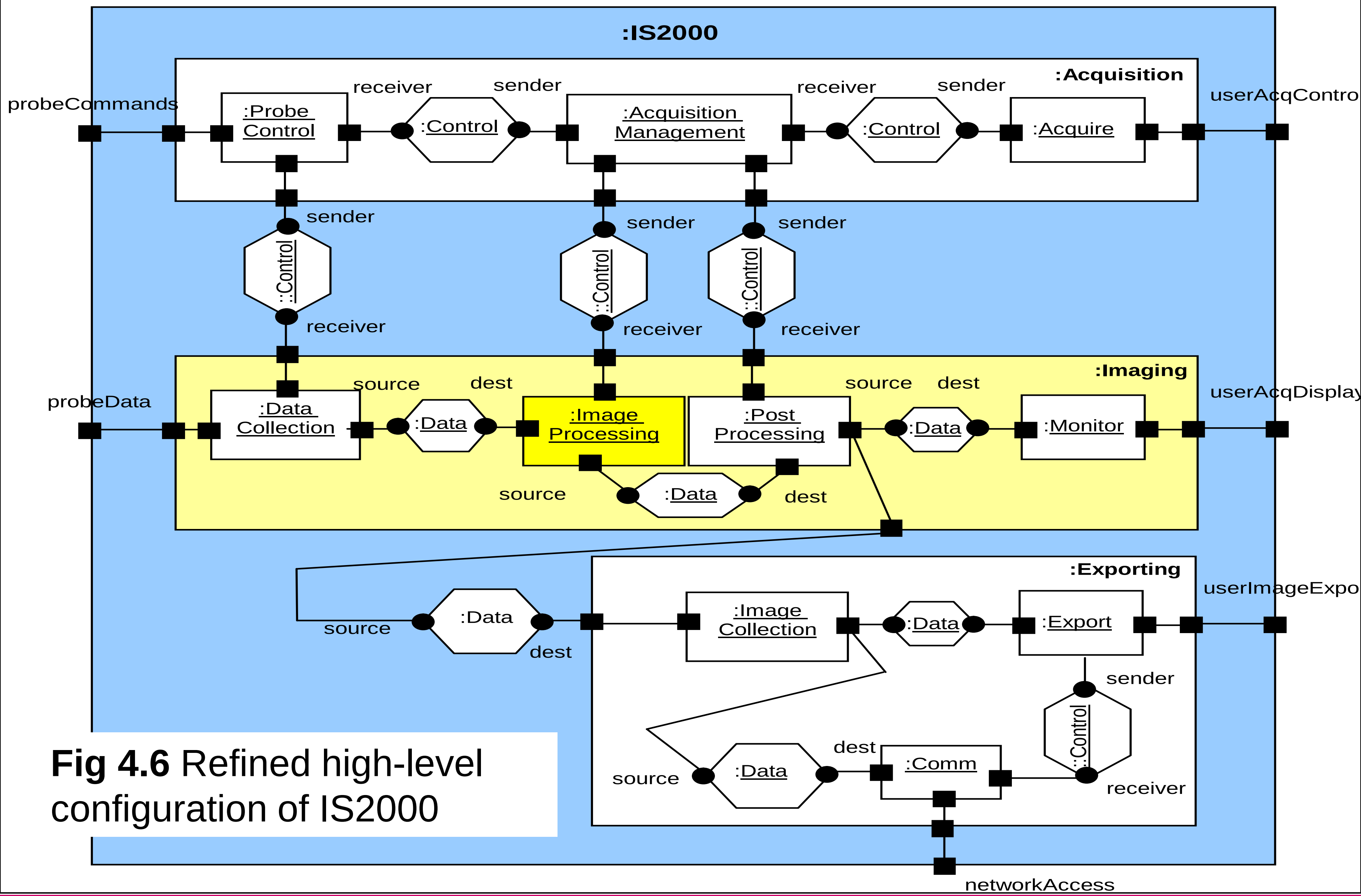
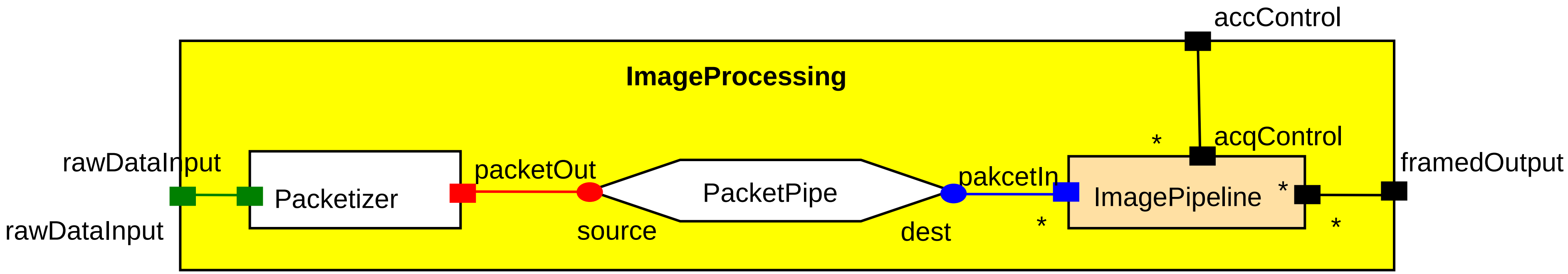
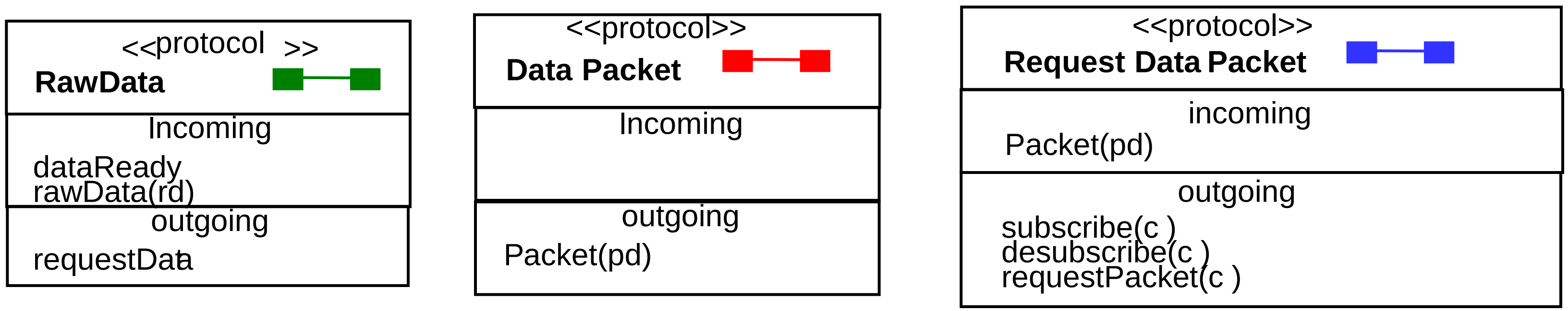
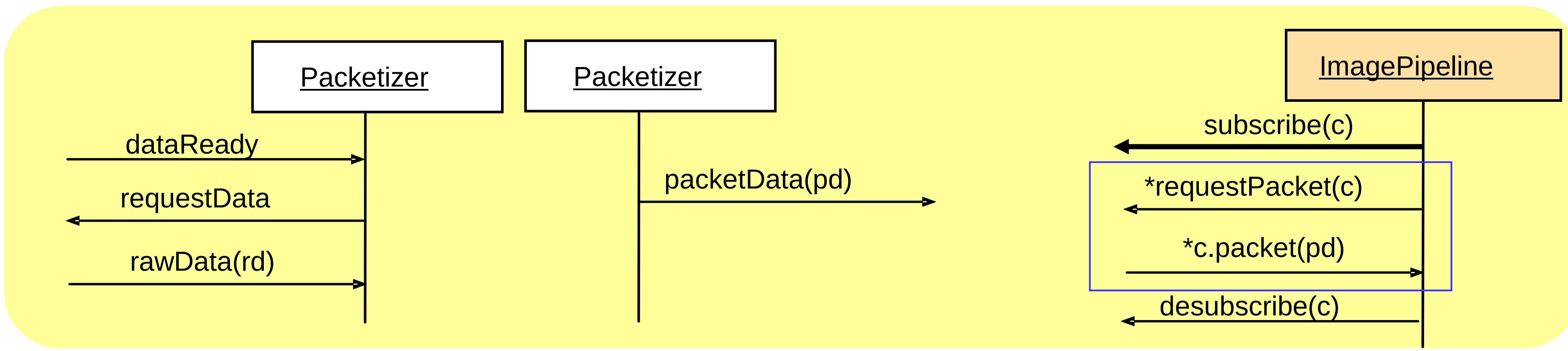


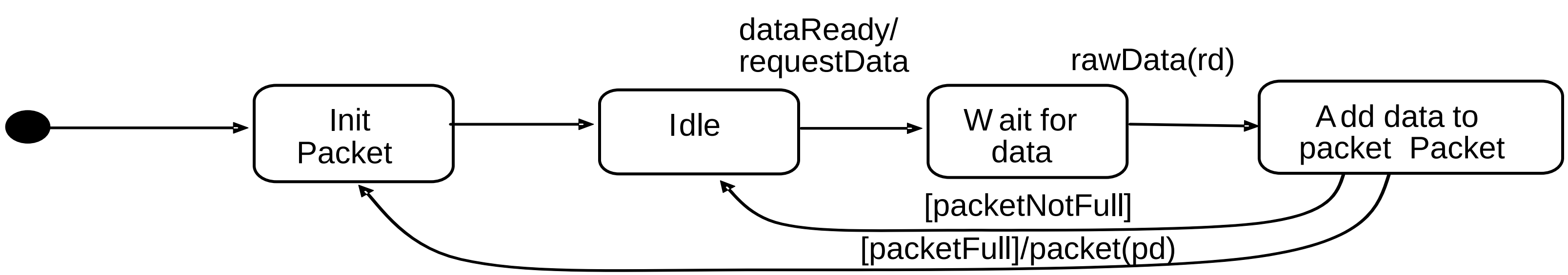
Fig 4.6 Refined high-level configuration of IS2000







Packetizer Behavior



Module View - Elements

Element	UML Element	New Stereotype	Notation
Module	Class	<<module>>	<<module>>
Interface	Interface	—	○ or <<interface>>
Subsystem	Subsystem	—	<<subsystem>>
Layer	Package	<<layer>>	<<layer>>

Module View - Relationships

Relation	UML Element	Notation	Description
contain	Association	Nesting	A subsystem can contain a subsystem or a module. A layer can contain a layer.
composition	Composition	Nesting (or ◆——)	A module can be decomposed into one or more modules.
use (also called use-dependency)	Usage	- - - - - ->	Module (layer) A uses module (layer) B when A requires an interface that B provides.
require	Usage	- - - - - ->	A module or layer can require an interface.
provide	Realization	——— (with ○)	A module or layer can provide an interface.
		- - ▷ (with ▭)	
implement	—	Table row	A module can implement a conceptual element.
	Trace	- - <<trace>> - ->	
assigned to	Association	Nesting	A module can be assigned to a layer.

Module View - Artifacts

Artifact	Representation
Conceptual-module correspondence	Table
Subsystem and module decomposition	UML Class Diagram
Module use-dependencies	UML Class Diagram
Layer use-dependencies, modules assigned to layers	UML Class Diagram
Summary of module relations	Table

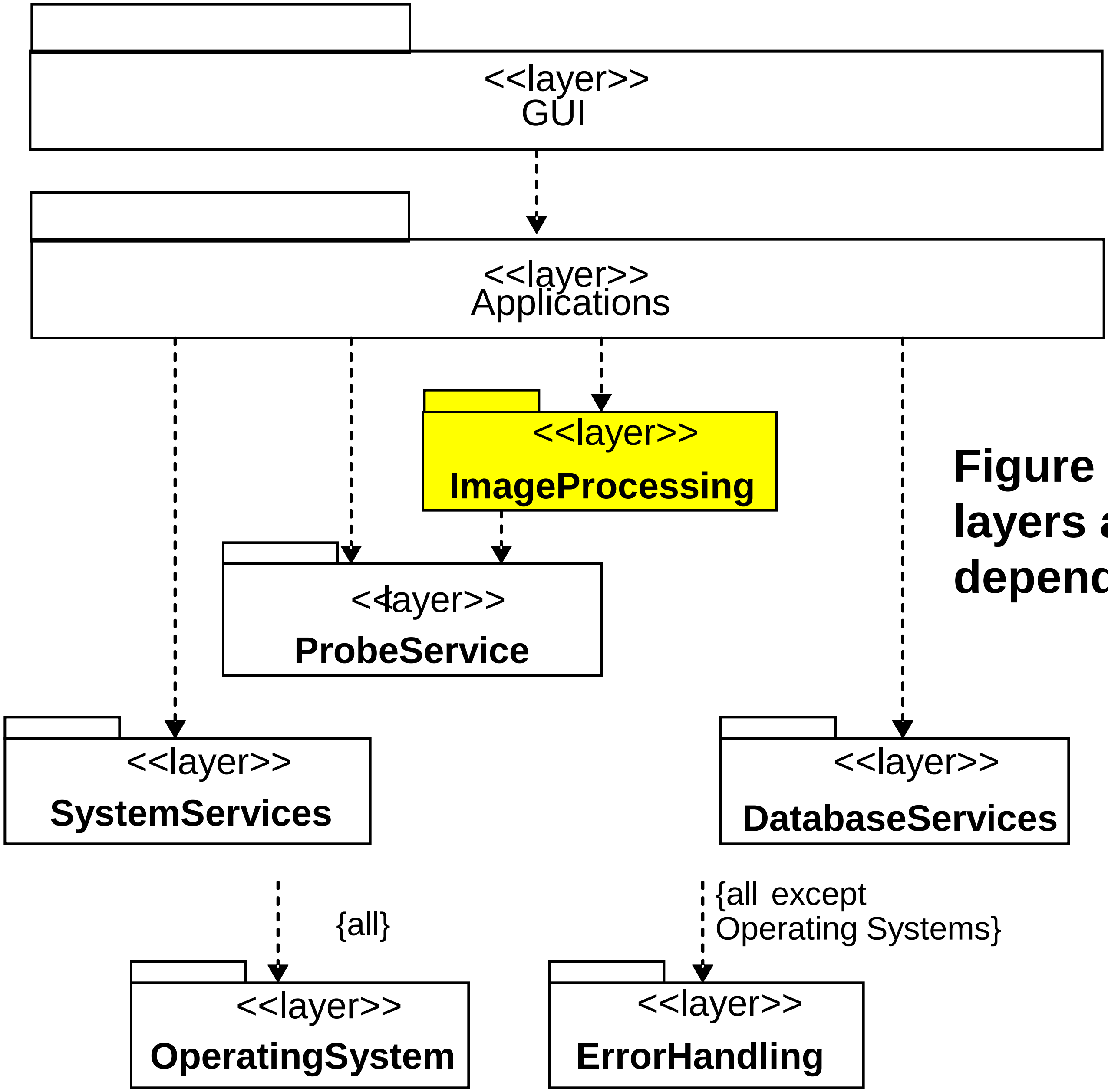


Figure 5.11. Final version of layers and their use-dependencies

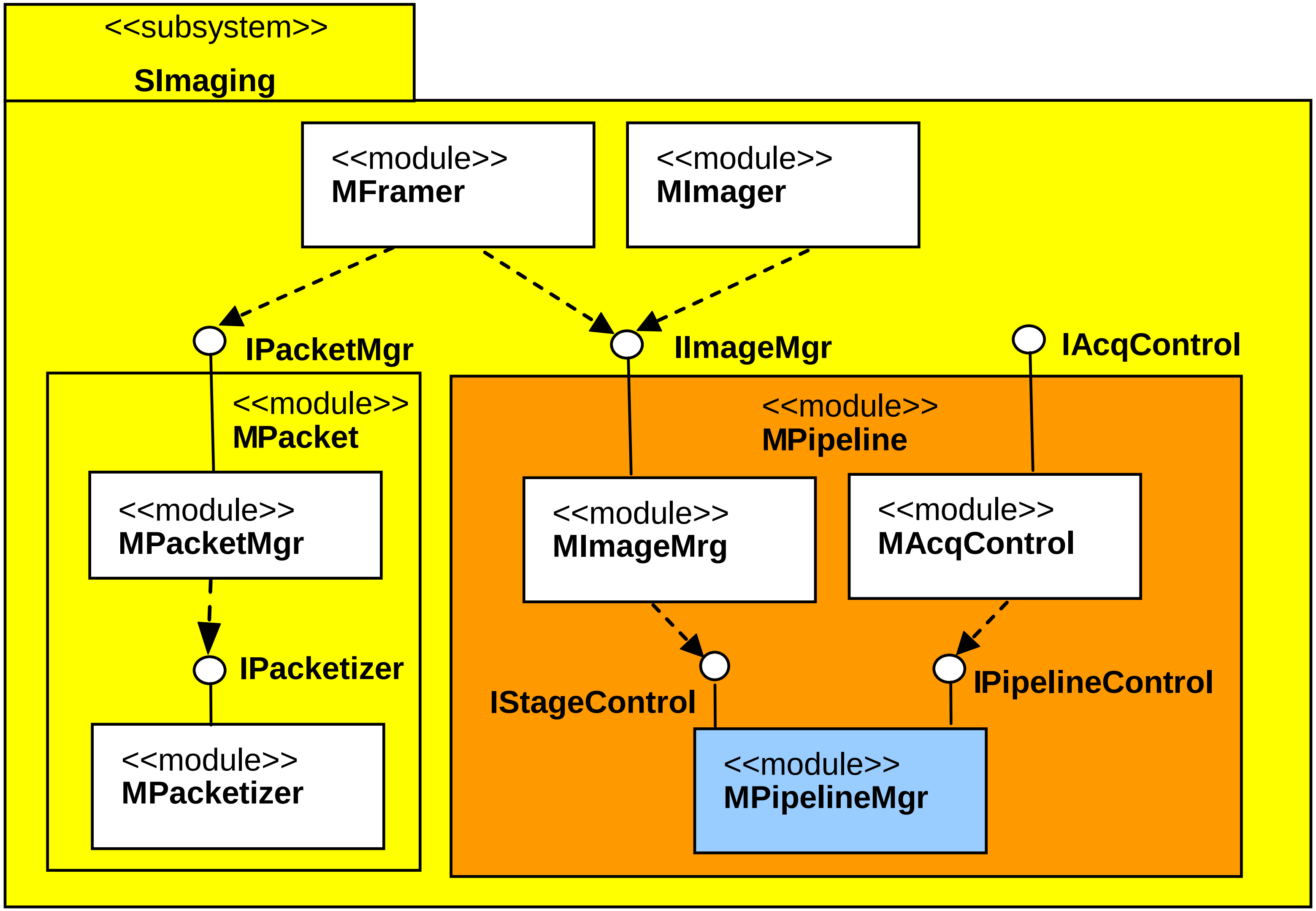


Figure 5.8. Imaging subsystem use-dependencies

Execution View - Elements

Element	UML Element	New Stereotype	Notation	Attributes	Associated Behavior
Runtime entity	Process	—	<<...>>	Host type, replication, resource allocation	—
	Thread	—			
	Class or active class	<<shared data>>, <<task>>, etc.			
Communication path	Association	—	———	—	Communication protocol

Execution View - Relationships

Relation	UML Element	Notation	Description
use mechanism	Association name	Name of communication mechanism; for example, IPC, RPC	A communication path uses a communication mechanism.
communicate over	—	(connection of class and association)	A runtime entity (or the module assigned to it) communicates over a communication path.
assigned to	Composition	Nesting (or)	A module is assigned to zero or more runtime entities.

Execution View - Artifacts

Artifact	Representation
Execution configuration	UML Class Diagram
Execution configuration mapped to hardware devices	UML Deployment Diagram
Dynamic behavior of configuration, or transition between configurations	UML Sequence Diagram
Description of runtime entities (including host type, replication, and assigned modules)	Table or UML Class Diagram
Communication protocol	Natural language description, or UML Sequence Diagram or Statechart Diagram

IPC = interprocess communication; RPC = remote procedure call; UML = Unified Modeling Language.

Figure 6.13 Final Execution configuration for the imaging subsystem of IS2000

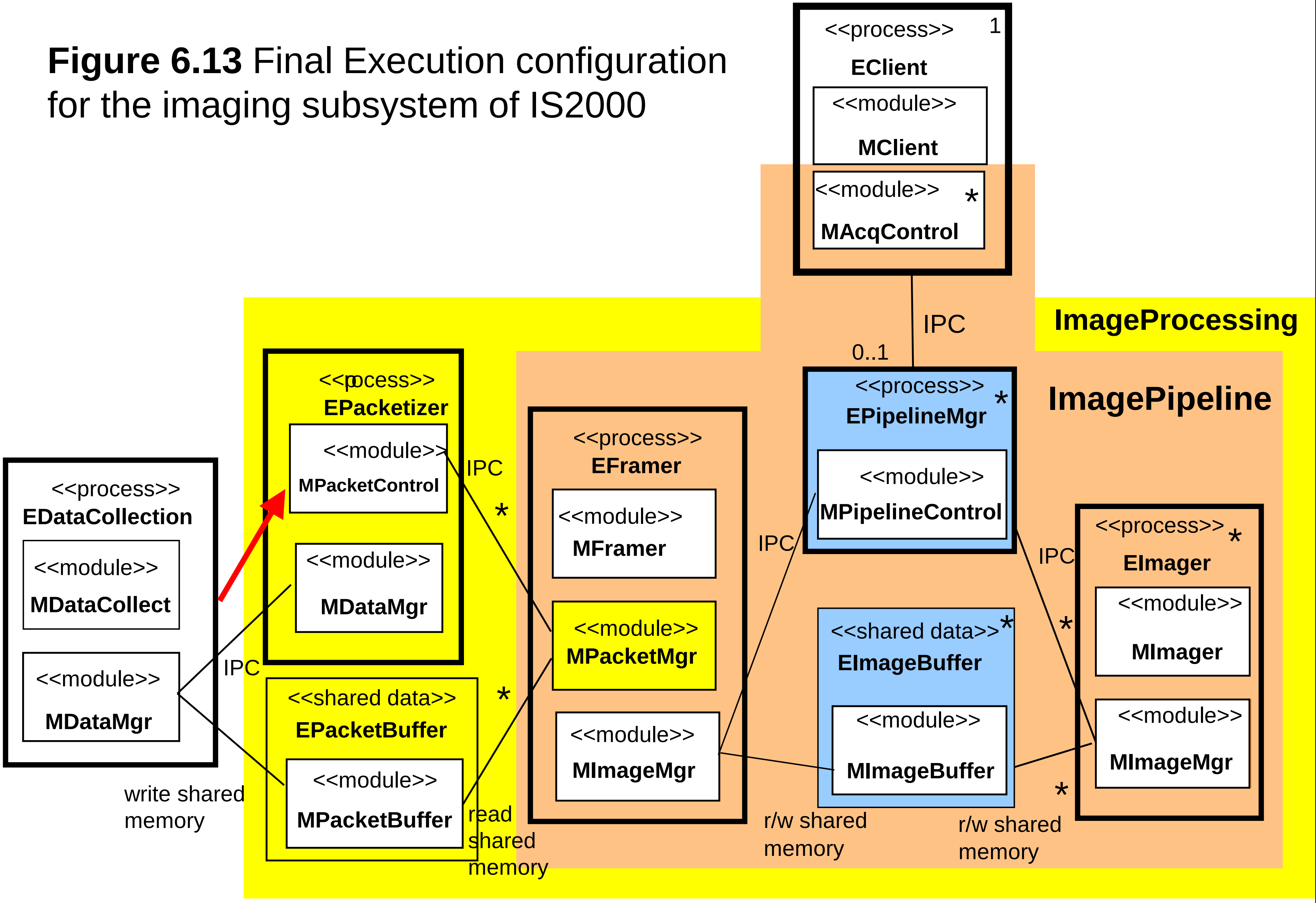
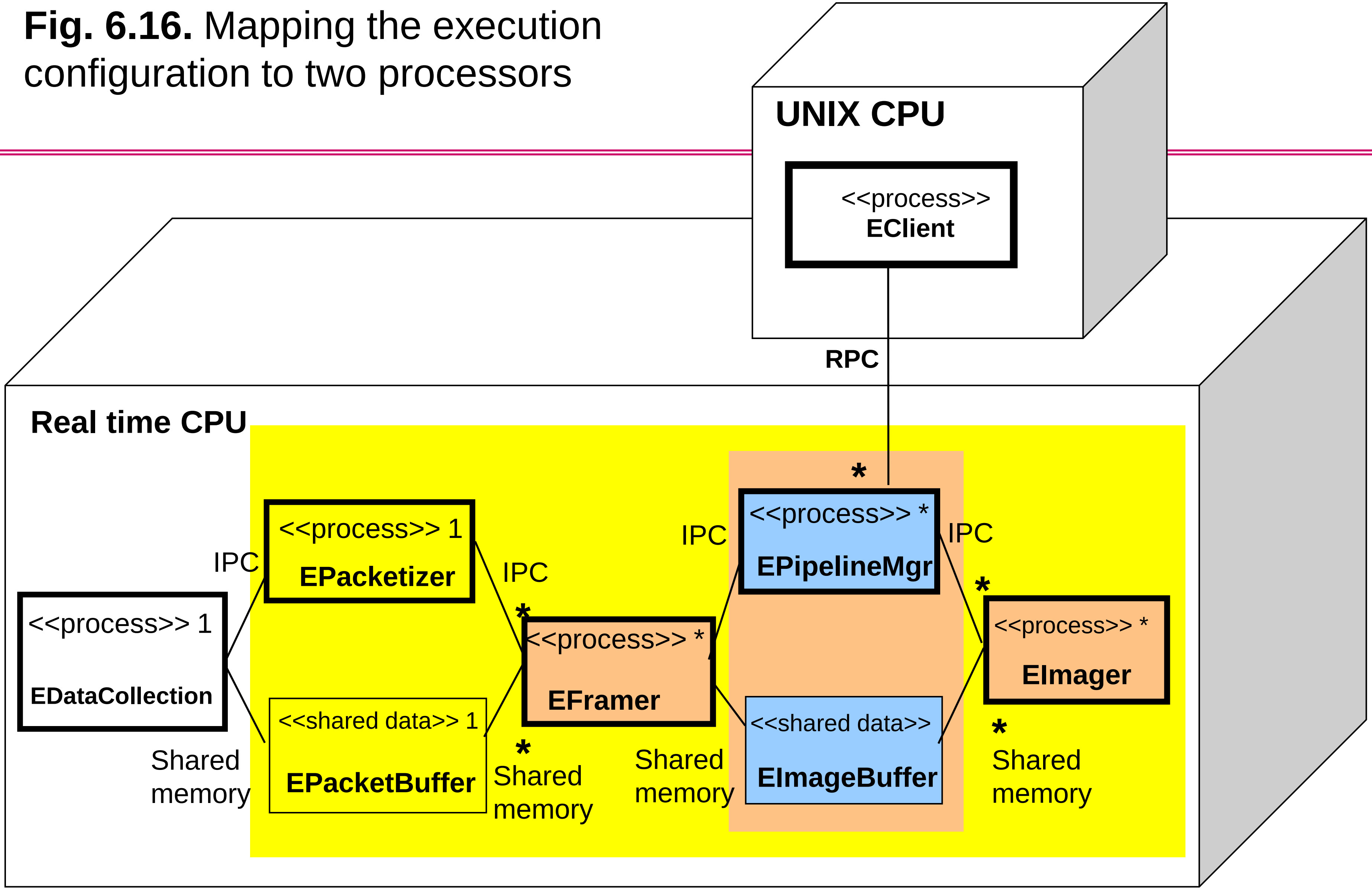
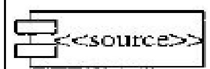


Fig. 6.16. Mapping the execution configuration to two processors



Code View - Elements

Element	UML Element	New Stereotype	Notation
Source component	Component	<<source>>	 Examples of source components are .H and .CPP files for C++.
Binary component	Component	<<binary>>	 These are intermediate components.
Library	Library	—	 Static or program libraries are intermediate components. Dynamic libraries are deployment components.
Executable	Executable	—	 These are deployment components.
Configuration description	Component	<<configuration description>>	 Describes execution configurations as well as resources.
Code group	Package	—	

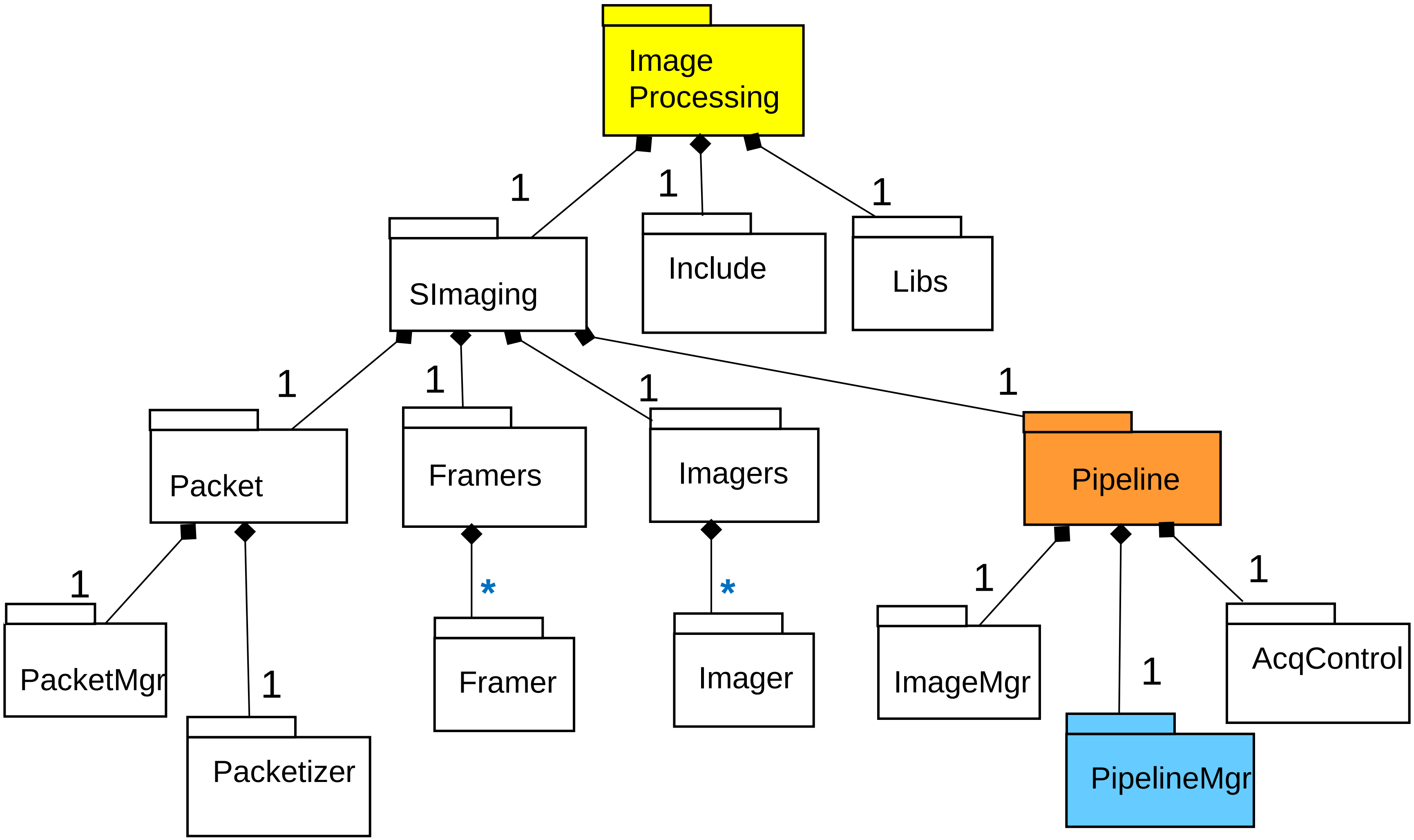
Code View - Relationships

Relation	UML Element	Notation	Description
generate	Dependency	- -generate- - ->	Source components can generate other source components.
import	Import	- -import- ->	Source components can import other source components.
compile	Dependency	- -compile- ->	Source components are compiled to binary components or libraries.
link	Dependency	- - link - ->	Binary components link statically to form libraries or executables. Dynamic or shared libraries link dynamically with and are loaded into executables.
use at runtime	Usage	use at runtime - - - - ->	An executable uses configuration descriptions at runtime.
trace	Trace	————	A code group may trace to a subsystem or layer. A source component may trace to a module or interface.
instantiate	Instantiate	Table row	At runtime, an executable instantiates a runtime entity (as a runtime instance).

Code View - Artifacts

Artifact	Representation
Module view, source component correspondence	Trace dependency, tables
Runtime entity, executable correspondence	Instantiation dependency, tables
Description of components in code architecture view, their organization, and their dependencies	UML Component Diagrams or tables
Description of build procedures	Tool-specific representations (for example, makefiles)
Description of release schedules for modules and corresponding component versions	Tables
Configuration management views for developers	Tool-specific representation

Figure 7.7 Organizing source components for the SImaging subsystem



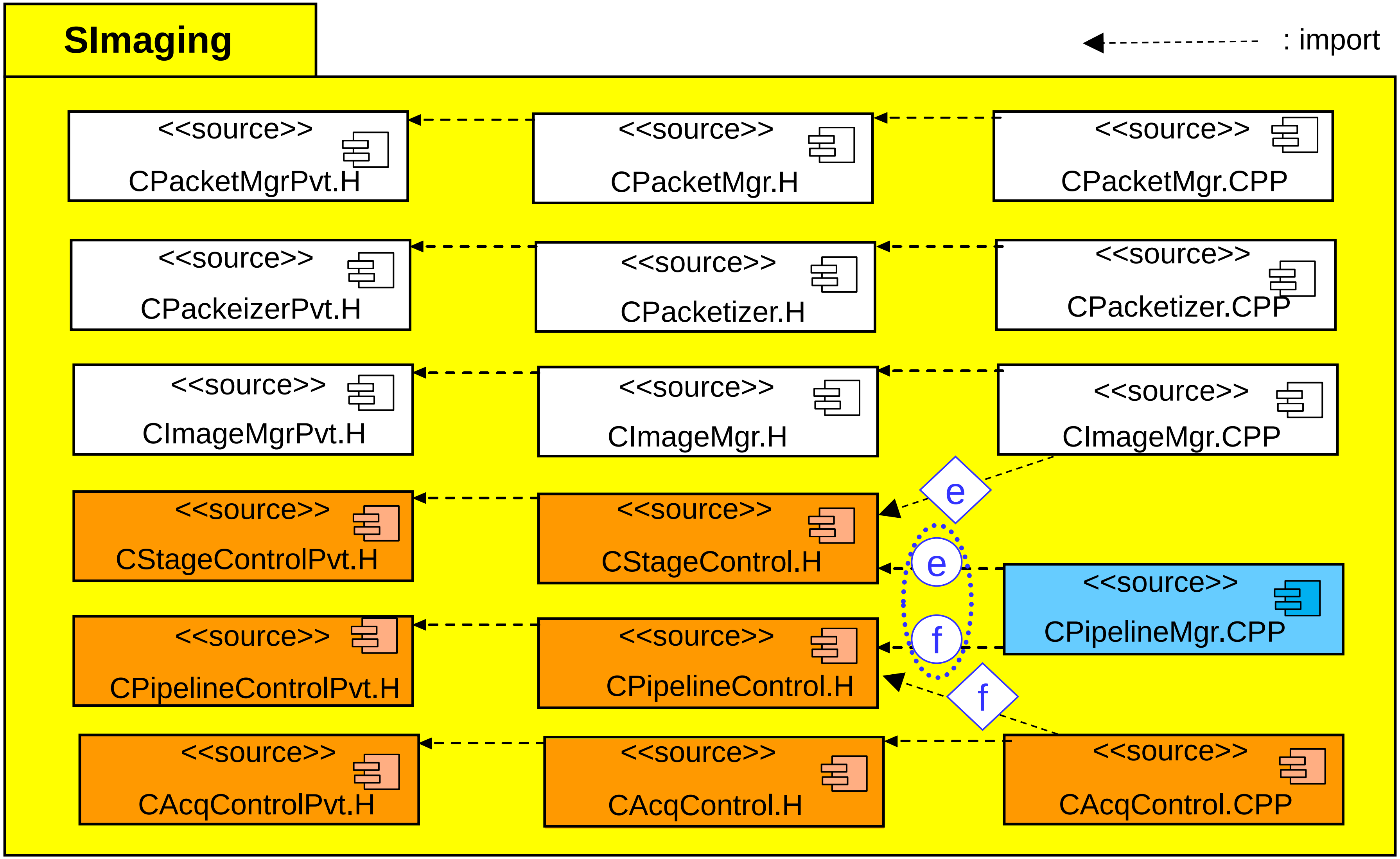


Figure 7.5. Source components and dependencies for the image-processing system

Questions?