Chapter 7: Component Diagrams, Deployment Diagrams, and Composite Structure Diagrams

Soumia Layachi

December 1, 2025

1. Introduction

Component diagrams and deployment diagrams are static view diagrams in UML. Component diagrams describe a system in terms of reusable components and the dependency relationships among them. Deployment diagrams, on the other hand, are closer to physical reality: they identify hardware elements such as PCs, servers, modems, or workstations, their physical connections, and the allocation of executables (represented as components) to these hardware resources.

A composite structure diagram serves a similar purpose to a class diagram, but it provides a more detailed view of the internal structure of multiple classes and their interactions.

2. Component Diagrams

2.1. Why Components?

Among the many factors contributing to software quality, reusability plays a crucial role. Reusability refers to the ability of software to be reused, either fully or partially, in new applications. However, the class concept in object-oriented programming—due to its fine granularity and fixed associations—does not provide an optimal solution for large-scale software reuse.

To overcome this limitation, concepts such as application templates and frameworks emerged in the 1990s, later evolving into a more general and unified notion: the software component.

Component-based development represents a major technological evolution, supported by platforms such as EJB, CORBA, .NET, and WSDL. This paradigm emphasizes the reuse of components and ensures that each component can evolve independently of the applications that employ it.

2.2. Definition of a Component

- A component is an autonomous unit represented by a structured, stereotyped "component" symbol. It includes one or more required and/or provided interfaces.
- Its internal behavior—usually implemented by a set of classes—is entirely hidden. Only its interfaces are visible. Two components can be interchanged as long as they conform to the same required and provided interfaces. Plugins and libraries are examples of components.
- The concept of a component is analogous to that of an object in terms of modularity and reusability, although the granularity differs. A component serves software architecture as an object serves code architecture.
- A component may evolve independently of the applications or other components that depend on it, provided that its interfaces remain consistent.

A component can be viewed in two complementary ways:

- **Black-box view:** The internal implementation is hidden; only the interfaces are visible.
- **White-box view:** The internal structure is shown, including the constituent objects and their relationships.

2.3. Graphical Representation of a Component

Components can be represented in several ways, as illustrated in Figure 1.

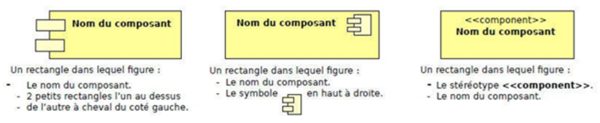
Nom du composant
Un rectangle dans lequel figure :
- Le nom du composant.
- 2 petits rectangles l'un au dessus
- de l'autre à cheval du coté gauche.
Nom du composant
Un rectangle dans lequel figure :
- Le nom du composant.
- Le symbole en haut à droite.
<<component>>
Nom du composant
Un rectangle dans lequel figure :
- Le stéréotype <<component>>.
- Le nom du composant.

Figure 1: Various graphical representations of a component.

2.4. Interfaces and Their Graphical Representation

There are two main types of interfaces:

- **Required interfaces:** These represent services that the component needs in order to function.

- **Provided interfaces:** These represent services that the component offers to other components.

Interfaces can be represented in several ways:

1. **Integrated within the component symbol:** Interfaces are listed in compartments using the stereotypes «`required interface`» and «`provided interface`».

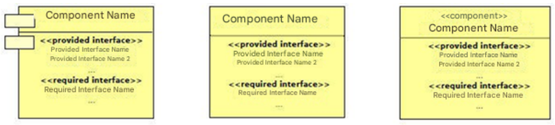

Figure 2: Interface representation integrated into the component.

2. **In a separate diagram element:**

   - Required interfaces are linked to the component using a dotted arrow labeled «`use`».
   - Provided interfaces are linked using a dotted arrow labeled «`realize`», with an open triangle at the arrow's end.

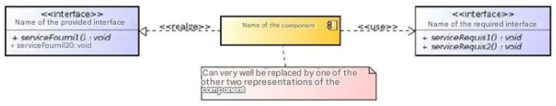
<<interface>>
Name of the provided interface
+ serviceFournil1() : void
+ serviceFournil20: void
<<realize>>
Name of the component
<<use>>
<<interface>>
Name of the required interface
+ serviceRequis1() : void
+ serviceRequis2() : void
Can very well be replaced by one of the other two representations of the component

Figure 3: Representation of interfaces in separate components.

3. **Using assembly connectors:**

   - Required interfaces are shown as semicircles.
   - Provided interfaces are shown as circles.

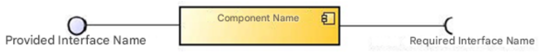

Figure 4: Interface representation with assembly connectors.

2.5. Ports

A port represents a connection point between a component and its external environment. Graphically, a port appears as a small square on the component's boundary, often labeled with its name. It serves as the physical manifestation of an interface.

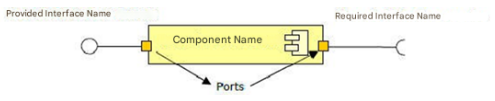

Figure 5: Representation of connection ports.

3. Deployment Diagrams

3.1. Purpose

A deployment diagram depicts the physical architecture of a system. It shows how software components are distributed across hardware nodes and defines the communication paths between them. Each hardware resource is modeled as a node, and the diagram specifies the nature of the connections (e.g., Ethernet, USB, or serial links).

3.2. Elements of a Deployment Diagram

3.2.1. Nodes

A node represents a hardware resource in the system. Typically, a node has memory and, in many cases, processing power. Human resources or peripheral devices may also be modeled as nodes. Hardware nodes can be labeled with the stereotype `«device»`.

Nodes are represented as rectangular parallelepipeds labeled with their names.

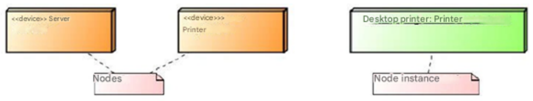

Figure 6: Representation of a node.

Nodes may also include attributes such as memory size, processor speed, or hardware type.

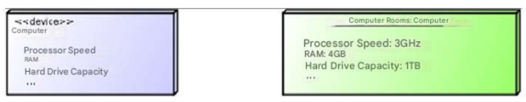

Figure 7: Node with specified attributes.

To indicate that a component is deployed on a node, one can either:

- Place the component within the node symbol, or
- Connect it with a dependency arrow labeled `«support»`.

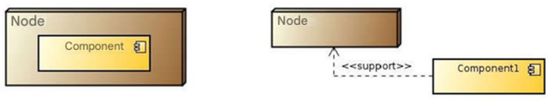

Figure 8: Representation of a component deployed on a node.

3.2.2. Communication Paths

A communication path models the connection between two nodes (e.g., Ethernet, USB, serial link). The link can include:

- Cardinalities,
- Constraints in curly braces (e.g., {secure access}), or
- Network type and speed specified as a stereotype.

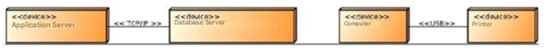

Figure 9: Representation of links between nodes.

3.2.3. Artifacts

An artifact is a tangible element such as a document, executable, file, database table, or script. It is represented by a rectangle labeled with the keyword «artifact» followed by its name.

A manifestation represents the relationship between a model element and the artifact that implements it. It is shown by a dotted dependency arrow labeled «manifest».

An instance of an artifact is deployed onto a node instance, represented by a dependency arrow labeled «deploy». Importantly, it is artifacts—not components—that are deployed onto nodes.

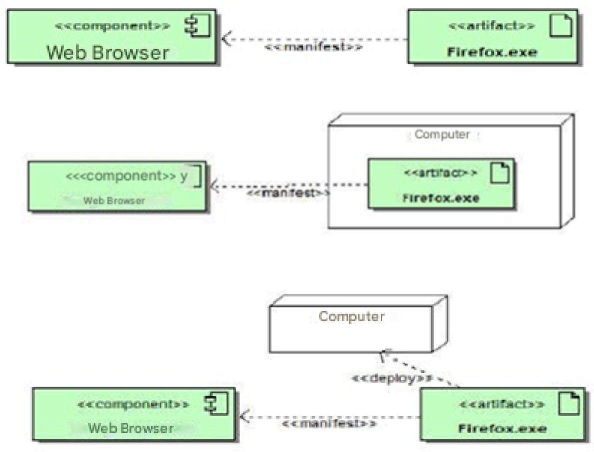

Figure 10: Examples of artifacts placed inside or outside nodes and their links with components.

4. Composite Structure Diagrams

4.1. Definition

A composite structure diagram is a UML diagram that, like a class diagram, describes system structure. However, it provides a deeper view of the internal organization of classes and the way their instances collaborate through communication links to achieve shared objectives.

Composite structure diagrams illustrate:

- The internal structure of a classifier,
- Interactions with the environment through ports, and
- Collaborative behaviors among parts.

4.2. Representation

Composite structure diagrams complement, rather than replace, class diagrams. In a composite structure diagram:

- The composite object is described by a classifier.
- Its components are represented as parts. Components are shown within the classifier symbol. Their multiplicity is indicated in brackets.
- Aggregated components are represented with dashed lines.
- Strongly composed components use solid lines.

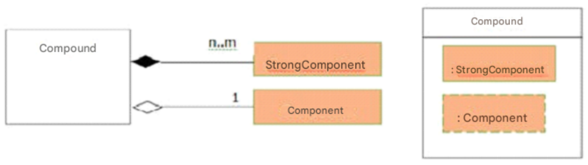

Figure 11: Representation of a composite structure diagram

Example: Consider a class diagram representing an automobile as a composite object. It includes linked associations between wheels and half-shafts, which transmit power from the engine to the front (driving) wheels. The cardinality of this association is 0..1—one for the front wheels and zero for the rear. This relationship cannot be expressed in a simple class diagram without introducing subclasses (e.g., *FrontWheel* and *RearWheel*), which would unnecessarily complicate the model.

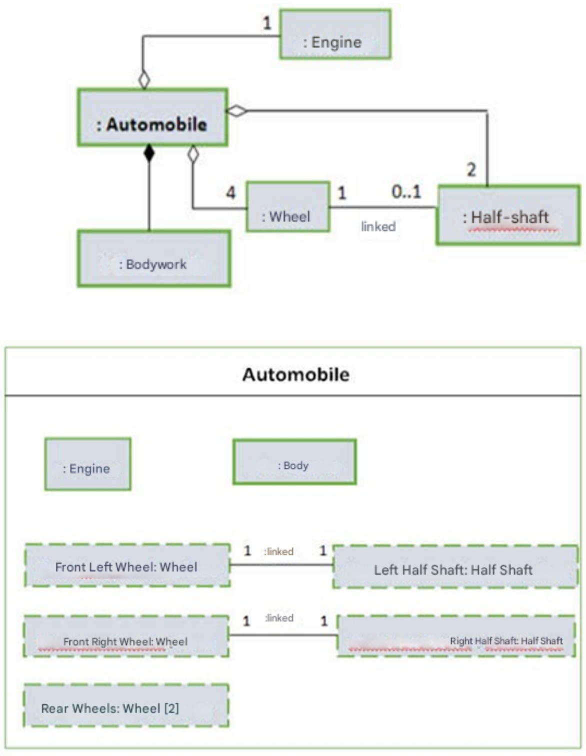

Figure 12: Composite figures illustrating an automobile as a composite object with internal structure.

References

1. Roques, P. (2009). *UML 2 in Practice: Case Studies and Corrected Exercises.* Eyrolles.

2. Charroux, B., Osmani, A., & Thierry, Y. (2010). *UML 2 Practical Modeling.* Pearson.

3. Vallée, F. (2005). *UML for Decision-Makers.* Paris: Éditions Eyrolles.

4. Chantal, M., Hugues, J., & Leblanc, B. (2000). *UML for the Analysis of an Information System.* Dunod.

5. Debrauwer, L., & Van der Heyde, F. (2008). *UML 2: Initiation, Examples, and Corrected Exercises.* Éditions ENI.