uc3m | Universidad Carlos III de Madrid

Logical and Physical Views-

Component and deployment Diagrams

Software Engineering, Course 2022-2023

Building on the ongoing example about Spotify:

1. Identify components and main operations (interface).
2. Establish dependencies between components (required vs offered interface).
3. Establish a deployment model.
4. Create a component diagram in UML2.
5. Create a physical deployment diagram in UML2.

Annex I: Component Diagram description

Component-based development (CBD) and object-oriented development go hand-in-hand, and it is generally recognized that object technology is the preferred foundation from which to build components. I typically use UML 2 component diagrams as an architecture-level artifact, either to model the business software architecture, the technical software architecture, or more often than not both of these architectural aspects. Physical architecture issues, in particular hardware issues, are better addressed via UML deployment diagrams or network diagrams. In fact I'll often iterate back and forth between these diagrams.

Component diagrams are particularly useful with larger teams. Your initial architectural modeling efforts during cycle 0 should focus on identifying the initial architectural landscape for your system. UML component diagrams are great for doing this as they enable you to model the high-level software components, and more importantly the interfaces to those components. Once the interfaces are defined, and agreed to by your team, it makes it much easier to organize the development effort between subteams. You will discover the need to evolve the interfaces to reflect new requirements or changes to your design as your project progresses, changes that need to be negotiated between the subteams and then implemented appropriately.

Interfaces and Ports

Components may both provide and require interfaces. An interface is the definition of a collection of one or more methods, and zero or more attributes, ideally one that defines a cohesive set of behaviors. A provided interface is modeled using the lollipop notation and a required interface is modeled using the socket notation. A port is a feature of a classifier that specifies a distinct interaction point between the classifier and its environment. Ports are depicted as small squares on the sides of classifiers.

Table 1. Component design principles.

Principle	Description
Acyclic Dependencies	Allow no cycles in the dependencies graph between components. For example disallow A è B è C è A because it includes a cycle.
Common Closure	The classes of a component should be closed together against the same kinds of changes. A change that affects a class within a component should not affect classes outside that component. In other words your components should be cohesive in that sweeping changes across several components are not required.
Common Reuse	The classes in a component are reused together. If you reuse one class in a component you reuse them all. This is another principle addressing cohesion.
Dependency Inversion	Abstractions should not depend on details, instead details should depend on abstractions.
Open-Closed	Software elements should be open for extension but closed for modification.
Release-Reuse Equivalency	The granule of reuse is the granule of release. In other words you should not reuse only part of a released software element.
Stable Abstractions	A component should be as abstract as it is stable. A component should be sufficiently abstract so that it can be extended without affecting its stability.
Stable Dependencies	Depend on the direction of stability - If component A depends on component B, then B should be more stable (e.g. less likely to change) than A.

Learn more: http://agilemodeling.com/artifacts/componentDiagram.htm

11.6 Components

11.6.1 Summary

This sub clause specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, it 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.

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.

11.6.2 Abstract Syntax

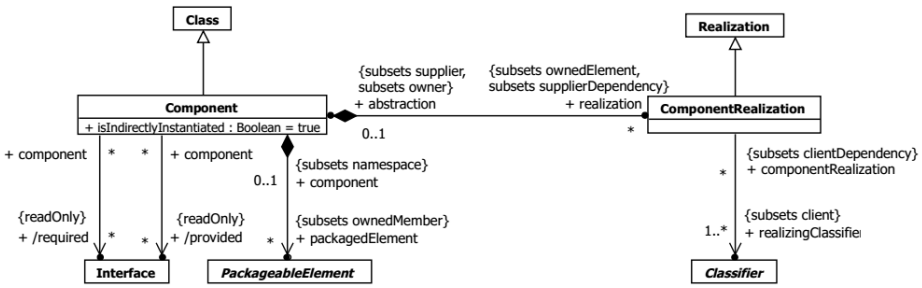

Figure 11.38 Components

11.6.3 Semantics

11.6.3.1 Components

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

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 is fully compatible with 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. 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 them together.

A Component is modeled throughout the development life cycle and successively refined into deployment and run-time. A Component may be manifested by one or more Artifacts, and in turn, that Artifact may be deployed to its execution environment. A DeploymentSpecification may define values that parameterize the Component's execution. (See Deployments – Clause 19).

The required and provided Interfaces of a Component allow for the specification of StructuralFeatures such as attributes and Association ends, as well as BehavioralFeatures such as Operations and Receptions. A Component may implement a provided Interface directly, or its realizing Classifiers may do so, or they may be inherited. 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 ProtocolStateMachine 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.

The wiring between Components in a system or other context can be structurally defined by using Dependencies between compatible simple Ports, or between Usages and matching InterfaceRealizations that are represented by sockets and lollipops (see 10.4.4) on Components on Component diagrams. Creating a wiring Dependency between a Usage and a matching InterfaceRealization, or between compatible simple Ports, means that there may be some additional information, such as performance requirements, transport bindings, or other policies that determine that the Interface is realized in a way that is suitable for consumption by the depending Component. Such additional information could be captured in a profile by means of stereotypes.

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. Dependencies on the external view provide a convenient overview of what may happen in the internal view; they do not prescribe what must happen. More detailed behavior specifications such as Interactions and Activities may be used to detail the mapping from external to internal behavior.

The execution time semantics for an assembly Connector in a Component are that requests (signals and operation invocations) travel along an instance of a Connector. The execution semantics for multiple Connectors directed to and from different roles, or n-ary Connectors where n> 2, indicates that the instance that will originate or handle the request will be determined at execution time.

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 Standard Profiles).

A Component may be realized (or implemented) by a number of Classifiers. In that case, a Component owns a set of ComponentRealizations to these Classifiers.

A component acts like a Package for all model elements that are involved in or related to its definition, which should be either owned or imported explicitly. Typically the Classifiers that realize a Component are owned by it.

The isDirectlyInstantiated property specifies 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.

11.6.4 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. If the icon symbol is shown, the keyword «component» may be hidden.

The attributes, operations and internal structure compartments all have their normal meaning. The internal structure uses the notation defined in StructuredClassifiers (11.2).

The provided and required Interfaces of a Component may be shown by means of ball (lollipop) and socket notation (see 10.4.4), where the lollipops and sockets stick out of the Component rectangle.

For displaying the full signature of a provided or required Interface of a Component, the Interfaces can also be displayed as normal expandable Classifier rectangles. For this option, the Interface rectangles are connected to the Component rectangle by appropriate dependency arrows, as specified in 7.7.4 and 10.4.4.

A conforming tool may optionally support compartments named “provided interfaces” and “required interfaces” listing the provided and required Interfaces by name. This may be a useful option in scenarios in which a Component has a large number of provided or required Interfaces.

Additional optional compartments “realizations” and “artifacts” may be used to list the realizing Classifiers (Classifiers reached by following the realization property) and manifesting Artifacts (Artifacts that manifest this component – see 19.3).

A ComponentRealization is notated in the same way as a Realization dependency (i.e., as a general dashed line with a hollow triangle as an arrowhead).

The packagedElements of a Component may be displayed in an optional compartment named “packaged elements,” according to the specification for optional compartments for ownedMembers set out in 9.2.4.

11.6.5 Examples

An overview diagram can show Components related by Dependencies, which signify some further unspecified kind of dependency between the components, and by implication a lack of dependency where there are no Dependency arrows.

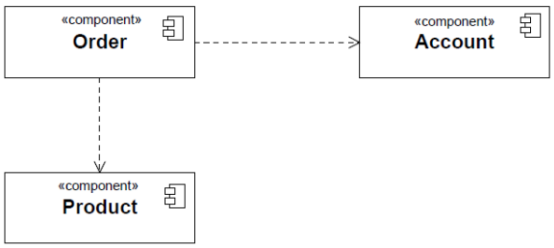

Figure 11.39 Example of an overview diagram showing Components and their general Dependencies

Figure 11.40 shows an external (“black-box”) view of a Component by means of interface lollipops and sockets sticking out of the Component rectangle.

Figure 11.40 A Component with two provided and three required Interfaces

Figure 11.41 shows provided and required interfaces listed in optional compartments.

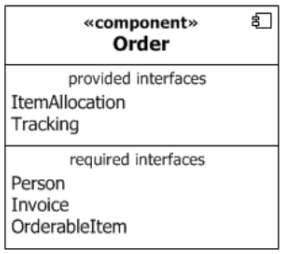

Figure 11.41 Black box notation showing a listing of provided and required interfaces

Figure 11.42 shows a "white box" view of a Component listing realizing Classifiers and manifesting Artifacts in additional optional compartments.

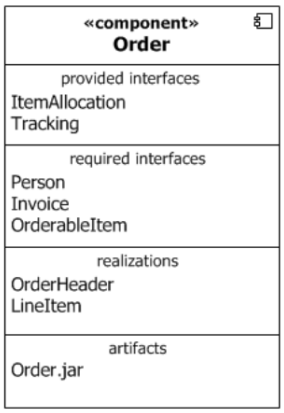

Figure 11.42 Optional "white-box" representation of a Component

Figure 11.43 shows explicit representation of the provided and required Interfaces using Dependency notations, allowing Interface details such as Operations to be displayed.

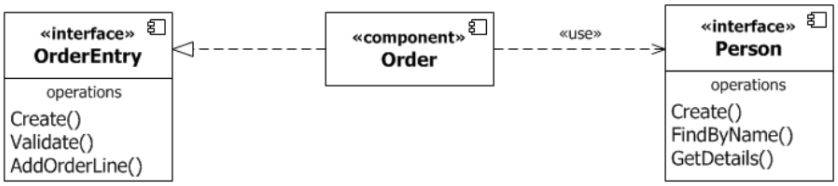

Figure 11.43 Explicit representation of provided and required Interfaces using Dependency notation.

Figure 11.44 shows a set of Classifiers that realize a Component with realization arrows representing the ComponentRealizations.

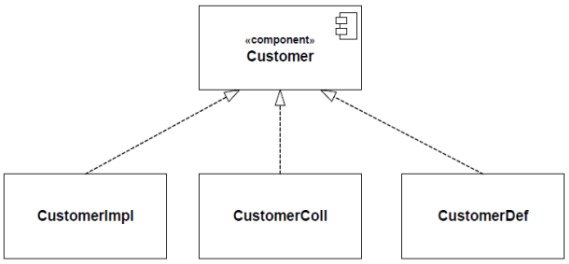

Figure 11.44 A representation of the realization of a complex Component

Figure 11.45 shows owned Classes that realize a Component nested within an optional "packaged elements" compartment of the Component shape.

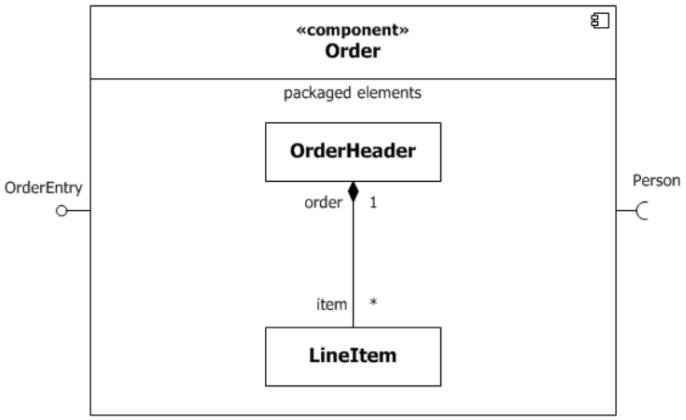

Figure 11.45 An alternative nested representation of a complex Component

Figure 11.46 shows various ways of wiring Components using Dependencies.

The Dependency on the right of the figure is from the Usage of OrderableItem to the InterfaceRealization of OrderableItem. This also shows that "/OrderableItem" is an Interface that is implemented by a supertype of Product, following the notation specified in 10.4.4.

The Dependency between the AccountPayable Ports illustrates the notational option of showing the dependency arrow joining the socket to the lollipop, when a Dependency is wired between simple Ports.

When realizing Classifiers are shown in a packaged elements compartment, a Dependency may be shown from a simple Port to a realizing Classifier to indicate that the Interface provided or required by the Port is dependent in some way upon the Classifier. This is illustrated by the Dependency from AccountPayable to OrderHeader, which indicates that something about the fact that the Component requires AccountPayable is dependent upon OrderHeader.

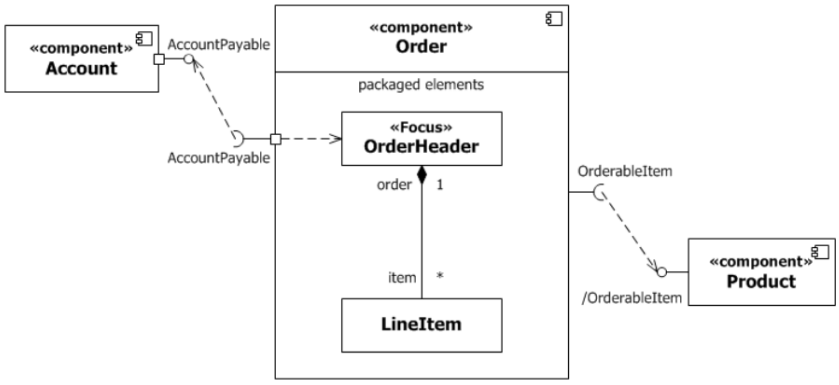

Figure 11.46 Example model of a Component, its provided and required Interfaces, and wiring through Dependencies

Figure 11.47 shows an internal or white-box view of the internal structure of a Component that contains other Components with simple Ports as parts of its internal assembly. The assembly Connectors use ball-and-socket notation. The delegation connectors use the notational option that the Connector line can end on the ball or socket, rather than the simple port itself.

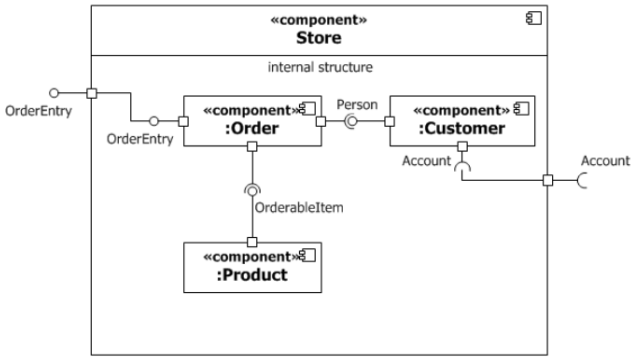

Figure 11.47 Internal structure of a Component

Figure 11.48 shows delegation Connectors from delegating Ports to handling parts; in this example the parts in the internal structure compartment are typed by Classes shown in the optional packaged elements compartment.

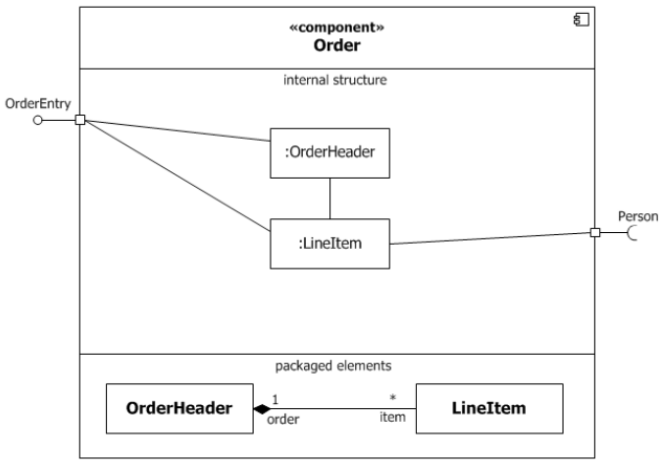

Figure 11.48 Delegation Connectors connect externally provided Interfaces to the parts that realize or require them.

11.7 Collaborations

Annex II: Deployment Diagram

A UML 2 deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes. In other words, deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another. You want to create a deployment diagram for applications that are deployed to several machines, for example a point-of-sales application running on a thin-client network computer which interacts with several internal servers behind your corporate firewall or a customer service system deployed using a web services architecture such as Microsoft's .NET. Deployment diagrams can also be created to explore the architecture of embedded systems, showing how the hardware and software components work together. In short, you may want to consider creating a deployment diagram for all but the most trivial of systems.

To determine whether you need to create a deployment model, ask yourself this: if you knew nothing about the system and someone asked you to install it and/or maintain and support it, would you want a description of how the parts of the system fit together? When I ask this question of the project teams I work with, we almost always decide to develop some form of deployment model. More important, practice has shown that deployment modeling is well worth it. Deployment models force you to think about important deployment issues long before you must deliver the actual system.

When determining how to model the deployment architecture for a system, regardless of the artifacts chosen, I will typically:

1. Identify the scope of the model. Does the diagram address how to deploy a version of a single application or does it depict the deployment of all systems within your organization?
2. Consider fundamental technical issues. What existing systems will yours need to interact/integrate with? How robust does your system need to be (will there be redundant hardware to failover to)? What/who will need to connect to and/or interact with your system and how will they do it (via the Internet, exchanging data files, and so forth)? What middleware, including the operating system and communications approaches/protocols, will your system use? What hardware and/or software will your users directly interact with (PCs, network computers, browsers, and so forth)? How do you intend to monitor the system once it has been deployed? How secure does the system need to be (do you need a firewall, do you need to physically secure hardware, and so forth)?
3. Identify the distribution architecture. Do you intend to take a fat-client approach where the business logic is contained in a desktop application or a thin-client approach where business logic is deployed to an application server? Will your application have two tiers, three tiers, or more? Your distribution architecture strategy will often be predetermined for your application, particularly if you are deploying your system to an existing technical environment.
4. Identify the nodes and their connections. Your distribution strategy will define the general type of nodes you will have, but not the exact details. You need to make platform decisions, such as the hardware and operating systems to be deployed, including how the various nodes will be connected (perhaps via RMI and a message bus).
5. Distribute software to nodes. Both versions of the deployment diagrams indicate the software that is deployed on each node, critical information for anyone involved in development, installation, or operation of the system.

Learn more: http://agilemodeling.com/artifacts/deploymentDiagram.htm

19 Deployments

19.1 Summary

The Deployments package specifies constructs that can be used to define the execution architecture of systems and the assignment of software artifacts to system elements. A streamlined model of deployment, sufficient for the majority of modern applications, is provided. Where more elaborate deployment models are required, the package can be extended through profiles or metamodels to represent specific hardware and/or software environments.

19.2 Deployments

19.2.1 Summary

Deployments capture relationships between logical and/or physical elements of systems and information technology assets assigned to them.

19.2.2 Abstract Syntax

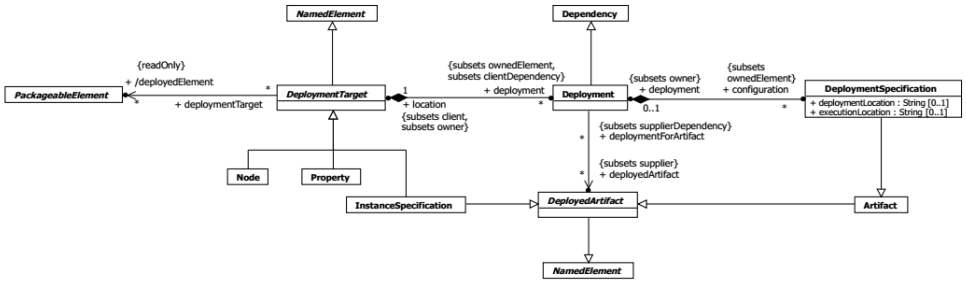

Figure 19.1 Deployments

19.2.3 Semantics

A Deployment captures the relationship between a particular conceptual or physical element of a modeled system and the information assets assigned to it. System elements are represented as DeployedTargets, and information assets, as DeployedArtifacts. DeploymentTargets and DeploymentArtifacts are abstract classes that cannot be directly instantiated. They are, however, elaborated as concrete classes as described in the Artifacts and Nodes sub clauses that follow.

Individual Deployment relationships can be tailored for specific uses by adding DeploymentSpecifications containing configurational and/or parametric information and may be extended in specific component profiles. For example, standard, non-normative stereotypes that a profile might add to DeploymentSpecification include «concurrencyMode», with tagged values {thread, process, none}, and «transactionMode», with tagged values {transaction, nestedTransaction, none}.

DeploymentSpecification information becomes execution-time input to components associated with DeploymentTargets via their deployedElements links. Using these links, DeploymentSpecifications can be targeted at specific container types, as long as the containers are kinds of Components. As shown in Figure 19.1, DeploymentSpecifications can be captured as elements of the Deployment relationships because they are Artifacts (described in the following sub clause). Furthermore, DeploymentSpecifications can only be associated with DeploymentTargets that are ExecutionEnvironments (described in the Nodes sub clause, below).

The Deployment relationship between a DeployedArtifact and a DeploymentTarget can be defined at the “type” level and at the “instance” level. At the “type” level, the Deployment connects kinds of DeploymentTargets to kinds of DeployedArtifacts. Whereas, at the “instance” level, the Deployment connects particular DeploymentTargets instances to particular DeployedArtifacts instances. For example, a ”type” level Deployment might connect an “application server” with an “order entry request handler.” In contrast, at the ”instance” level, three specific application services (say, “app-server1”, “app-server-2” and “app-server3”) may be the DeploymentTargets for six different “request handler” instances.

For modeling complex models consisting of composite structures, a Property, functioning as a part (i.e., owned by a composition), may be the target of a Deployment. Likewise, InstanceSpecifications can be DeploymentTargets in a Deployment relationship, if they represent a Node that functions as a part within an encompassing Node composition hierarchy, or if they represent an Artifact.

19.2.4 Notation

DeployedTargets are shown as a perspective view of cube labeled with the name of the DeployedTarget shown prepended by a colon. System elements deployed on a DeployedTarget, and Deployments that connect them, may be drawn inside the perspective cube. Alternately, deployed system elements can be shown as a textual list of element names.

Deployments are depicted using the same dashed line notation as Dependencies. When Deployment relationships are shown within a DeploymentTarget graphic, no labeling is required. Alternately, Deployment relationships can be decorated with the «deploy» keyword when not contained inside a DeployedTarget graphic. Deployment arrows are generally drawn from DeployedArtifacts to the DeployedTargets.

DeploymentSpecifications are graphically displayed as classifier rectangles and may be decorated with the «deployment spec» keyword. They may be attached to a component deployed on a container using a regular dependency arrow.

19.2.5 Examples

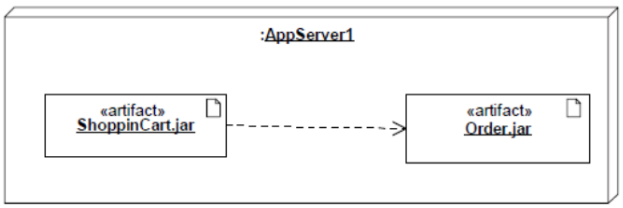

Figure 19.2 A visual representation of the deployment location of artifacts, including a dependency between them, inside a DeployedTarget graphic.

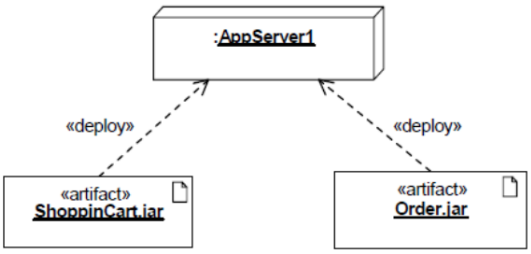

Figure 19.3 Alternative deployment representation of using a dependency called «deploy» used when DeployedArtifacts are visually outside their DeployedTarget graphics

Figure 19.4 Textual list based representation of DeployedArtifacts.

Figure 19.5 DeploymentSpecification for an artifact. On the left, a type-level specification, and on the right, an instance-level specification.

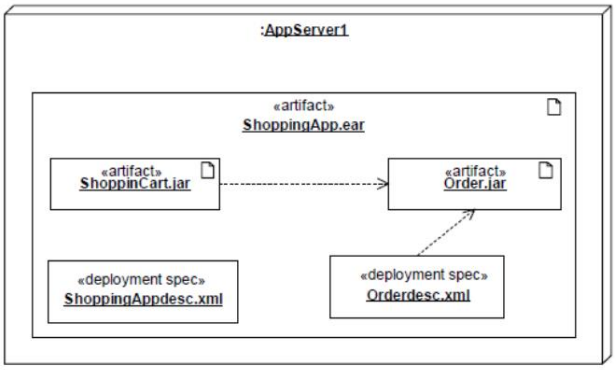

Figure 19.6 DeploymentSpecifications related to the DeployedArtifacts that they parameterize.

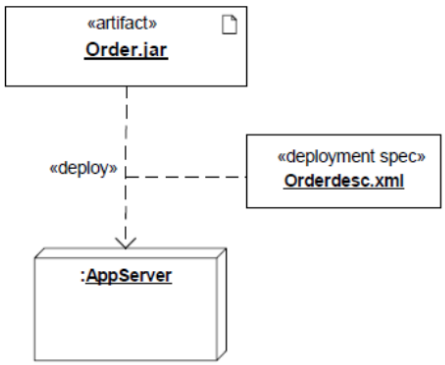

Figure 19.7 A DeploymentSpecification for a DeployedArtifact.