Date: April 2006

Deployment and Configuration of Component-based Distributed Applications Specification

OMG Available Specification

Version 4.0

formal/06-04-02

OBJECT MANAGEMENT GROUP

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

This specification defines metadata and interfaces to facilitate the deployment and configuration of component-based applications into heterogeneous distributed target systems.

The specification defines:

- Metadata to describe component-based applications and their requirements (“Platform Independent Model”) and interfaces to store, browse, and retrieve such metadata (Section 7.4, “Component Management Model,” on page 42).
- Metadata to describe heterogeneous distributed target systems and their capabilities (Section 7.5, “Target Data Model,” on page 44) and interfaces to collect and retrieve such metadata (Section 7.6, “Target Management Model,” on page 50).
- Metadata to describe a specific deployment of an application into a distributed target system (Section 7.7, “Execution Data Model,” on page 54) and interfaces to execute deployments (Section 7.8, “Execution Management Model,” on page 67).
- A deployment process which includes installation, configuration, planning, preparation, and launch of the distributed application.

2 Conformance

2.1 Summary of optional versus mandatory interfaces

All interfaces are mandatory within the compliance points. The interfaces are RepositoryManager, TargetManager, ExecutionManager, NodeManager, ApplicationManager, and Application.

2.2 Conformance Points

In general, the PIM suggests and enables several independent compliance points to enable different vendor implementations or user replacement of implementations. These are:

- RepositoryManager
  Rationale is that this function can be standalone, and implementations can offer a wide range of persistence, database, security, file system or web functionality.
- TargetManager
  Rationale is that this function can be standalone for independent offline planning or fully dynamic at runtime. Both could coexist.
- NodeManager
  Rationale is that this function is related to the node OS, ORB, development system etc., and there would likely be multiple vendors’ implementations in a given distributed system. it should be a modest effort for a node platform supplier to implement this without the rest of the deployment system.
- ExecutionManager
  This is the core of the deployment system.

The PSMs define their own specific compliance points. For the CCM PSM, all 4 are defined.

In Chapter 8, the UML Profile for D&C Tool Support, suggests a further set of conformance points for tools:

- Modeling Tools that can create a well formed conformant M0 model of the PIM for CCM
- Forward Engineering Tools that can generate well formed XML, based on the XML schema for the PSM for CCM, of conformant M0 models.

3 References

3.1 Normative References

[CCM] Object Management Group, "*CORBA Components*," version 3.0. Adopted specification. June 2002. Available from http://www.omg.org/cgi-bin/doc?formal/02-06-65

[HTTP] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, RFC 2616: "*Hypertext Transfer Protocol -- HTTP/1.1*." June 1999. Available from http://www.ietf.org/rfc/rfc2616.txt

[MOF1] Object Management Group, "*Meta Object Facility Specification*," version 1.4. Adopted specification. April 2002. Available from http://www.omg.org/cgi-bin/doc?formal/02-04-03

[UML1] Object Management Group, "*Unified Modeling Language Specification*," version 1.5. Adopted specification. March 2003. Available from http://www.omg.org/cgi-bin/doc?formal/03-03-01

[UPC] Object Management Group, "*UML™ Profile for CORBA™ Specification*," version 1.0. Adopted specification. April 2002. Available from http://www.omg.org/cgi-bin/doc?formal/02-04-01

[URI] T. Berners-Lee, R. Fielding, L. Masinter, RFC 2396: "*Uniform Resource Identifiers (URI): Generic Syntax*." August 1998. Available from http://www.ietf.org/rfc/rfc2396.txt

[URN] R. Moats, RFC 2141: "*URN Syntax*." May 1997. Available from http://www.ietf.org/rfc/rfc2141.txt

[XMI] Object Management Group, "*XML Metadata Interchange (XMI) Specification*," version 2.0. Adopted specification. May 2003. Available from http://www.omg.org/cgi-bin/doc?formal/03-05-02

[XML] World Wide Web Consortium (W3C), "*Extensible Markup Language (XML)*," version 1.0 (second edition). W3C Recommendation, October 6, 2000. Available from http://www.w3.org/TR/REC-xml

[XSD] World Wide Web Consortium (W3C), "*XML Schema Part 1: Structures*." W3C Recommendation, May 2, 2001. Available from http://www.w3.org/TR/xmlschema-1/

World Wide Web Consortium (W3C), "*XML Schema Part 2: Datatypes*." W3C Recommendation, May 2, 2001. Available from http://www.w3.org/TR/xmlschema-2/

[ZIP] Pkware, Inc. "*.ZIP File Format Specification*," version 5.2. June 2, 2003. Available from http://www.pkware.com/products/enterprise/white_papers/appnote.txt

3.2 Non-normative References

[MOF2] Adaptive, Ceira Technologies, Incl, Compuware Corporation, Data Access Technologies, Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA, MetaMatrix, Softeam, SUN, Telelogic AB, Unisys, "*Meta Object Facility (MOF) 2.0 Core Proposal*." Recommended for adoption. April 2003. Available from http://www.omg.org/cgi-bin/doc?ad/03-04-07

[UML21] U2 Partners, "*Unified Modeling Language: Infrastructure*," Recommended for adoption. January 2003. Available from http://www.omg.org/cgi-bin/doc?ad/03-01-01

[UML2S] U2 Partners, "*Unified Modeling Language: Superstructure*," version 2.0. Recommended for adoption. April 2003. Available from http://www.omg.org/cgi-bin/doc?ad/03-04-01

[UMLQOS] I-Logix Inc., Open-IT, THALES, "*UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*." Revised submission. May 4, 2003. Available from http://www.omg.org/cgi-bin/doc?realtime/03-05-02

4 Terms and Definitions

Terms marked with a reference in square brackets, e.g., **[UML2S]**, are copied verbatim from the referenced specification. They are compiled here to provide a concise source of all relevant definitions.

4.1 Artifact

A physical piece of information that is used or produced by a deployment process. Examples of artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component.

4.2 Bridge

A resource that provides connectivity between interconnects, supplying an indirect communication path between nodes.

4.3 Capability

A feature offered by a component implementation.

4.4 Component

A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).

4.5 Component Assembly

An implementation of a specific component interface using a set of interconnected components and a mapping of the implemented component interface's features to these subcomponents.

4.6 Component Implementation

An abstract class that contains the attributes and associations that are common to both a “Monolithic Implementation” and a “Component Assembly”.

4.7 Component Interface

A named set of provided and required interfaces that characterize the behavior of a component.

4.8 Component Package

A set of alternative implementations of a component interface contained in a set of artifacts and compiled code modules. (Has a set of component implementations, and each of these implementations is equally valid.)

4.9 Configuration

A set of default run-time application options used to customize non-deployment related application features. See Section 6.3.1, “Preconditions for the Process of Deployment,” on page 8 for more information.

4.10 Connection

A connection is either a communication path among the ports of two or more subcomponents allowing them to communicate with each other, or it is a communication path between an assembly’s external ports and an assembly’s subcomponents that delegates the external port’s behavior to the subcomponent’s ports. The endpoint of a connection may also refer to a location outside the assembly.

4.11 Deployment Plan

A mapping of a configured application into a domain, this includes mapping monolithic implementations to nodes, connections to interconnects and bridges, and requirements to resources. Output of Planning, input to Preparation.

4.12 Domain

A target environment composed of independent nodes, interconnects, bridges and resources.

4.13 Installation

The act of taking a published software package and bringing it into a repository. See Section 6.3.2, “Installation,” on page 9 for more information.

4.14 Interconnect

A target used for the deployment of connections between components.

4.15 Interface

A named set of operations that characterize the behavior of an element.

4.16 Implementation Artifact

A artifact used or produced as a result of an implementation. These are commonly constituted as partial component implementations or monolithic implementations (usually "executable code").

4.17 Launch

The process of instantiating components on nodes in the target environment according to a deployment plan. Launching includes interconnecting and configuring component instances, as well as starting execution. (See Section 6.3.6, “Launch,” on page 9 for further details.)

4.18 Metadata

Information that characterizes data, Metadata are used to provide documentation for data products. In essence, metadata answer who, what, when, where, why, and how about every facet of the data that are being documented.

4.19 Monolithic Implementation

An indivisible implementation of a specific component interface using one or more deployable implementation artifacts.

4.20 Node

A run-time computational resource which generally has at least memory and often processing capability. Run-time implementation objects and components may reside on nodes.

4.21 Planning

The process of taking the requirements of the component package to be deployed and the resources of the target environment (where the software will be executed), and deciding which implementation and how and where the software will be run in that environment. (See Section 6.3.4, “Planning,” on page 9 for further information.)

4.22 Preparation

The process of performing work in the target environment to be ready to launch the software, such as moving binary files to the specific nodes in the target environment on which the software will execute. See Section 6.3.5, “Preparation,” on page 9 for further information.

4.23 Repository

A facility for storing metadata, and implementations.

4.24 Requirement

A feature requested by component implementations. Monolithic implementation requirements must be satisfied by node resources. Assembly subcomponent requirements must be satisfied by component implementation capabilities. Assembly connection requirements must be satisfied by interconnect and bridge resources.

4.25 Resource

A feature offered by a node, interconnect or bridge.

4.26 Shared Resource

A feature shared between two or more nodes. Either node can host monolithic implementations with a requirement that is satisfied by a shared resource.

5 Symbols and abbreviated terms

There are no special symbols or terms.

6 Introduction

"A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided ports." **[UML2S]**

In short, the idea of component-based development is to divide an application into small reusable components that can be connected to other components via ports, or, speaking the other way around, to compose applications by reusing and interconnecting existing components. An important idea is recursion, that an assembly — a set of interconnected components — can be seen as a component in itself, and therefore be reused the same way: an assembly always "implements" a specific component interface. Within an assembly, connections must be made between its subcomponents, and arrangements must be made for the assembly's external ports — ports of the component interface that the assembly is implementing — to delegate their behavior to subcomponent ports.

In order to instantiate, or deploy, a component-based application, instances of each subcomponent must first be created, then interconnected and configured. This specification deals with the deployment and configuration of component-based applications onto distributed systems, anticipating that subcomponents might be distributed among a set of independent, interconnected nodes called domain.

In this specification, an "application" is nothing special; an application is just a component that is assumed to be independently useful. As before, this component can be implemented directly (by a monolithic implementation), or it can be implemented by an assembly, where the implementations for its subcomponents can again be either monolithic or assemblies. Ultimately, any application can be decomposed into components that have monolithic implementations. At deployment time, decisions must be made about which implementations to deploy (execute) where.

6.1 Component-based Applications

In this specification, software components can have implementations that are either

- compiled code (called *monolithic* implementations), or
- assemblies of other components (*assembly* implementations, providing a recursive definition).

An assembly is defined as a set of components and interconnections that *implement a component*. There is no special "top level assembly," since assemblies are simply a method of specifying component implementations. To actually execute a component whose implementation is an assembly of lower level components, there must eventually be monolithic implementations at the "leaves" of the hierarchical implementation.

This definition of assembly means that the "application being deployed" is in fact a component. Its interface is defined as any component interface is defined. There is no special distinguished interface for "components that can be deployed as applications." Launching a component-based application results in an object that satisfies the interface of the component interface of the "application." Thus this specification has no need to treat the "thing being deployed" differently than a component, and enables implementation alternatives to be either monolithic compiled code artifacts or a hierarchical description of other components. This also means that any implementation, whether monolithic or assembly based, is reusable inside a larger application, *without being touched*.

A component package is a set of metadata and compiled code modules that contains implementations of a component interface. The implementations in a package can be a mix of monolithic and assembly implementations, with either or both present at any level of the hierarchy. Thus the creator of a component-based application produces a component package whose top level component interface represents the interface of the application.

Assemblies can consist of subcomponents whose implementations are inside the same package of software, or they can reference component packages that must exist in the environment outside the package containing the assembly. This not only allows packages from different vendors to be used together, but also allows dependent packages to be replaced without changing the other package or its configuration. No online update functionality is implied here.

To support heterogeneous systems, a package can contain more than one implementation, so that there is a choice at deployment time to find the implementation that best matches the target environment. For example, a package might contain implementations of the same component for Windows, Linux, or Java.

Monolithic component implementations express requirements that must be fulfilled by properties of the system on which they will be executed, e.g., the CPU type, or available hardware. The requirements of an assembly based implementation are implied by the requirements of its subcomponents, plus additional requirements on the connections between them.

6.2 The Target Environment

The target environment is termed a domain. Domains are composed of nodes, interconnects and bridges. Nodes have computational capabilities and are a target for executing component implementations; this definition encompasses personal computers as well as SMP systems, DSPs, or FPGAs. Interconnects provide a direct shared connection between nodes, e.g., representing an ethernet cable or a RapidIO fabric. Bridges route between interconnects, representing both routers and switches.

Nodes, interconnects, and bridges have resources that define their features, resources, and capacities. For a node, this might be the operating system type, memory, or available special hardware; an interconnect might describe its bandwidth as a resource. The platform independent model does not define types of resources, it just introduces the concept. Platform specific models or domain profiles may list concrete types of resources that are relevant to the platform or the domain.

An important aspect of the target environment is that the software that supports component execution on a particular node, must be able to be implemented independently of the deployment service as a whole. This interoperability boundary allows those interested in or knowledgeable of specific types of nodes to implement deployment support for those nodes without touching the overall deployment system for the target environment.

6.3 The Deployment Process

The model in this specification is based on a process definition of deployment. The process starts after the software is developed, packaged, and published by a software provider, and is acquired by the software owner, who deploys it. We call the owner at this point the *deployer.*

6.3.1 Preconditions for the Process of Deployment

Prior to deployment, the software has been packaged according to this specification, by the producer of the software, such that the metadata describing the software, and the binary compiled code artifacts, are combined into a *package*.

The package is published and somehow made available to the deployer, e.g., via a CDROM or web URL at an FTP site.

There is a *target environment*, consisting of a distributed system infrastructure (computers, networks, services), on which the software will ultimately run. There is a *repository*, which, at a minimum, is a staging area where the packaged software is captured prior to decisions about how it will run in the target environment.

6.3.2 Installation

We define *installation* as the act of taking the published software package and bringing it into a component software *repository* under the deployer's control, but the location (computer, file system, database) of this repository is not necessarily related to where the software will actually execute. It is a staging area where various policies of the deployer, such as security authentication, can be applied to the software prior to activities related to execution of the software. In the process defined here, installation is *not* related to moving software to the computers on which it will actually execute. Repositories do not necessarily need to be persistent, and they do not necessarily need to store or copy the software or metadata. Deep copy and shallow copy of the software are both supported under this specification.

6.3.3 Configuration

When the software is "in-house," in a repository, it can be functionally configured as to various default configuration options for later execution. An example would be: when this spreadsheet runs, the background color should be blue. Various configurations of a software package could be created. Configuration is *not* intended to capture the deployment decisions as to which implementation will be used or where the parts of the application will execute, but only functional configuration.

6.3.4 Planning

Planning how and where the software will run in the target environment is an activity that takes the requirements of the software to be deployed, along with the resources of the target environment on which the software will be executed, and decides which implementation and how and where the software will be run in that environment. We take care to separate this decision making step from actually acting on the decisions since there are important use cases for "advanced planning" that have *no immediate effect on the target environment*.

Advanced planning also allows for faster ultimate execution since all decisions can be made in advance (in cases where resource availability is not changing). Advanced planning can be done with an offline tool that does not interact with the actual runtime environment at all, but merely "keeps score" of how it is using up the resources known to be in the target environment. Of course there are also important use cases for "just-in-time" planning, where execution follows immediately after making planning decisions based on current dynamic resource availability in the target environment.

Planning results in a *deployment plan* specific to both the software being deployed and the target environment being deployed on.

6.3.5 Preparation

Given that we define planning as deciding how and where the software will run, we define *preparation* as performing work in the target environment to be ready to execute the software, such as moving binary files to the specific computers in the target environment on which the software will execute. This work is reusable if the software is executed more than once based on the same plan. Doing this work in advance reduces the startup time when the software is actually run. Just like planning, preparation can be done "just in time," as part of an automated scenario where the entire process happens at once.

6.3.6 Launch

Launching the application brings the application to an executing state, taking all resources that are known to be required based on the metadata in the packages. Component-based applications are launched by instantiating components, as planned, on nodes in the target environment. Launching includes interconnecting and configuring component instances, as well as starting execution. In this executing state, the application runs until it completes or is terminated via the same infrastructure that launched it.

6.3.7 All at Once, or Step by Step

This process model supports use cases where various combinations of these steps are done at different times using different tools. Of course there is the completely monolithic and automated case where a single deployment tool takes a web URL for a component package and executes it.

6.4 Relationship to the MDA

This specification is compliant with the Model Driven Architecture (MDA) defined by the OMG. It is composed of four main levels of models:

- A D&C Platform Independent Model (PIM), which constitutes the core of the specification. The D&C PIM defines the set of concepts and classes that are relevant for the implementation of the specification. The D&C PIM is explicitly independent of distributed component middleware technology (e.g., CORBA or J2EE), information formatting technology (e.g., XML DTD and XML), and programming languages (e.g., C++ and Java). Mappings to CORBA and XML are possible at the PSM level.
- A D&C UML profile designed to enhance the D&C PIM's readability and to facilitate the PIM-to-PSM mapping.
- A set of D&C Platform Specific Models that constitute realizations of the D&C PIM on concrete platforms. A required CCM PSM constitutes an integral part of this specification. A PIM-to-PSM mapping is explicitly defined for each PSM.
- A D&C Tool-Support Profile. This profile is closely related to the D&C PIM. The D&C PIM, in effect, defines the abstract syntax of a language for specifying the deployment and configuration of distributed components. The D&C Tool Support Profile defines, in effect, a concrete, UML-based syntax for this language. This concrete syntax can be employed using generic UML tools. The use of these stereotypes enables the automatic generation of D&C classes and descriptors from Deployment and Configuration UML models.

Based on the current requirements, there is no need to extend the UML metamodel at the M2 level. The use of profiles and stereotypes is sufficient to support the concepts defined in the D&C specification.

While not an explicit part of the current specification, it is also possible that different profiles of the D&C specification will be defined to satisfy the needs of different application domains, e.g., a D&C profile for web-based systems and a D&C profile for embedded systems. Because of the compatibility of the current D&C specification with the MOF 1.4, D&C profiles can be defined using the profiling mechanisms provided by UML. Such profiles would, most likely, extend the profiles defined in this specification.

7 Platform Independent Model

7.1 Segmentation of the Model

The Platform Independent Model (PIM) is segmented in two dimensions. This breaks down the overall model in a modular way such that interdependencies and complexity are minimized. The breakdown effectively creates six top level diagrams with a modest number of “external” dependencies between diagrams. The dependencies and relationships between these model segments are depicted on separate diagrams at the end of the model.

7.1.1 Dimension #1: Data Models vs. Management (or Runtime) Models

This distinction is between a model of descriptive information, vs. the model of runtime entities that process, create, provide, or store that information. In general, data models can be used to generate XML Schemas for storing and interchanging the data, and also to generate IDL data (or value) types and structures for the purpose of using the modeled data as parameters in the runtime interfaces. We use the word “management” in the sense of an active runtime entity that is dealing with (managing) the data. In general, data models are “leaves” in that they do not have intrinsic dependencies on the management/runtime models, whereas it is common for the runtime models to refer to the data models to describe parameter types in the interfaces.

In the PSMs, the IDL data structures and/or XML Schemas can be generated from the data models based on rules.

7.1.2 Dimension #2: Component Software vs. Target vs. Execution

In creating this PIM for the D&C of components, it is useful to segment the model elements according to the deployment process defined above. This should allow the different segments to be isolated according to usage (“need to know”) by actors, and then introduce (minimal) linkages or relationships between the elements as required in the different segments. This segmentation is roughly based on the process of deployment. It partitions the model with reduced/minimized interdependencies.

7.1.2.1 Component Software — output of the development, packaging, publishing processes

Component software models are about packaged component software, created by the component software development process, mostly independent of the specific target system(s) on which it will be deployed, although some requirements of the target are obviously included (compiled binary types, OS, etc.). Component software (all the packaged metadata and compiled code artifacts) is installed in a repository, configured, and used for deployment planning. It exists independent of any specific target system since the planning process (and the results of the planning process) is the bridge between this information and the ultimate execution on the target.

7.1.2.2 Target Environment — where the software will run

Target models are about the computing resource environment in which a component-based application will be executed. There is static basic configuration information as well as dynamic resource (and availability) information. This is the basic “platform” on which component based applications are run, including the

- *nodes* where software artifacts are loaded and used to instantiate components,
- *interconnects* among nodes, to which inter-component software connections are mapped, to allow the instantiated components to intercommunicate, and
- *bridges* among interconnects. While interconnects provide a direct connection between nodes, bridges provide a routing capability between interconnects.

Interconnects are like networks or busses that multiple nodes could be attached to, and similarly, a node might be attached to multiple interconnects (like a multi-homing network host). Nodes, interconnects, and bridges are collected into a *domain*, representing a particular target environment.

7.1.2.3 Execution — how the software is prepared to run, and executed based on its configuration

Execution models result from using component software models and target models to then express how component based applications will be run on a target. After creating and acquiring software, and after defining and using target information, there is planning and execution. Execution data models capture the results of planning — how the software will execute in the target environment (which implementations, running where). Execution management models use this planning information to actually prepare and launch applications. This execution happens at two levels: the whole application executing in the target environment, and the parts of the application that run on each node.

7.1.3 Summary of Model Segmentation Dimensions

Below is a table that summarizes the Data vs. Management/Runtime dimension as well as the Component Software vs. Target vs. Execution dimension. Thus the result of this segmentation can be thought of as 6 different “pages” of the model. The table below (which is not normative) summarizes the segments that are described in the next sections. PIM and PSM distinctions are weak in this summary.

Table 7.1 - D&C Model Segmentation Summary

	Data Model In PSMs, can generate XML Schemas and IDL data definitions	Management/Runtime Model Can imply interface IDL that may use data IDL derived from Data model. "Manager" applied to class names for consistency.	Deployment Process Usage How/when are the models used in the deployment process. "Tool" is used here for the client that performs and controls the process.
Component Software	**Component Data Model** of deployable component software, including descriptors for packages, interfaces, configurations, assemblies, and implementations. The top-level element is the **Package-Configuration**.	**Component Management Model**: The **RepositoryManager** interface, which manages descriptive information about Component Software. Key operations include: • Install Package from URL into Repository, with name and label. • Configure package, with name and label. • Retrieve package configuration info by name or top level interface UUID. Repository parses Component Software XML, and may be trivial in-memory (with data in IDL form only), file system based, database based. Repository can store data in persistent-IDL, XML, or private form. XML parsing can be early or late.	The software is produced and packaged according to this data model, and made available to the deployer. **Installation** tool supplies URL/location of the package to the RepositoryManager, which stores the package, *possibly* parsing, validating, authenticating etc., and creates a default configuration for the package in the repository. **Configuration** tool stores settings referring to a package, *optionally* after retrieving package information for config property validation. **Planning** tool retrieves information *in IDL data form* for decision making. Repository provides URL/location of binary artifacts so that plan need not reference repository.
Target	**Target Data Model** of the target domain, including nodes, interconnects, bridges, and resources. The top level collection of this information is the **Domain**.	**Target Management Model**: The **TargetManager** interface manages **Domain** information, either offline (simply parsed from private XML) or online. It needs to allow for efficient static vs. dynamic information. Key methods: • Get base info (to allow planning tool to do preprocessing/caching of static data). • Get current info (to plan based on dynamic resource information). • Commit resources (to commit resources that are used up in the plan).	Target configuration tools can provide user interfaces to build and emit target data model XML. **Planning** tool obtains target information (in IDL data form) and creates plans. An online **TargetManager** would know and supply dynamic information collected from nodes. A **TargetManager** would initially read provided target description from XML files, and then provide the information using the data model. The TargetManager can be told about changed or new domain elements at run time.
Execution	**Execution Data Model** of decisions configuring and connecting and locating component software on a target. This is the **DeploymentPlan**.	**Execution Management Model**: The **ExecutionManager** is the runtime entity for execution of component software on the target according to the plan. Key methods: • Prepare for execution, using plan, returning **"**factory**"** reference (**Application Manager**) • Launch based on factory, returning **Application** reference. • Lifecycle control, using **Application** ref. **NodeManager** performs the subset of execution on each node.	**Preparation** tool may parse plan XML (if not bundled with planning tool), and deliver plan in IDL-data form to Execution Manager. Thus an all-in-one tool would only have the plan in memory. **Launch** tools simply use the factory reference (Application Manager) to launch application, possibly managing the lifecycle.

The table above introduces the main elements of the platform independent model for deployment and configuration. The first column lists the three top-level data elements **PackageConfiguration**, **Domain**, and **DeploymentPlan**. The second column lists the three top-level management interfaces, **RepositoryManager**, **TargetManager**, and **ExecutionManager**/ **NodeManager**. Each of these classes is elaborated in the upcoming sections. The third column lists use cases that are supported by this model: Installation, Configuration, Planning, Preparation, and Launch. Use cases imply actors that enact them: an Administrator enacts Installation and Configuration, a Planner does the Planning, and an Executor enacts Preparation and Launch.

While the component, target, and execution models are self-contained and passive, actors are the glue between them. Actors actively interface with the various management models and exchange information using the various data models. All behavior of deployment and configuration is defined by actors, as elaborated in the next chapter.

7.2 Model Diagram Conventions

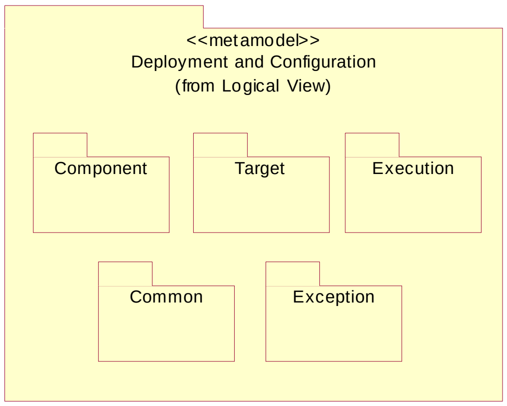


Figure 7.1 - Deployment and Configuration Model Package Structure

This specification uses UML diagrams **[UML1]** to show classes and their relationships. All classes are part of the Deployment and Configuration package, which contains the Component, Target, Execution, Common, and Exception subpackages.

The Deployment and Configuration package is restricted to the MOF 1.4 subset of UML **[MOF1]**. Some non-normative diagrams from other packages are shown for explanatory purposes.

If, in a UML diagram, a class's attribute and operation compartments are suppressed, then this class is elaborated elsewhere. In this case, the diagram might also not show all of the class' associations. However, if a class is shown to have only an attribute or an operation compartment, then this signifies that the not-shown compartment is empty. I.e., if a class is shown with an attribute but no operation compartment, then the class does not have any operations.

The name of an AssociationEnd is suppressed in a diagram if and only if the AssociationEnd is not a referencedEnd for any Reference. Therefore, if the name of an AssociationEnd is present, then the class at the otherEnd of the association contains a Reference with this AssociationEnd as its referencedEnd.

If the name of an AssociationEnd is suppressed, the name of the AssociationEnd's type, but with a lowercase character, is used as the AssociationEnd's name. (This is the same implicit rule as in OCL.)

Association names are suppressed in diagrams, default names are used throughout the model. For unidirectional associations (where exactly one AssociationEnd is navigable), the name of the class at the source (non-navigable) end plus an underscore plus the name of the navigable end is used as the name of the association. For bidirectional associations, the concatenation of the class names at both ends, in alphabetical order, with an underscore inbetween, is used as the name of the association. (The model does not contain associations with two non-navigable AssociationEnds.)

Unless otherwise mentioned, the multiplicity on the near end of navigable associations is zero to many, and the multiplicity on the near end of compositions is one to one.

This specification uses the notation of placing the multiplicity in square brackets after the type, as in "`label`: **String** [1]." If the multiplicity is omitted from an attribute, parameter, or return value, the default of exactly one [1] is used.

Standard attributes are used as needed on classes for readability and identity purposes. The standard attribute names are:

- `label`: A human-readable label that is not evaluated by the deployment system. It can be used to annotate classes with a user-defined string. Content is optional.
- `UUID`: A globally unique identifier (of type **String**) that is a URN **[URI]** (defined as a URI whose purpose is identity, not access (which is a URL)). The value must be an "absolute URI" with a URI scheme that allows hierarchical URIs. If two entities having such an attribute have identical UUIDs, then the deployment system can assume they are functionally identical and interchangeable, with identical contents. This enables the deployment system to cache information based on this UUID attribute, and know that a previously cached/processed entity can be used if it had the same UUID value. The value of this attribute is optional; if it is missing or the empty string, it is interpreted as meaning that the object is transient, the UUID value will be considered unequal to any other UUID value (including another empty string), and thus the object will never be considered the same as any other object, thus precluding any caching or any aliasing. This optionality is convenient in many development scenarios (e.g., recompile with no change in metadata) and provides certain advice to the implementation, but is generally unsuitable for true deployed, configuration-managed, production versions of such objects. Implementations of this specification may have options that insist on the existence of UUID values, but this is not necessary for compliance. Human readable URI schemes are recommended, but not mandatory.
- `name`: Names are both human-readable and machine-readable. Names are mandatory, and they must be unique within their container or context. For example, in the case of a **Node**, the **Node**'s name must be unique within the **Domain**. Furthermore, entities with the "name" attribute are contained by entities that have either "name" attributes or "UUID" attributes, and thus there is a "virtual" URN for each entity, with a "name" attribute formed by tracing the containment relationship upwards until a "UUID" attribute exists and forming an absolute and hierarchical URN from the UUID and the "name" attributes as path components according to **[URI]**.
- `location`: References an entity outside of the model. The location attribute is of type **String**, its value(s) must comply to the URI syntax **[URI]**. The value represents a URL, which is a URI whose purpose is access, not identity (which is a URN). Multiple alternative locations to the same entity may be supplied since the multiplicity is "1..*"; applications can then choose any of these equivalent locations to access the entity (e.g., choosing a local file URI over an http reference). The contents accessed are identical, although the actual locations (e.g., servers or file systems) may in fact be different. One of the values must use the "`http`" scheme, which is the only protocol that is required to be supported.
- `specificType`: Identifies the most specific type of an interface. Components or ports with equal specificType are type equivalent. The `specificType` attribute is of type **String**; consequently, string comparison is used to compare them. PSMs define the format.
- `supportedType`: Identifies all types that an interface can support. The type of this attribute is a sequence of Strings. A component or port can satisfy a requirement on any of the types listed among the supported types. The `supportedType` attribute includes the most specific type (from the `specificType` attribute) and all directly or indirectly inherited types in no particular order.
- `source`: A string formatted as a relative URI **[URI]** that identifies an element in the Component Data Model, along with the containing elements. Each top level **PackageConfiguration** (directly retrieved from a repository) is represented by an empty segment followed by a segment containing the URL (appropriately escaped) of the repository followed by a segment containing the `installationName` of the PackageConfiguration. All other segments represent "name" attributes of contained model elements. This supports complete navigation to model elements that were chosen from multiple repositories used by the planner, without requiring any collaboration with the planner.

Several classes contain a set of informational properties. These properties can be used by tools to annotate model elements with non-functional information (e.g., authorship, license, digital signature). The names of informational properties shall be valid URIs. PSMs may define a set of well-known informational properties (by identifying their URI and a corresponding property type).

To enhance readability, in the PIM below we annotate classes with stereotypes that define two orthogonal dimensions to the class structure and relationships in the model. The first follows the Data Model vs. Management/Runtime Model dimension in the segmentation discussion above. We will use the «**Description**» and «**Manager**» stereotypes to make this distinction.

In general, «**Description**» classes generate data structures and schema, and «**Manager**» classes generate runtime interfaces.

The second annotation dimension is to identify, for «**Description**» classes, the actor in the development process for which this class a work product. These stereotypes are essentially an annotation that highlights authorship (and inherits from «**Description**», without introducing extra relationship detail in the diagrams).

Although these development actors are defined in detail later, we will briefly introduce them here:

- The «**Specifier**» specifies the interface and functional contract for components' implementations.
- The «**Implementer**» creates concrete (monolithic, coded) implementations of components including their metadata.
- The «**Packager**» creates packages (bundles) of component implementations.
- The «**Planner**» makes decisions about deployment based on target capabilities and component requirements.
- The «**DomainAdministrator**» prepares information about the target environment.

The «**Implementer**» is in fact inherited by two derived stereotypes:

- The «**Developer**» creates monolithic (e.g., source coded/compiled) implementations.
- The «**Assembler**» creates assembly-based implementations of components.

Classes that are the work product of more than one actor are annotated with the generic «**Description**» stereotype. The creating actor can be inferred from context.

The «**Exception**» stereotype is used for exceptions that are raised by operations of management classes.

These stereotypes are represented by the “profile” diagram:

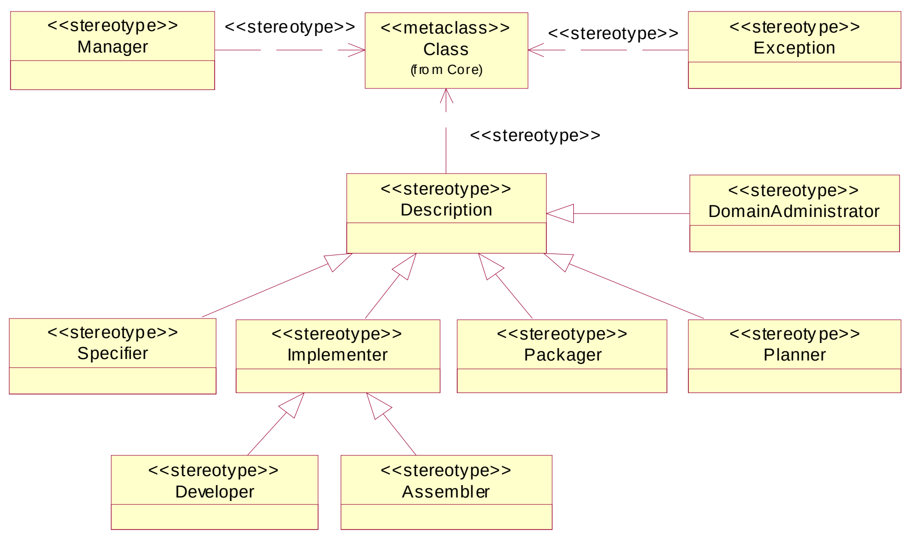
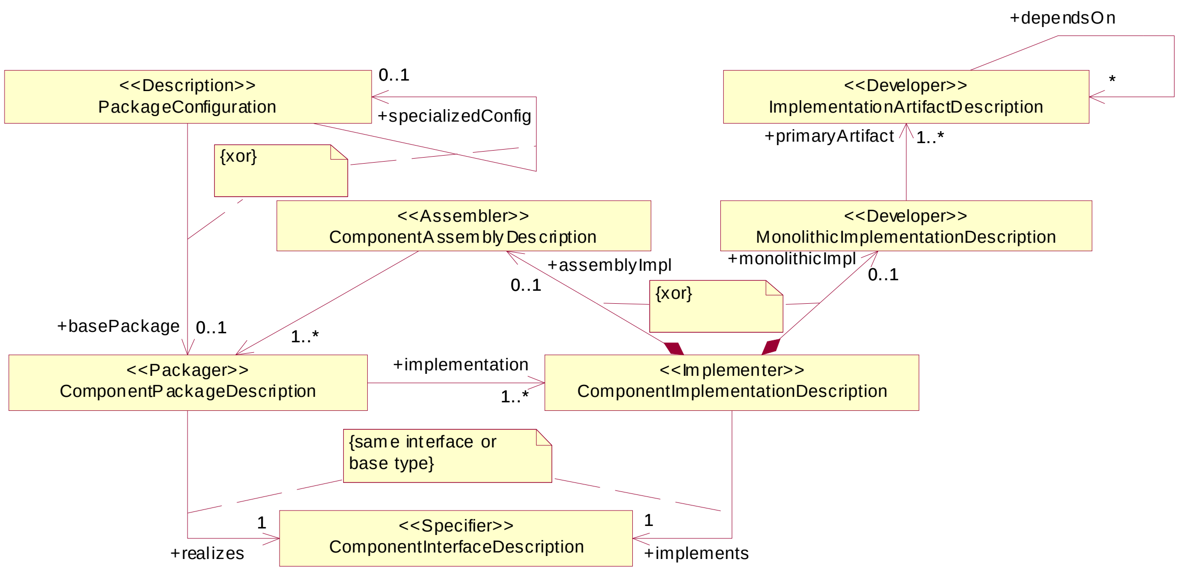


Figure 7.2 - Stereotypes used for class annotations

7.3 Component Data Model

A component has an interface composed of operations, attributes, and ports that may be connected to other components. A component may have a concrete (monolithic) implementation contained in an artifact (e.g., an executable file or library), or it may be recursively implemented by an assembly: a set of interconnected sub-components.

A component package contains multiple implementations of the same component. This allows distribution of a set of implementations with different properties (e.g., for different operating systems) or different hierarchies, to be distributed in a single package. Packages are installed into a repository, where they may be configured (e.g., overriding default property values) prior to deployment.



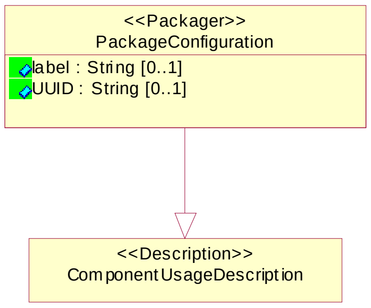
+dependsOn
<<Description>> PackageConfiguration
0..1
+specializedConfig
<<Developer>> ImplementationArtifactDescription
*
+primaryArtifact
1..*
{xor}
<<Assembler>> ComponentAssemblyDescription
<<Developer>> MonolithicImplementationDescription
+assemblyImpl
+monolithicImpl
0..1
0..1
{xor}
+basePackage
0..1
1..*
<<Packager>> ComponentPackageDescription
+implementation
1..*
<<Implementer>> ComponentImplementationDescription
{same interface or base type}
1
<<Specifier>> ComponentInterfaceDescription
1
+realizes
+implements

Figure 7.3 - Component Data Model Overview

Figure 7.3 is an overview of the Component Data Model and represents the information about installed and configured packages provided by the **RepositoryManager**. Details about each class will be presented in the following sections.

7.3.1 PackageConfiguration

7.3.1.1 Description



<<Packager>> PackageConfiguration
label : String [0..1]
UUID : String [0..1]
<<Description>> ComponentUsageDescription

A **PackageConfiguration** describes one configuration of a component package, and represents a reusable work product. It inherits from **ComponentUsageDescription**, which allows for several means of specifying a **ComponentPackageDescription** that is to be configured for (re-)use. It adds a label and a UUID to identify the work product.

7.3.1.2 Attributes

- `UUID`: **String** [0..1]
  A unique identifier for this **PackageConfiguration**.
- `label`: **String** [0..1]
  An optional human-readable label.

7.3.1.3 Associations

See **ComponentUsageDescription**.

7.3.1.4 Constraints

If the UUID attribute is not the empty string, then it must contain a unique identifier; **PackageConfiguration** instances with the same non-empty UUID must be identical.

```
context PackageConfiguration inv:
    self.UUID <> "" implies
        PackageConfiguration.allInstances->forAll (p |
            p.UUID = self.UUID implies p = self)
```

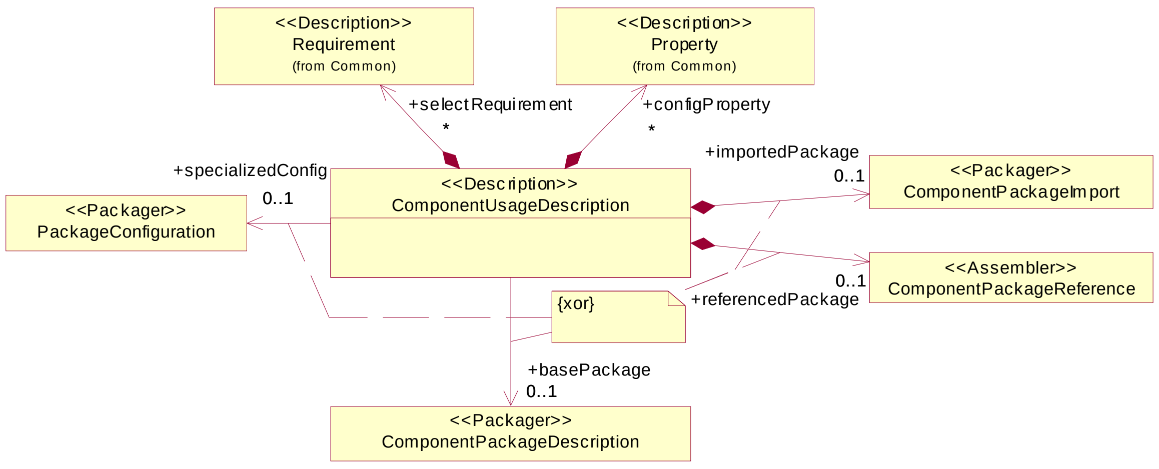
7.3.1.5 Semantics

PackageConfiguration elements represent a reusable work product. They are the top-level element in a package.

PackageConfiguration exhibits the same semantics as **ComponentUsageDescription**.

7.3.2 ComponentUsageDescription

7.3.2.1 Description



<<Description>> Requirement (from Common)
<<Description>> Property (from Common)
+selectRequirement
*
+configProperty
*
+specializedConfig
0..1
<<Packager>> PackageConfiguration
<<Description>> ComponentUsageDescription
+importedPackage
0..1
<<Packager>> ComponentPackageImport
<<Assembler>> ComponentPackageReference
0..1
+referencedPackage
{xor}
+basePackage
0..1
<<Packager>> ComponentPackageDescription

ComponentUsageDescription describes the (re-)use and configuration of an existing package. Configuration properties are used to configure the package's properties; their names and types must match external properties of the component that the package implements. Selection requirements are used to influence deployment decisions by matching them against implementation capabilities in the **ComponentImplementationDescription**.

ComponentUsageDescription is used (inherited) by **PackageConfiguration**, which configures a package as a work product, which may thus be recursive (a package can be no more than a reference to another package, with some configuration data), and by **SubcomponentInstantiationDescription**, which configures a package to provide an instance for a subcomponent in an assembly.

7.3.2.2 Attributes

No attributes.

7.3.2.3 Associations

- `basePackage`: **ComponentPackageDescription** [0..1]
  Links to a **ComponentPackageDescription** that this **ComponentUsageDescription** uses and configures.
- `specializedConfig`: **PackageConfiguration** [0..1]
  Links to a **PackageConfiguration** that is specialized and configured.
- `importedPackage`: **ComponentPackageImport** [0..1]
  Imports a package using a URI location. Imported packages are resolved during installation in a repository.
- `referencedPackage`: **ComponentPackageReference** [0..1]
  References a package that is installed in a repository, using its installation name, UUID or interface type. Referenced packages are resolved during planning.
- `selectRequirement`: **Requirement** [*]
  During planning, selection requirements are matched against capabilities in the **ComponentImplementationDescription** elements to identify implementations with desired characteristics.
- `configProperty`: **Property** [*]
  Properties to configure the application component with.

7.3.2.4 Constraints

A **ComponentUsageDescription** must have a base package, specialize a **PackageConfiguration**, import a package, or reference a package.

```
context ComponentUsageDescription inv:
  Set{self.basePackage, self.specializedPackage,
    self.importedPackage, self.referencedPackage}->size() = 1
```

Configuration properties must match configurable properties, as identified by the component interface (**ComponentInterfaceDescription** element), in name and type.

7.3.2.5 Semantics

A **ComponentUsageDescription** ultimately resolves to a single **ComponentPackageDescription** in one of four different ways:

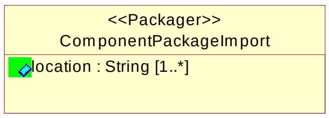
- By pointing directly to a **ComponentPackageDescription** “base package.”
- By (recursively) specializing a **PackageConfiguration**.
- By importing a package by URI. Imported packages are resolved during installation; the **RepositoryManager** replaces all **ComponentPackageImport** instances with a “`specializedConfig`” **PackageConfiguration**.

- By referencing a package in a repository. Referenced packages are resolved during planning; the Planner uses information in the **ComponentPackageReference** to locate a matching installed package, acting as if the referenced **PackageConfiguration** replaced the "`referencedPackage`."

The set of configuration properties and selection requirements that are accumulating by traversing one or more **ComponentUsageDescription** elements are then applied to the "ultimate" **ComponentPackageDescription** base package. A configuration property in a **ComponentUsageDescription** overrides a configuration property with the same name in a specialized **PackageConfiguration** or in a base **ComponentPackageDescription**.

7.3.3 ComponentPackageImport

7.3.3.1 Description



Imports a package via an URL.

7.3.3.2 Attributes

- `location`: **String** [1..*]
  Alternative locations of the package that is to be imported.

7.3.3.3 Associations

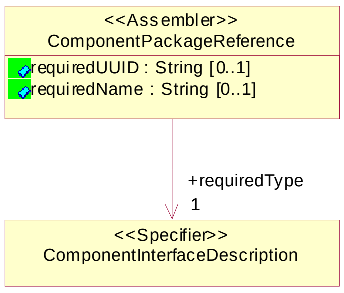
None.

7.3.3.4 Semantics

A **ComponentPackageImport** can be used instead of a **PackageConfiguration** to import a package, rather than providing the package "inline."

7.3.4 ComponentPackageReference

7.3.4.1 Description


<<Assembler>>
ComponentPackageReference
requiredUUID : String [0..1]
requiredName : String [0..1]
+requiredType
1
<<Specifier>>
ComponentInterfaceDescription

Indirectly references a package to be found in a repository. The reference is accomplished by using a combination of the `requiredUUID` and `requiredName` attributes and the `requiredType` association. All requirements that are present must be satisfied. The `requiredName` refers to the installation name under which the package was installed into a repository. The three attributes satisfy a variety of reference patterns similar to those found in DLL or shared library systems.

7.3.4.2 Attributes

- `requiredUUID`: **String** [0..1]
  The reference is to an installed **PackageConfiguration** with this specified identity.

- `requiredName`: **String** [0..1]
  The reference is to an installed **PackageConfiguration** with this specified installation name in its repository.

7.3.4.3 Associations

- `requiredType`: **ComponentInterfaceDescription**
  The reference is to an installed **PackageConfiguration** that ultimately references a **ComponentPackageDescription** that is compatible with this specified type.

7.3.4.4 Constraints

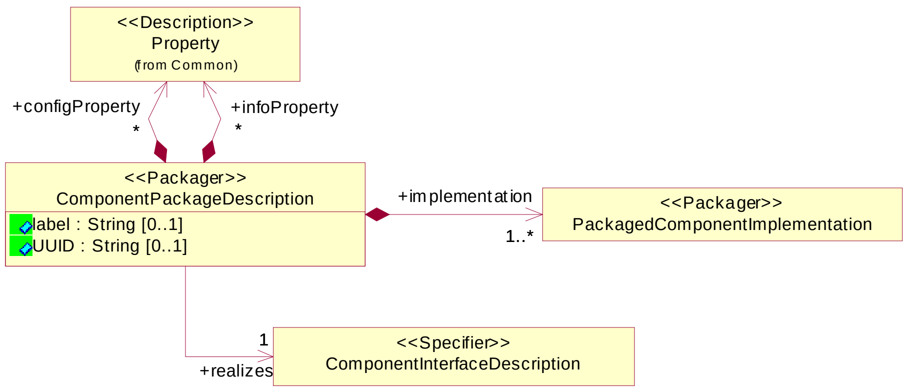
No constraints.

7.3.4.5 Semantics

The planner will search one or more repositories for package configurations that support the `requiredType`, and that match the `requiredUUID` and `requiredName` attributes, if present.

7.3.5 ComponentPackageDescription

7.3.5.1 Description



A **ComponentPackageDescription** describes multiple alternative implementations of the same component interface. It references the interface description for the component and contains a number of configuration properties to configure the running components (which may override implementation-defined properties and which may be overridden by a **PackageConfiguration**). These configuration properties enable the packager to define default values for a component's properties regardless of which implementation for that component is chosen at deployment (planning) time.

7.3.5.2 Attributes

- `label`: **String** [0..1]
  An optional human-readable label for the package.

- `UUID`: **String** [0..1]
  An optional unique identifier for this package.

7.3.5.3 Associations

- `realizes`: **ComponentInterfaceDescription** [1]
  A **ComponentPackageDescription** describes implementations that realize a certain component interface.

- `implementation`: **PackagedComponentImplementation** [1..*]
  The alternative implementations for this component.

- `configProperty`: **Property** [*]
  These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen.

- `infoProperty`: **Property** [*]
  Non-functional annotation properties.

7.3.5.4 Constraints

All implementations referenced by this **ComponentPackageDescription** must implement the same interface as realized by the package, or a derived interface.

```
context ComponentPackageDescription inv:
    self.implementation.referencedImplementation->forAll (
        implements.supportedType->includes (self.realizes.primaryType))
```

If the `UUID` attribute is not the empty string, then it must contain a unique identifier for the package; packages with the same non-empty `UUID` must be identical.

```
context ComponentPackageDescription inv:
    self.UUID <> “” implies
        ComponentPackageDescription.allInstances->forAll (p |
            p.UUID = self.UUID implies p = self)
```

The names assigned to implementations must be unique within this package.

```
context ComponentPackageDescription inv:
    implementation->forAll (i1, i2 | i1.name = i2.name implies i1=i2)
```

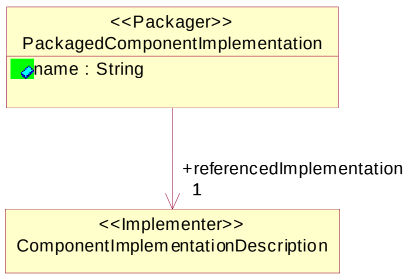
Configuration properties must match configurable properties, as identified by the component interface (**ComponentInterfaceDescription** element), in name and type.

7.3.5.5 Semantics

Configuration properties can be overridden in a **PackageConfiguration**. All implementations in the package are considered equally suitable for deployment, pending compatibility between implementation artifact requirements and node resources, and selection properties required by a **PackageConfiguration**.

7.3.6 PackagedComponentImplementation

7.3.6.1 Description


<<Packager>>
PackagedComponentImplementation
name : String
+referencedImplementation
1
<<Implementer>>
ComponentImplementationDescription

PackagedComponentImplementation is used by the **ComponentPackageDescription** to assign names to alternative **ComponentImplementationDescription** elements within that package. This information can be used to identify elements within the Component Data Model using a "path name" from the top level package downwards.

7.3.6.2 Attributes

- `name`: **String**
  The name assigned to this implementation.

7.3.6.3 Associations

- referencedImplementation: **ComponentImplementationDescription** [1]
  The implementation that is referenced by this package.

7.3.6.4 Constraints

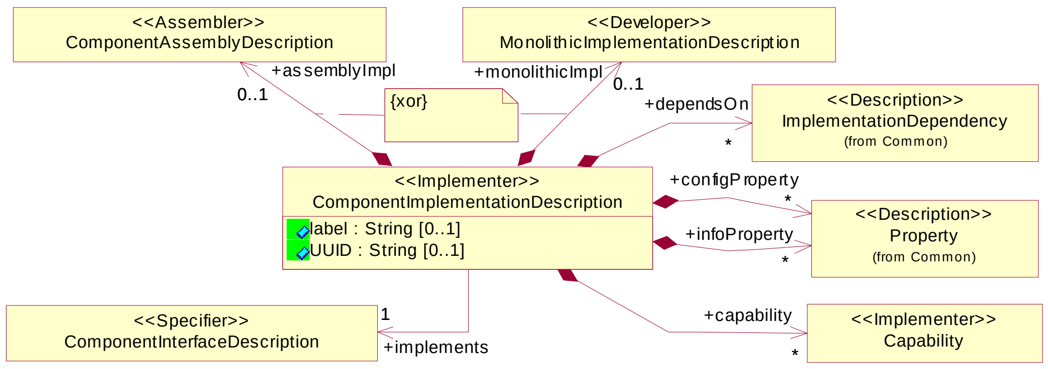
No constraints.

7.3.6.5 Semantics

No semantics.

7.3.7 ComponentImplementationDescription

7.3.7.1 Description



<<Assembler>> ComponentAssemblyDescription
<<Developer>> MonolithicImplementationDescription
+assemblyImpl
0..1
+monolithicImpl
0..1
{xor}
+dependsOn
*
<<Description>> ImplementationDependency (from Common)
<<Implementer>> ComponentImplementationDescription
label : String [0..1]
UUID : String [0..1]
+configProperty
*
+infoProperty
*
<<Description>> Property (from Common)
<<Specifier>> ComponentInterfaceDescription
1
+implements
+capability
*
<<Implementer>> Capability

A **ComponentImplementationDescription** describes a specific implementation of a component interface. This implementation can be either assembly based or monolithic. The **ComponentImplementationDescription** may contain configuration properties that are used to configure each component instance (“default values”). Implementations may be tagged with user-defined capabilities. Administrators can then select among implementations using selection requirements in a **PackageConfiguration**; Assemblers can place requirements on implementations in a **SubcomponentInstantiationDescription**.

7.3.7.2 Attributes

- `label`: **String** [0..1]
  An optional human-readable label for the implementation.

- `UUID`: **String** [0..1]
  An optional unique identifier for this implementation.

7.3.7.3 Associations

- `implements`: **ComponentInterfaceDescription** [1]
  The component interface implemented by this implementation.

- `assemblyImpl`: **ComponentAssemblyDescription** [0..1]
  In case of an assembly based implementation, this describes the assembly.

- `monolithicImpl`: **MonolithicImplementationDescription** [0..1]
  In case of a monolithic implementation, this describes the monolithic implementation.

- `configProperty`: **Property** [*]
  These are implementation specific configuration properties that are used to configure the component once instantiated.

- `infoProperty`: **Property** [*]
  Non-functional annotation properties.

- `capability`: **Capability** [*]
  These are tags that a **PackageConfiguration** can match against to discriminate between implementations.

- `dependsOn`: **ImplementationDependency** [*]
  Expresses a dependency on other packages; implementations of the referenced interfaces must be deployed in the target environment before this implementation can be deployed.

7.3.7.4 Constraints

An implementation is either assembly based or monolithic, consequently there must be either a **ComponentAssemblyDescription** or a **MonolithicImplementationDescription**, but not both.

```
context ComponentImplementationDescription inv:
    self.assemblyImpl.size() = 1 xor
    self.monolithicImpl.size() = 1
```

If the `UUID` attribute is not the empty string, then it must contain a unique identifier for the implementation; implementations with the same non-empty `UUID` must be identical.

```
context ComponentImplementationDescription inv:
    self.UUID <> "" implies
        ComponentImplementationDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

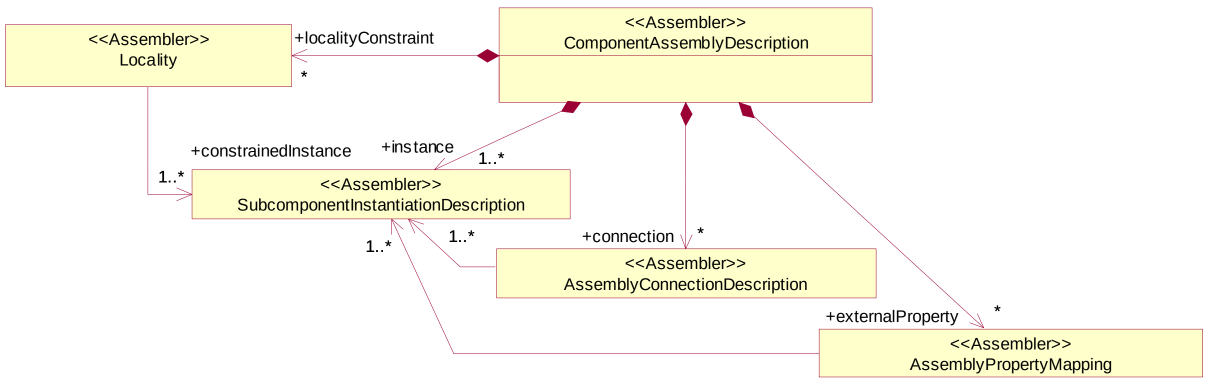
Configuration properties must match configurable properties, as identified by the component interface (**ComponentInterfaceDescription** element), in name and type.

7.3.7.5 Semantics

Configuration properties can be overridden in a **ComponentPackageDescription** or in a **PackageConfiguration**.

7.3.8 ComponentAssemblyDescription

7.3.8.1 Description



<<Assembler>> Locality
+localityConstraint
*
<<Assembler>> ComponentAssemblyDescription
+constrainedInstance
1..*
+instance
1..*
<<Assembler>> SubcomponentInstantiationDescription
1..*
1..*
+connection
*
<<Assembler>> AssemblyConnectionDescription
+externalProperty
*
<<Assembler>> AssemblyPropertyMapping

In the case of an assembly based implementation, the **ComponentAssemblyDescription** contains information about sub-component instances (**SubcomponentInstantiationDescription**), connections among ports (**AssemblyConnectionDescription**), and about the mapping of the assembly's properties (i.e., of the component that the assembly is implementing) to properties of its subcomponents. In addition, it may contain locality requirements on subcomponent instances that influence the deployment planning process.

7.3.8.2 Attributes

No attributes.

7.3.8.3 Associations

- `instance`: **SubcomponentInstantiationDescription** [1..*]
  Describes instances of subcomponents.
- `localityConstraint`: **Locality** [*]
  Describes locality constraints for subcomponent instances.
- `connection`: **AssemblyConnectionDescription** [*]
  Describes connections between ports.
- `externalProperty`: **AssemblyPropertyMapping** [*]
  Maps the external properties of the component that is implemented by the assembly to properties of subcomponent instances.

7.3.8.4 Constraints

The elements within this **ComponentAssemblyDescription** (**SubcomponentInstantiationDescription**, **AssemblyConnectionDescription**, and **AssemblyPropertyMapping**) must have unique names within this context.

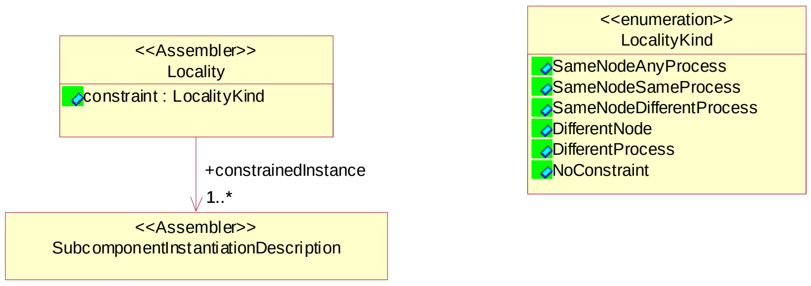
```
context ComponentAssemblyDescription:
    let elements = Set {self.instance, self.connection,
                        self.externalProperty}
    elements->forAll (e1, e2 | e1.name = e2.name implies e1 = e2}
```

7.3.8.5 Semantics

An assembly is composed of components and itself implements a component, as implied by the **ComponentImplementationDescription** that this **ComponentAssemblyDescription** is contained in. The component being implemented by the assembly is referred to as the "external component" of the assembly. Connections exist among the subcomponents' ports and the external component's ports, similar to a wiring diagram in circuit design, where a circuit is designed by wiring chips among themselves and wiring them to external pins.

7.3.9 Locality

7.3.9.1 Description



The Locality element specifies locality requirements for the subcomponent instances of an assembly. These requirements are evaluated during the deployment planning process and influence the decision where a particular subcomponent instance shall be deployed to.

The Locality element allows for requesting the collocation or separation of two or more subcomponent instances of an assembly. The major reason for separating component instances is to support fault tolerance models. The rationale for requesting the collocation of component instances is often performance reasons.

Collocation constraints should be used very carefully since they limit the possible decisions of the deployment planning process. Collocation constraints are often dedicated to a particular target environment and may thus limit the reuse of component assemblies for other target environments having a different topology and different capabilities. It is often better to put a special requirement on the communication path between two components requesting a high communication bandwidth instead of requesting collocation of those components.

7.3.9.2 Attributes

- `constraint`: **LocalityKind**
  The value of this attribute identifies the kind of the locality constraint.

7.3.9.3 Associations

- `constrainedInstance`: **SubcomponentInstantiationDescription** [1..*]
  Refers to the description of those subcomponent instances the locality requirements are dedicated to.

7.3.9.4 Constraints

No constraints.

7.3.9.5 Semantics

See the description of the **LocalityKind** class for the semantics of locality constraint options.

At planning time, the planner evaluates node locality constraints, and assigns constrained instances to the same node (`SameNodeAnyProcess`, `SameNodeSameProcess`, `SameNodeDifferentProcess`) or to different nodes (`DifferentNode`). Process locality constraints are recorded in the **PlanLocality** element of the **DeploymentPlan**, to be evaluated by the node manager.

7.3.10 LocalityKind

7.3.10.1 Description

See above.

7.3.10.2 Attributes

No attributes.

7.3.10.3 Associations

No associations.

7.3.10.4 Constraints

No constraints.

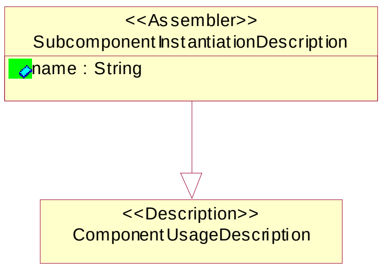
7.3.10.5 Semantics

The choices for locality constraints are:

- `SameNodeAnyProcess`: This value specifies that the subcomponent instances the **Locality** element refers to shall be deployed onto the same node. They can be started in the same process or in different processes by the deployment engine.
- `SameNodeSameProcess`: This value specifies that the subcomponent instances the **Locality** element refers to shall be deployed onto the same node. In addition, they shall also be instantiated in the same process by the deployment engine.
- `SameNodeDifferentProcess`: This value specifies that the subcomponent instances the **Locality** element refers to shall be deployed onto the same node but instantiated in different processes by the deployment engine.
- `DifferentNode`: This value specifies that the subcomponent instances the **Locality** element refers to shall be deployed onto different nodes. This implies that they will be instantiated in different processes.
- `DifferentProcess`: This value specifies that the subcomponent instances the **Locality** element refers to shall be instantiated in different processes by the deployment engine. It is not of interest whether the subcomponent instances are deployed onto the same node or onto different nodes.
- `NoConstraint`: This value specifies that there is no locality constraint defined for the subcomponent instances the **Locality** element is associated with. The purpose of this special value is to enable to switch locality constraints temporarily off without loosing the structure of the **ComponentAssemblyDescription**, i.e., the **Locality** class instance with its associations to **SubcomponentInstantiationDescription** can be kept for later reuse but has currently no impact on the deployment planning.

7.3.11 SubcomponentInstantiationDescription

7.3.11.1 Description


<<Assembler>>
SubcomponentInstantiationDescription
name : String
<<Description>>
ComponentUsageDescription

In an assembly based implementation, the **SubcomponentInstantiationDescription** describes one instance of a subcomponent. This instance is provided by a package. For this purpose, **SubcomponentInstantiationDescription** inherits **ComponentUsageDescription**, allowing to link directly, link indirectly, import, or reference a package that will be used to provide an implementation for the subcomponent instance. **SubcomponentInstantiationDescription** adds a `name` attribute to identify the subcomponent instance within the assembly.

7.3.11.2 Attributes

- `name`: **String**
  Identifies this subcomponent instance within the assembly.

7.3.11.3 Associations

See **ComponentUsageDescription**.

7.3.11.4 Constraints

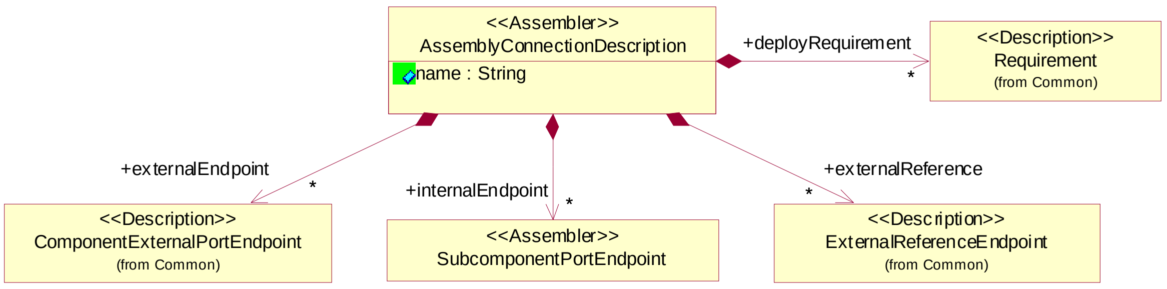
No additional constraints.

7.3.11.5 Semantics

Same as for **ComponentUsageDescription**.

7.3.12 AssemblyConnectionDescription

7.3.12.1 Description



An **AssemblyConnectionDescription** element describes a connection that is to be made among ports within an assembly. A connection can be thought of as a single path in a circuit wiring diagram with multiple endpoints. In this analogy, a signal that is sent onto the path is received by all receiving endpoints. There are three different types of endpoints, the most obvious being the **SubcomponentPortEndpoint**, which reflects a connection to the port of a subcomponent within the assembly. The **ComponentExternalPortEndpoint** reflects a connection to an external port of the component that is implemented by the assembly. The **ExternalReferenceEndpoint** reflects a connection to a location outside the assembly by URL (e.g., using a corbaname reference).

Some deployment requirements may be associated with the connection information; these requirements must be satisfied by the interconnect(s) in the target model over which the connection is routed at deployment time. PSMs and domain specific profiles will define a vocabulary for deployment requirements.

7.3.12.2 Attributes

- `name`: **String**
  Identifies this connection within the assembly.

7.3.12.3 Associations

- `deployRequirement`: **Requirement** [*]
  These connection requirements must be satisfied by the interconnects over which the connection is routed.

- `internalEndpoint`: **SubcomponentPortEndpoint** [*]
  Identifies a port of a component within the assembly as an endpoint of this connection.

- `externalEndpoint`: **ComponentExternalPortEndpoint** [*]
  Identifies a port of the component that is implemented by the assembly as an endpoint of this connection.

- `externalReference`: **ExternalReferenceEndpoint** [*]
  Identifies a location outside the assembly as an endpoint of this connection.

7.3.12.4 Constraints

The number of endpoints to a connection must be at least two.

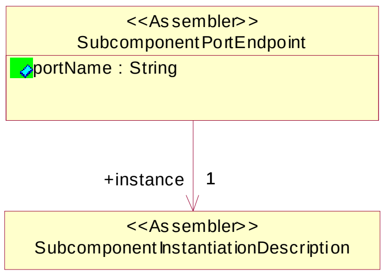
```
context AssemblyConnectionDescription inv:
    Set{self.externalEndpoint,
        self.internalEndpoint,
        self.externalReference}->size() >= 2
```

7.3.12.5 Semantics

At assembly design time, the compatibility of the endpoints can be verified based on the information known about the endpoints, e.g., appropriate user, provider, multiplex semantics. At planning time, compatibility of the connection's requirements with the resources of the interconnects that the connection is routed over will be verified. At execution time, connections between the endpoints will be established.

7.3.13 SubcomponentPortEndpoint

7.3.13.1 Description



Identifies a port of a component within the assembly as an endpoint of the connection described by the **AssemblyConnectionDescription** that this element is contained in.

7.3.13.2 Attributes

- `portName`: **String**
  The name of the port of the associated subcomponent instance that is to be an endpoint of this connection.

7.3.13.3 Associations

- `instance`: **SubcomponentInstantiationDescription** [1]
  The associated subcomponent instance.

7.3.13.4 Constraints

The port name must be valid for the referenced component.

context SubcomponentPortEndpoint inv:
self.instance.package->size() = 1 implies
self.instance.package.interface.port.exists (name = self.portName)

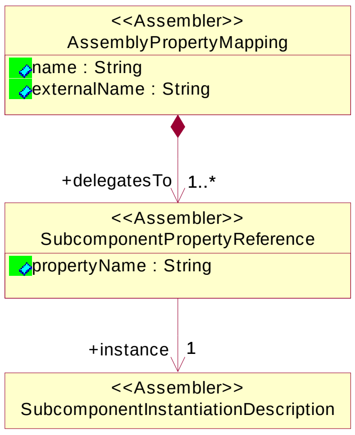
If the **SubcomponentInstantiationDescription** references a package instead of containing it (i.e., if it contains a **ComponentPackageReference**), then the constraint cannot be expressed within the repository but must be checked by the Planner.

7.3.13.5 Semantics

See above.

7.3.14 AssemblyPropertyMapping

7.3.14.1 Description



AssemblyPropertyMapping is part of the **ComponentAssemblyDescription**. It identifies a property of the external component and the subcomponents' properties that it delegates to.

7.3.14.2 Attributes

- `name`: **String**
  Identifies this property mapping within the assembly.

- `externalName`: **String**
  The name of a property of the external component.

7.3.14.3 Associations

- `delegatesTo`: **SubcomponentPropertyReference** [1..*]
  References ports of subcomponents within the assembly that the property is delegated (or propagated) to.

7.3.14.4 Constraints

The `externalName` must match the name of a property of the external component.

7.3.14.5 Semantics

If the component's property is configured, the configuration value will be delegated (propagated) to the specified subcomponent ports in the assembly.

7.3.15 SubcomponentPropertyReference

7.3.15.1 Description

Identifies a property of a component within the assembly or deployment plan that an property of the external component delegates to.

7.3.15.2 Attributes

- `propertyName`: **String**
  The name of the property of that subcomponent instance that the external property is delegated to.

7.3.15.3 Associations

- `instance`: **SubcomponentInstantiationDescription** [1]
  The associated subcomponent instance.

7.3.15.4 Constraints

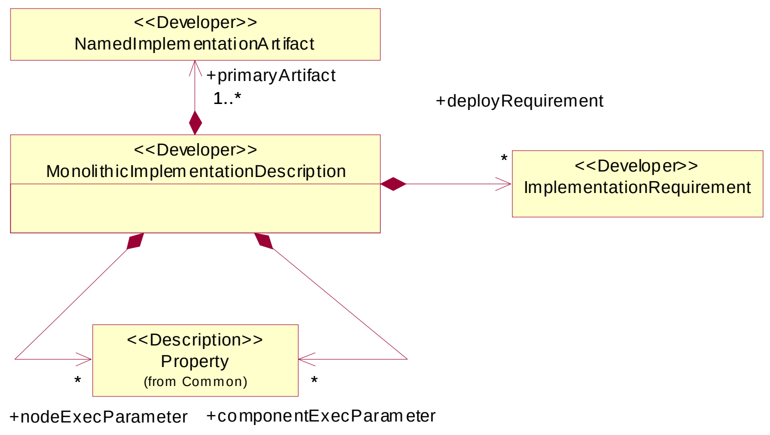
The `propertyName` must match the name of a property of the referenced subcomponent.

7.3.15.5 Semantics

No semantics.

7.3.16 MonolithicImplementationDescription

7.3.16.1 Description



In the case of a monolithic implementation, the **MonolithicImplementationDescription** describes the artifacts that are involved in this implementation. It references primary implementation artifacts (that may then depend on other supporting implementation artifacts). There may be some requirements associated with the monolithic implementation that are matched against node resources during deployment. The author of the implementation may associate some execution parameters with the implementation to be provided to either the execution environment, in the `nodeExecParameter` list, or the primary artifact (e.g., entry point, business logic) in the `componentExecParameter` list.

7.3.16.2 Attributes

No attributes.

7.3.16.3 Associations

- `nodeExecParameter`: **Property** [*]
  Execution parameters that are passed to the target environment. Parameters that are undefined or unknown should generate an exception during the `startLaunch` operation.

- `componentExecParameter`: **Property** [*]
  Execution parameters that are passed to the primary artifact (implementation code). Parameters that are undefined or unknown should generate an exception during the `startLaunch` operation.

- `deployRequirement`: **ImplementationRequirement** [*]
  Requirements that are matched against node resources during planning.

- `primaryArtifact`: **NamedImplementationArtifact** [1..*]
  The primary implementation artifacts.

7.3.16.4 Constraints

The names assigned to primary artifacts must be unique within this context.

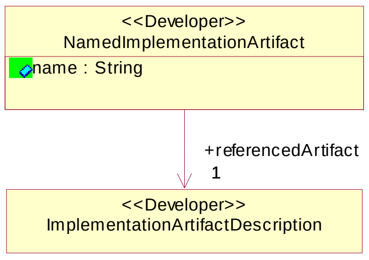
context MonolithicImplementationDescription:
primaryArtifact->forAll (a1, a2 | a1.name = a2.name implies a1 = a2)

7.3.16.5 Semantics

Execution parameters are evaluated by the target environment and may include hints about how to instantiate a component from the implementation artifacts.

7.3.17 NamedImplementationArtifact

7.3.17.1 Description


<<Developer>>
NamedImplementationArtifact
name : String
+referencedArtifact
1
<<Developer>>
ImplementationArtifactDescription

NamedImplementationArtifact is used by **MonolithicImplementationDescription** and **ImplementationArtifactDescription** to assign names to primary artifacts and dependee artifacts, respectively. This information can be used to identify implementation artifacts within the Component Data Model using a “path name” from the top level package downwards.

7.3.17.2 Attributes

- `name`: **String**
  The name assigned to this implementation artifact.

7.3.17.3 Associations

- `referencedArtifact`: **ImplementationArtifactDescription** [1]
  The named implementation artifact.

7.3.17.4 Constraints

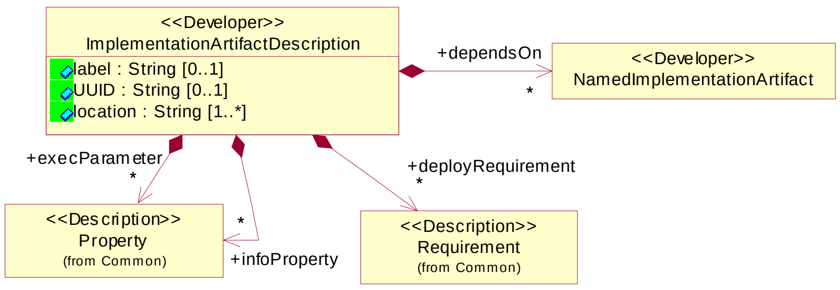
No constraints.

7.3.17.5 Semantics

No semantics.

7.3.18 ImplementationArtifactDescription

7.3.18.1 Description



The **ImplementationArtifactDescription** describes an implementation artifact that is associated with a monolithic component implementation. It contains a reference to the location of the implementation artifact and may refer to other **ImplementationArtifactDescription** elements that this implementation artifact depends on (e.g., shared libraries or support files). The **ImplementationArtifactDescription** may contain deployment requirements that must be matched by a node's resources during deployment. The **ImplementationArtifactDescription** also contains execution parameters that are relevant to the target node's infrastructure (e.g., command line parameters).

7.3.18.2 Attributes

- `label`: **String** [0..1]
  An optional human-readable label.

- `UUID`: **String** [0..1]
  An optional unique identifier for this artifact.

- `location`: **String** [1..*]
  The location of the implementation artifact.

7.3.18.3 Associations

- `dependsOn`: **NamedImplementationArtifact** [*]
  References other **ImplementationArtifactDescription** elements for implementation artifacts that this implementation artifact depends on, assigning names to each.
- `execParameter`: **Property** [*]
  Execution parameters with hints to the target environment about the execution of this implementation artifact.
- `infoProperty`: **Property** [*]
  Non-functional annotation properties.
- `deployRequirement`: **Requirement** [*]
  Requirements that are matched against node resources.

7.3.18.4 Constraints

If the `UUID` field is non-empty, then it must contain a unique identifier for the artifact; artifacts with the same non-empty `UUID` must be identical.

```
context ImplementationArtifactDescription inv:
    self.UUID <> “” implies
        ImplementationArtifactDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

The names assigned to dependee artifacts must be unique within this context.

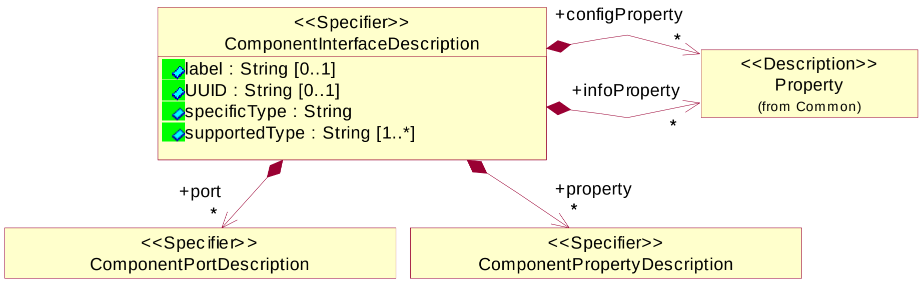
```
context ImplementationArtifactDescription:
    dependsOn->forAll (a1, a2 | a1.name = a2.name implies a1 = a2)
```

7.3.18.5 Semantics

All dependent implementation artifacts have to be installed on (or available to) a node before a component can be instantiated from them.

7.3.19 ComponentInterfaceDescription

7.3.19.1 Description



ComponentInterfaceDescription describes a component's interface. This information can be used by e.g., an assembly tool to verify interface compatibility. The component interface is identified by a unique identifier. A component has properties and ports.

7.3.19.2 Attributes

- `label`: **String** [0..1]
  An optional human-readable label for this interface.

- `UUID`: **String** [0..1]
  An optional unique identifier for this interface.

- `specificType`: **String**
  The most specific type supported by this component interface.

- `supportedType`: **String** [1..*]
  Component interface types supported by this interface (e.g., by inheritance).

7.3.19.3 Associations

- `port`: **ComponentPortDescription** [*]
  Describes the ports of this component interface.

- `property`: **ComponentPropertyDescription** [*]
  Identifies the configurable properties of a component interface.

- `configProperty`: **Property** [*]
  Optional default values for properties.

- `infoProperty`: **Property** [*]
  Non-functional annotation properties.

7.3.19.4 Constraints

The supported types must include the specific type.

context ComponentInterfaceDescription inv:
self.supportedType->includes (self.specificType)

If the `UUID` field is non-empty, then it must contain a unique identifier for the interface; interfaces with the same non-empty `UUID` must be identical.

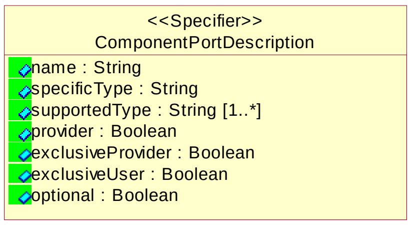
```
context ComponentInterfaceDescription inv:
    self.UUID <> “” implies
        ComponentInterfaceDescription.allInstances->forAll (i |
            i.UUID = self.UUID implies i = self)
```

7.3.19.5 Semantics

Default configuration values can be overridden by assemblies, implementations, packages or package configurations.

7.3.20 ComponentPortDescription

7.3.20.1 Description



ComponentPortDescription describes a port within a component interface. Tools can use this information to e.g., verify port compatibility in connections.

7.3.20.2 Attributes

- `name`: **String**
  The name of the port.
- `specificType`: **String**
  The most specific type supported by the port.
- `supportedType`: **String** [1..*]
  All types supported by this port, including the specific and inherited types. All of the types listed in this attribute are acceptable for a connection.
- `provider`: **Boolean**
  Identifies whether the port acts in the role of provider or user, for any connection attached to it.
- `exclusiveProvider`: **Boolean**
  If set to true, then this port expects that there is at most one provider on the connection that it is an endpoint to.
- `exclusiveUser`: **Boolean**
  If set to true, then this port expects that there is at most one user on the connection that it is an endpoint to.
- `optional`: **Boolean**
  Identifies whether connecting this port is optional or mandatory.

7.3.20.3 Associations

No associations.

7.3.20.4 Constraints

The supported types must include the specific type.

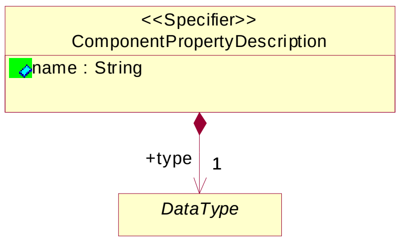
context ComponentPortDescription inv:
    self.supportedType->includes (self.specificType)

7.3.20.5 Semantics

Ports that are endpoints of a connection must support the same type (protocol). Endpoints to a connection can act in the role of either provider or user. For user or provider ports, if `exclusiveProvider` is true, then the connection may not have more than one provider port as an endpoint; if `exclusiveUser` is true, then at most one user port may be an endpoint. For both provider and user ports, if optional is true, then it is not mandatory to use this port as an endpoint to any connection. Thus any implementations would have to function when there was no connection.

7.3.21 ComponentPropertyDescription

7.3.21.1 Description



ComponentPropertyDescription describes a component property.

7.3.21.2 Attributes

- `name`: **String**
  The name of the property.

7.3.21.3 Associations

- `type`: **DataType** [1]
  The data type of this property.

7.3.21.4 Constraints

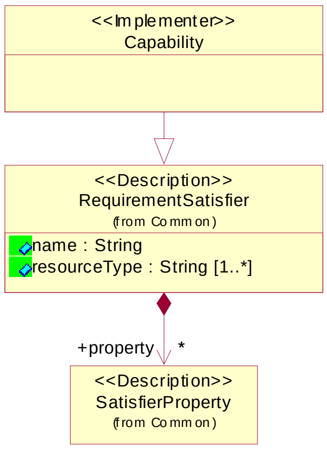
No constraints.

7.3.21.5 Semantics

If this property is configured, the value must conform to the type.

7.3.22 Capability

7.3.22.1 Description



Capability is used within the **ComponentImplementationDescription** to describe an implementation's capabilities, which are matched against selection requirements in **SubcomponentInstantiationDescription** or **PackageConfiguration**. It extends the **RequirementSatisfier** class, but does not add any attributes or associations.

7.3.22.2 Attributes

No additional attributes.

7.3.22.3 Associations

No additional associations.

7.3.22.4 Constraints

Capabilities are not consumable. **SatisfierProperty** elements that are part of **Capability** cannot use the "`Quantity`" or "`Capacity`"**SatisfierPropertyKind** kinds.

```
context Capability inv:
    self.property->forAll (
        kind <> SatisfierPropertyKind::Quantity and
        kind <> SatisfierPropertyKind::Capacity)
```

Capabilities cannot be dynamic. The "`dynamic`" attribute of **SatisfierProperty** elements that are part of a **Capability** cannot be true.

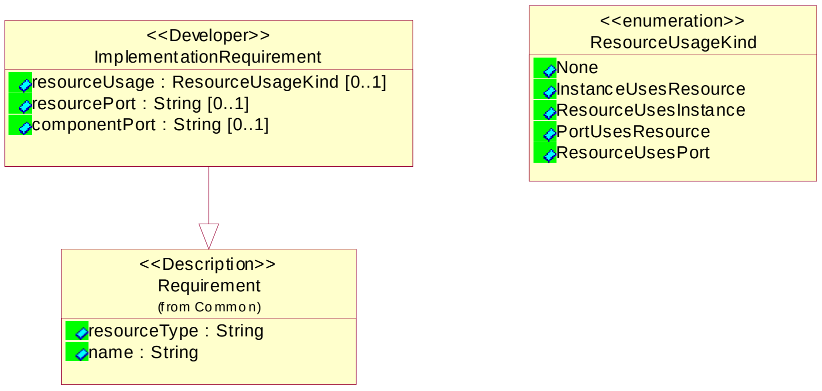
```
context Capability inv:
    self.property->forAll (dynamic = false)
```

7.3.22.5 Semantics

Same as for **RequirementSatisfier**.

7.3.23 ImplementationRequirement

7.3.23.1 Description



The **ImplementationRequirement** class specializes the **Requirement** class with additional attributes which are needed to express how an implementation instance will actually use a resource. This information is ultimately needed by the container to “hook up” the implementation to the resources granted to it. In particular, this enables a component implementation to connect or delegate some of its ports to a resource.

7.3.23.2 Attributes

- `resourcePort`: **String** [0..1]
  When the resource granted to satisfy this requirement is itself a component, and thus the resource value is a component reference, and the component instance needs to use a particular port of the granted resource, this attribute specifies the name of the port of the resource component.

- `componentPort`: **String** [0..1]
  When the resource itself actually acts as a component port of the implementation (essentially delegating the port to the resource), this attribute specifies the name of the port of the component that is being delegated.

- `resourceUsage`: **ResourceUsageKind** [0..1]
  How the resource granted to satisfy this requirement will be used by the container and/or instance. If this attribute is missing, “`None`” is assumed as default value.

7.3.23.3 Associations

None.

7.3.23.4 Constraints

If the value of the `resourceUsage` attribute is “`InstanceUsesResource`,” the `componentPort` attribute must be absent.

```
context ImplementationRequirement:
  self.resourceUsage = “InstanceUsesResource” implies
    self.componentPort->size() = 0
```

If the value of the `resourceUsage` attribute is “`ResourceUsesInstance`,” the `componentPort` attribute must be absent, and the `resourcePort` attribute must be present.

context ImplementationRequirement:
self.resourceUsage = “ResourceUsesInstance” implies
self.componentPort->size() = 0 and
self.resourcePort->size() = 1

If the value of the `resourceUsage` attribute is “`PortUsesResource`,” the `componentPort` attribute must be present.

context ImplementationRequirement:
self.resourceUsage = “PortUsesResource” implies
self.componentPort->size() = 1

If the value of the `resourceUsage` attribute is “`ResourceUsesPort`,” the `componentPort` attribute must be present, and the `resourcePort` attribute must be absent.

context ImplementationRequirement:
self.resourceUsage = “ResourceUsesPort” implies
self.componentPort->size() = 1 and
self.resourcePort->size() = 1

7.3.23.5 Semantics

The choices for the `resourceUsage` attribute are:

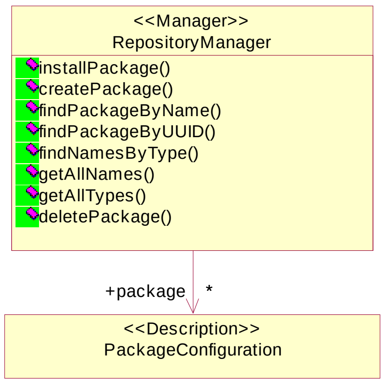
- `InstanceUsesResource`: The resource value is given to the instance when it is created. If the `resourcePort` attribute is present, it indicates that the resource value must be a component reference, and that the port reference obtained from that component reference, using that attribute, should be given to the instance as the value of the resource.
- `ResourceUsesInstance`: The instance provides a reference for use by the resource (i.e., a callback from the resource to the instance). The resource value is a component reference. Thus the `resourcePort` attribute indicates which “uses” port of the resource should use the reference provided by the instance. The instance constructor provides a reference associated with the requirement, to provide to the resource to enable the “callback.”
- `PortUsesResource`: The resource value is used as one of the (provided) ports of the component instance (rather than by the instance itself). The `componentPort` attribute indicates which of the instance’s component ports is being provided by (or delegated to) the resource. The `resourcePort` attribute, if present, indicates that the resource value is a component that provides the reference at one of its ports. Otherwise, the resource value is used directly as the instance’s provided port reference.
- `ResourceUsesPort`: The resource value uses the component port indicated by the `componentPort` attribute, rather than the instance itself implementing that port. Thus the implementation is delegating its “uses” port to the resource. The resource value is a component reference, and the specified port of the resource uses the component port.
- `None`: The resource is not directly used by the instance.

7.4 Component Management Model

The **RepositoryManager** class is placed in the Component subpackage of the Deployment and Configuration package.

7.4.1 RepositoryManager

7.4.1.1 Description



A **RepositoryManager** manages component data. It maintains a collection of **PackageConfiguration** elements. Package installation results in a **PackageConfiguration** existing in the repository under an installer-assigned name. **PackageConfiguration** elements can be installed by value (with the caller supplying the actual data structure) or by location (with the caller supplying a URL). **PackageConfiguration** elements themselves have UUIDs and labels, assigned by the creator of the **PackageConfiguration**. Installation names are unique within a repository. The **RepositoryManager** can provide a list of the names of all **PackageConfiguration** elements or all that support a given component type. It can retrieve **PackageConfiguration** elements by name or UUID. A **PackageConfiguration** in the repository can directly contain a **ComponentPackageDescription** or have indirect references to another **PackageConfiguration**, either in the same repository or in other repositories in the planner's search path. **PackageConfiguration** elements in the repository can be replaced or removed.

7.4.1.2 Operations

- `installPackage` (`installationName`: **String**, `location`: **String**, `replace`: **Boolean**)
  Installs a package in the repository, under the given installation name. If the `replace` parameter is true, an existing **PackageConfiguration** with the same name is replaced. If the `replace` parameter is false, and if a package with the same name exists in the repository, the **NameExists** exception is raised. Raises the **PackageError** exception if an internal error is detected in the package.

- `createPackage` (`installationName`: **String**, `package`: **PackageConfiguration**, `baseLocation`: **String**, `replace`: **Boolean**)
  Installs a **PackageConfiguration** in the repository, assigning a given name. Relative URIs in the `location` or `idlFile` attributes are interpreted according to the `baseLocation`. If the replace parameter is true, replace any existing **PackageConfiguration** with the same name, otherwise raise the **NameExists** exception if a configuration by this name already exists. Raises the **PackageError** exception if an internal error is detected in the package.

- `findPackageByName` (`name`: **String**): **PackageConfiguration**
  Locates a **PackageConfiguration** by name. Raises the **NoSuchName** exception if the name does not exist.

- `findPackageByUUID` (`name`: **String**): **PackageConfiguration**
  Locates a **PackageConfiguration** by UUID. Raises the **NoSuchName** exception if no package with this UUID exists in the repository.

- `getAllNames` (): **String** [*]
  Returns a list of all package configuration names.

- `findNamesByType` (`type`: **String**): **String** [*]
  Finds all configurations of packages that support the given interface type. Returns a sequence of names.

- `getAllTypes` (): **String** [*]
  Returns a sequence of all interface types for which packages are available.

- `deletePackage` (`name`: **String**)
  Deletes the **PackageConfiguration** that is referenced by name. Raises the **NoSuchName** exception if the name does not exist.

7.4.1.3 Associations

- `package`: **PackageConfiguration** [*]
  A **RepositoryManager** manages a number of package configurations.

Constraints

No constraints.

Semantics

The `installPackage` and `createPackage` operations recursively resolve all imported packages: in all **PackageConfiguration** and **SubcomponentInstantiationDescription** elements, **ComponentPackageImport** elements (using the `importedPackage` association) are replaced with **PackageConfiguration** elements (using the `basePackage` association), containing the data that was loaded from the imported package. **PackageConfiguration** elements returned from the `findPackageByName` or `findPackageByUUID` operations do not contain any **ComponentPackageImport** elements.

7.5 Target Data Model

The following classes are part of the Target Data Model. They are placed in the Target subpackage of the Deployment and Configuration package.

The Target Model describes and manages information about the domain into which applications can be deployed. A domain is a set of interconnected nodes with bridges routing between interconnects. Shared resources are logically contained in the domain itself.

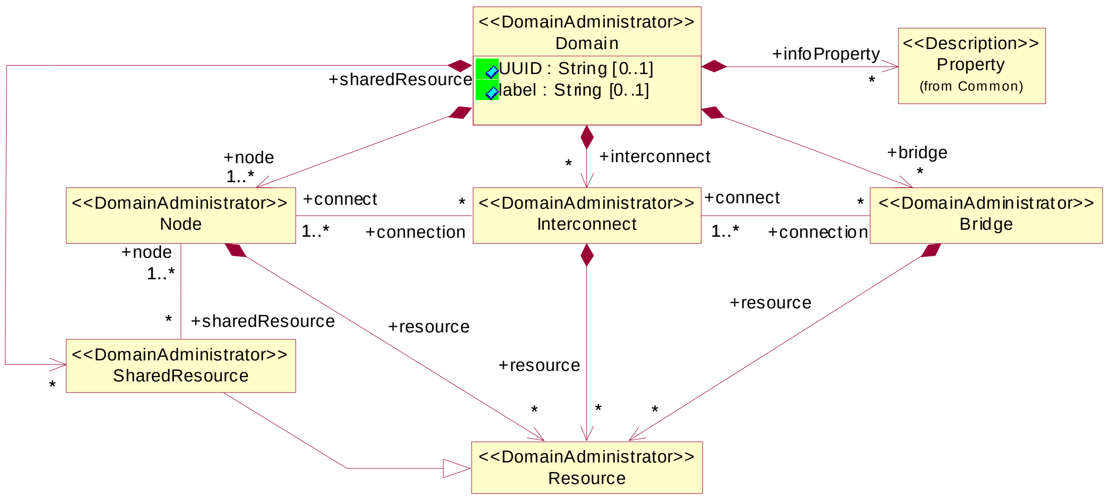


Figure 7.4 - Target Data Model Overview

The top-level entity of target information is the **Domain**. A **Domain** is composed of **Node**, **Interconnect**, **Bridge**, and **SharedResource** elements. Nodes have computational capabilities and are targets for the execution of component instances. Nodes may have resources and be associated with shared resources. While resources belong to the node, a shared resource may be shared between nodes. Artifact requirements must be satisfied by the resources and shared resources of the node that it is to be installed on.

Interconnects provide direct connections among nodes. They have resources but no shared resources. Interconnects are targets for the deployment of connections between components. Connection requirements must be satisfied by the interconnect's resources. Bridges route between interconnects and therefore provide indirect connections between nodes. Connections use some combination of the resources of interconnects and bridges to accomplish the communication between connected ports of instances.

The above is an overview of the Target Data Model. Details about each class in the Target Data Model will be presented in the following sections.

7.5.1 Domain

7.5.1.1 Description

The **Domain** is the container that wraps information about its **Node**, **Interconnect**, **Bridge**, and **SharedResource** elements. It represents the entire target environment.

7.5.1.2 Attributes

- `label`: **String** [0..1]
  An optional human-readable label for the domain.

- `UUID`: **String** [0..1]
  An optional unique identifier for this domain.

7.5.1.3 Associations

- `node`: **Node** [1..*]
  Node elements that belong to the domain.

- `interconnect`: **Interconnect** [*]
  Interconnect elements that provide direct connections between nodes.

- `bridge`: **Bridge** [*]
  Bridge elements route between interconnects and therefore provide indirect connections between nodes.

- `sharedResource`: **SharedResource** [*]
  Shared resources that belong to the domain.

- `infoProperty`: **Property** [*]
  Non-functional annotation properties.

7.5.1.4 Constraints

The top-level elements in a domain all have `name` attributes. These names must be unique within the domain.

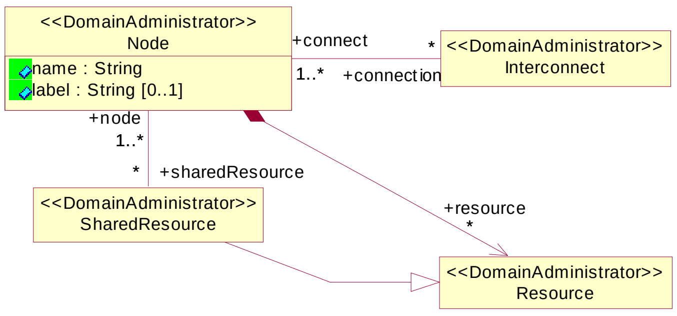
```
context Domain inv:
    let elements = Set {self.node, self.interconnect,
                        self.bridge, self.sharedResource}
    elements->forAll (e1, e2 | e1.name = e2.name implies e1 = e2)
```

7.5.1.5 Semantics

No additional semantics.

7.5.2 Node

7.5.2.1 Description



Nodes are connected to zero or more interconnects that enable components that are instantiated on this node to communicate with components on other nodes. Nodes may own resources and may have access to shared resources that are shared between nodes.

7.5.2.2 Attributes

- `name`: **String**
  The node's name.

- `label`: **String** [0..1]
  An optional human readable label for the node.

7.5.2.3 Associations

- `connection`: **Interconnect** [*]
  A node may be connected to interconnects.

- `resource`: **Resource** [*]
  A node may have resources.

- `sharedResource`: **SharedResource** [*]
  A node may have access to shared resources.

7.5.2.4 Constraints

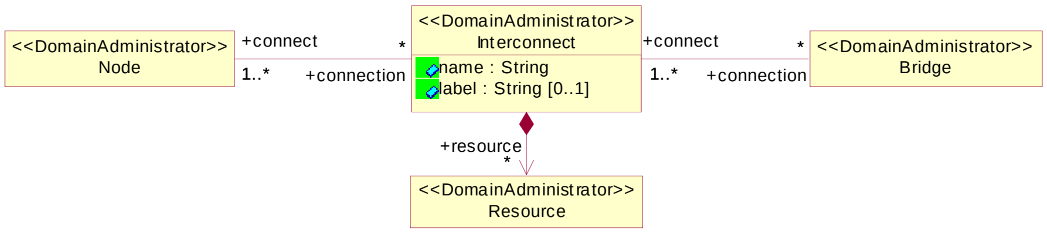
The `name` of the **Node** must be unique within the **Domain** (see above).

7.5.2.5 Semantics

A node's resources and shared resources are matched against implementation requirements.

7.5.3 Interconnect

7.5.3.1 Description



An **Interconnect** provides a shared direct connection between one or more nodes. It has resources, but no shared resources. Resources are matched against a connection's requirements (from the **AssemblyConnectionDescription**) at deployment time.

An **Interconnect** that is attached to only a single node can be used to describe the loopback connection. A loopback connection is implicit; components can always be interconnected locally. Sometimes, it may be useful or necessary to describe the type(s) of available loopback connections (e.g., "shared memory"), or their resources or capabilities (e.g., latency).

7.5.3.2 Attributes

- `name`: **String**
  The interconnect's name.

- `label`: **String** [0..1]
  An optional human-readable label for the interconnect.

7.5.3.3 Associations

- `connect`: **Node** [1..*]
  The nodes that this interconnect provides a connection in between.

- `connection`: **Bridge** [*]
  The bridges that provide connectivity to other interconnects.

- `resource`: **Resource** [*]
  Interconnects have resources.

7.5.3.4 Constraints

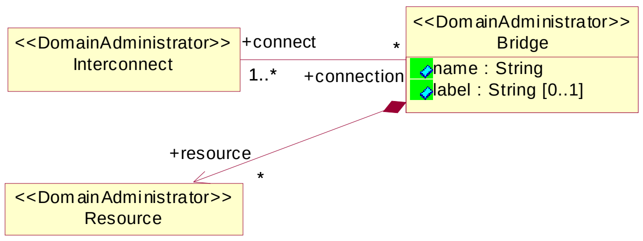
The `name` must be unique within the domain (see above).

7.5.3.5 Semantics

An interconnect's resources are matched against connection requirements.

7.5.4 Bridge

7.5.4.1 Description



A **Bridge** exists between interconnects to describe an indirect communication path between nodes. If a connection is to be deployed between components that are instantiated on nodes that are not directly connected, therefore requiring bridging, the connection's requirements must be satisfied by the resources of each interconnect and bridge in between.

7.5.4.2 Attributes

- `name`: **String**
  The bridge's name.

- `label`: **String** [0..1]
  An optional human-readable label for this bridge.

7.5.4.3 Associations

- `connect`: **Interconnect** [1..*]
  The interconnects that this bridge provides connectivity between.

- `resource`: **Resource** [*]
  Bridges have resources.

7.5.4.4 Constraints

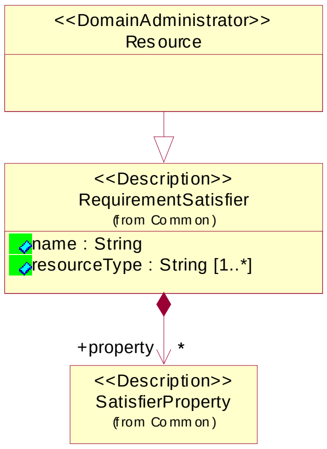
The `name` must be unique within the domain (see above).

7.5.4.5 Semantics

A bridge's resources are matched against connection requirements.

7.5.5 Resource

7.5.5.1 Description



Resource elements express **Node**, **Interconnect**, and **Bridge** features within the target environment. They are matched against implementation requirements at planning time. **Resource** extends the **RequirementSatisfier** class, but does not add any attributes or associations.

7.5.5.2 Attributes

No additional attributes.

7.5.5.3 Associations

No additional associations.

7.5.5.4 Constraints

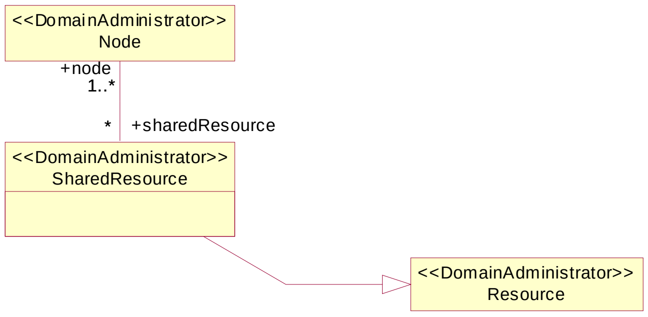
The name of a resource must be unique within the container.

7.5.5.5 Semantics

Same as for **RequirementSatisfier**.

7.5.6 SharedResource

7.5.6.1 Description



Shared resources are resources that are shared between nodes. They are semantically equivalent to “normal” resources; however, the planner must make sure that a shared resource is not exhausted by using it from multiple nodes in parallel.

7.5.6.2 Attributes

No additional attributes.

7.5.6.3 Associations

- `nodes`: **Node** [1..*]
  The nodes that have access to this **SharedResource**.

7.5.6.4 Constraints

The `name` of the **SharedResource** must be unique within the domain (see above).

7.5.6.5 Semantics

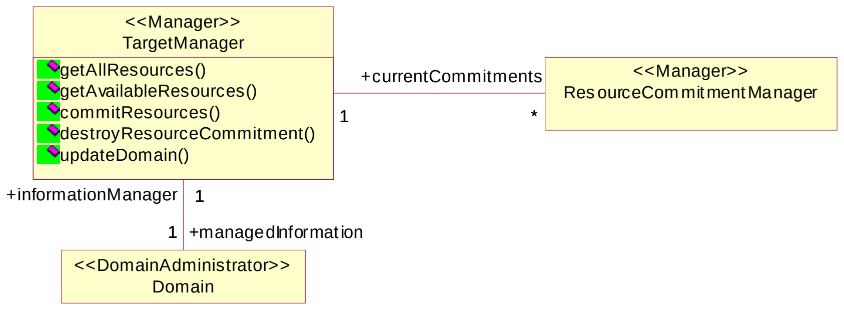
Same as for **Resource** and for **RequirementSatisfier**.

7.6 Target Management Model

The **TargetManager** and **DomainUpdateKind** classes are placed in the Target subpackage of the Deployment and Configuration package.

7.6.1 TargetManager

7.6.1.1 Description



The **TargetManager** provides information about the **Domain** using the Target Data Model and tracks resource usage within the domain. Note that this specification limits the features of the **TargetManager** to those related to deployment. While domains and nodes may have properties, exposing an interface to configure them is out of the scope of this specification.

7.6.1.2 Operations

- `getAllResources` (): **Domain**
  Returns static information about the domain, with resources at their full capacity.

- `getAvailableResources` (): **Domain**
  Returns online information about the domain; resources will reflect their remaining capacity.

- `createResourceCommitment` (`manager`: **ResourceAllocation** [*]): **ResourceCommitmentManager**
  Creates a new resource commitment management object, with an initial set of resource commitments. This is equivalent to creating a **ResourceCommitmentManager** with an empty initial set of resource commitments, and then passing the set of resources to its `commitResources` operation. Raises the **ResourceCommitmentFailure** exception if one of the resources cannot be committed.

- `destroyResourceCommitment` (`resources`: **ResourceCommitmentManager**)
  Destroys a resource commitment management object, releasing all remaining resources that were committed but not previously released.

- `updateDomain` (`elements`: **String** [1..*], `domainSubset`: **Domain**, `updateKind`: **DomainUpdateKind**)
  Updates **Domain** information within the **TargetManager**. The elements parameter identifies the names of nodes, interconnects, bridges, and shared resources to be updated. The `domainSubset` contains information about the elements and their associations. The `updateKind` identifies whether the elements are to be added, deleted, or updated.

7.6.1.3 Associations

- `managedInformation`: **Domain** [1]
  A **TargetManager** manages information about a single **Domain**.

- `currentCommitments`: **ResourceCommitmentManager** [*]
  The current set of resource commitment management objects.

7.6.1.4 Constraints

No constraints

7.6.1.5 Semantics

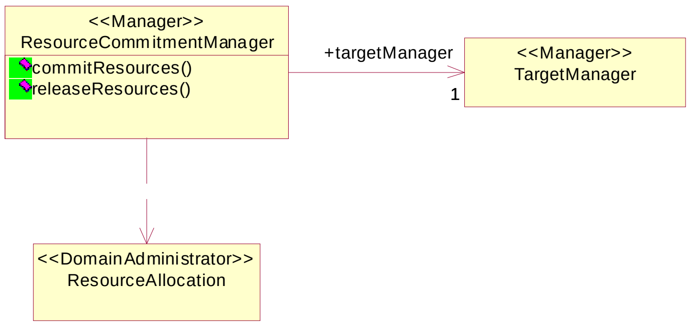
Resources are centrally managed by the **TargetManager**, it is assumed that the **TargetManager** has complete knowledge of available resources. This implies worst-case resource allocation (implementations may not use any more resources than declared), and that resources may not be used by processes outside of this specification.

Planning for deployment can happen “online” or “offline.” In the online case, the planner considers the presently available resources that are returned from **getAvailableResources**. In offline planning, the planner considers all available resources in order to plan for an application that is to be deployed into an “empty” target environment.

It may be necessary to serialize access to resource information and planning using means beyond the scope of this specification, in order to avoid race conditions in online planning – otherwise resources might be committed elsewhere while planning, or multiple plans might end up competing for the same resources.

7.6.2 ResourceCommitmentManager

7.6.2.1 Description



The **ResourceCommitmentManager** interface manages a set of resources that are allocated, e.g., for the execution of an application. Resources can be individually committed and released as necessary. At the end of its life cycle, all remaining committed resources are released automatically.

7.6.2.2 Operations

- `commitResources` (`resources`: **ResourceAllocation** [*])
  Adds a set of resources to the current list of committed resources. Raises the **ResourceCommitmentFailure** exception if a resource cannot be committed, e.g., due to resource exhaustion.

- `releaseResources` (`resources`: **ResourceAllocation** [*])
  Releases a set of previously committed resources. Raises the **ResourceCommitmentFailure** exception when trying to release more resources than were committed.

7.6.2.3 Associations

- `targetManager`: **TargetManager** [1]
  A **ResourceCommitmentManager** cooperates with a **TargetManager** to evaluate and manipulate the set of available resources in the **Domain** it manages.

7.6.2.4 Constraints

Each **ResourceAllocation** element must identify an element (a **Node**, **Interconnect**, or **Bridge**) within the **Domain** that is managed by the associated **TargetManager**, and a **Resource** or **SharedResource** that is available to the element.

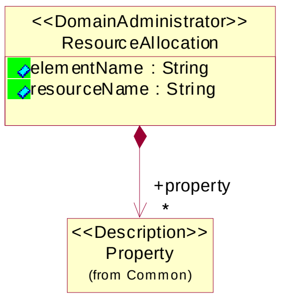
7.6.2.5 Semantics

Committing a **ResourceAllocation** involves comparing the allocated properties against the available properties of the **Resource**'s **SatisfierProperty**, according to its **SatisfierPropertyKind**, and "substracting" the committed values from the available values, if applicable.

Releasing a **ResourceAllocation** "adds" the values that are being released to the available values, and "subtracts" the values that are being released from the set of committed values, if applicable.

7.6.3 ResourceAllocation

7.6.3.1 Description



A **ResourceAllocation** element identifies a resource within the **Domain** that is allocated from, and the amount that is being allocated.

7.6.3.2 Attributes

- `elementName`: **String**
  The name of a top-level element in a **Domain**, i.e., the name of a **Node**, **Interconnect**, or **Bridge**.

- `resourceName`: **String**
  The name of a **Resource** or **SharedResource** that is accessible to the **Node**, **Interconnect**, or **Bridge** that is identified by the `elementName` attribute.

7.6.3.3 Associations

- property: **Property** [*]
  The properties of the resource to be allocated or verified against the **Resource**'s properties. Their values must be of the appropriate type, according to the **SatisfierProperty**'s **SatisfierPropertyKind** attribute. For `Quantity`, the value must be the integer value 1. For `Capacity`, is the ordinal that is allocated. For the other kinds, the value must be of the same type as the value of the **SatisfierProperty**.

7.6.3.4 Constraints

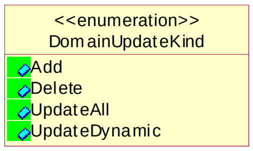
No constraints.

7.6.3.5 Semantics

No semantics.

7.6.4 DomainUpdateKind

7.6.4.1 Description



The **DomainUpdateKind** is an enumeration used as a parameter to the updateDomain operation of the **TargetManager** to describe how **Domain** information is to be updated.

7.6.4.2 Attributes

No attributes.

7.6.4.3 Associations

No associations.

7.6.4.4 Constraints

No constraints.

7.6.4.5 Semantics

If the `Add` kind is used, then information about nodes, interconnects, bridges, and shared resources is added to the **Domain**. In case of `Delete`, information is removed. In case of `UpdateAll`, existing information about the full capacity of resources is updated. In case of `UpdateDynamic`, the update contains new dynamic values for **SatisfierProperty** elements that are dynamic.

7.7 Execution Data Model

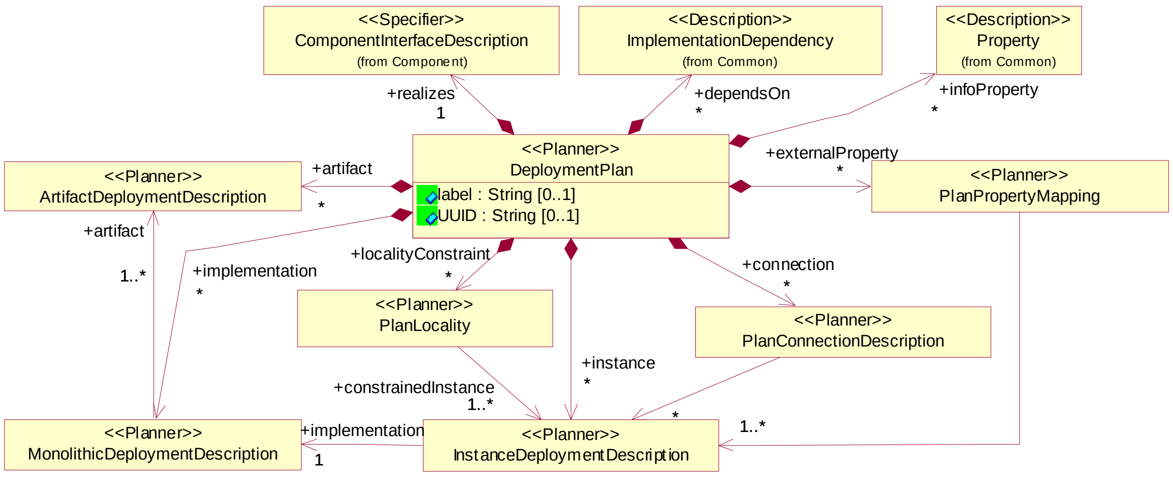
The following classes are part of the Execution Data Model. They are placed in the Execution subpackage of the Deployment and Configuration package.

Before deployment can occur, decisions must be made about the implementations to select (if multiple implementations exist in a package) and where to deploy each monolithic component implementation. All information about an application's deployment is collected in a **DeploymentPlan**. This plan can be used transiently (i.e., executed right away), or it may be stored to avoid the overhead of planning in the future. The **DeploymentPlan** can be used by an **ExecutionManager** to create a specific factory object for the application. A **DeploymentPlan** is “standalone” in that it does not necessarily refer to a repository, only to artifacts, which, depending on the implementation, may or may not reside in the repository.

Details about each class in the Execution Data Model will be presented in the following sections.

7.7.1 DeploymentPlan

7.7.1.1 Description



The **DeploymentPlan** contains information about artifacts that are part of the deployment (**ArtifactDeploymentDescription**), how to create component instances from artifacts (**MonolithicDeploymentDescription**), and where to instantiate them (**InstanceDeploymentDescription**). It then contains information about connections between them (**AssemblyConnectionDescription**) and about the mapping of external properties. It finally contains information about the component interface that is realized by the application. The **DeploymentPlan** is analogous to the **ComponentAssemblyDescription** in the Component Data Model. In fact, the **DeploymentPlan** can be seen as a flattened assembly (without recursion). In the plan, all assemblies have been recursively replaced by their white-box representation, and concrete implementations have been chosen for each subcomponent. All that remains are the leaf nodes, i.e., components that have a monolithic implementation. It also may contain additional locality information (PlanLocality) associated with the **InstanceDeploymentDescription**.

To avoid redundancy, a Planner can compare the identity of artifacts and component implementations for identity (using their `UUID` attributes) and then share **ArtifactDeploymentDescription** and **MonolithicDeploymentDescription** elements.

7.7.1.2 Attributes

- `label`: **String** [0..1]
  Users may optionally assign a human readable label to a **DeploymentPlan**.

- `UUID`: **String** [0..1]
  A unique identifier for this **DeploymentPlan**.

7.7.1.3 Associations

- `artifact`: **ArtifactDeploymentDescription** [*]
  Implementation artifacts related to the deployment.

- `implementation`: **MonolithicDeploymentDescription**
  Component implementations used in the deployment.

- `instance`: **InstanceDeploymentDescription** [*]
  Component instances that are to be created.

- `localityConstraint`: **PlanLocality** [*]
  Describes process locality constraints for component instances.

- `connection`: **PlanConnectionDescription** [*]
  Connections that are to be made between the component instances, the application's external ports, or external locations.

- `externalProperty`: **PlanPropertyMapping** [*]
  Maps the application's external properties to properties of component instances.

- `realizes`: **ComponentInterfaceDescription** [1]
  The component interface implemented by the application.

- `dependsOn`: **ImplementationDependency** [*]
  Implementations of these interfaces must be executing in the target environment before deploying this plan is possible. Copied from the **ComponentImplementationDescription** element.

- `infoProperty`: **Property** [*]
  Non-functional annotation properties.

7.7.1.4 Constraints

The top-level elements in a **DeploymentPlan** all have name attributes. These names must be unique within the plan.

```
context DeploymentPlan inv:
    let elements = Set {self.artifact, self.implementation,
                  self.instance, self.connection,
                  self.externalProperty}
    elements.forAll (e1, e2 | e1.name = e2.name implies e1 = e2)
```

7.7.1.5 Semantics

The **DeploymentPlan** is a self-contained piece of information that contains all necessary data about the deployment of an application to a specific target environment.

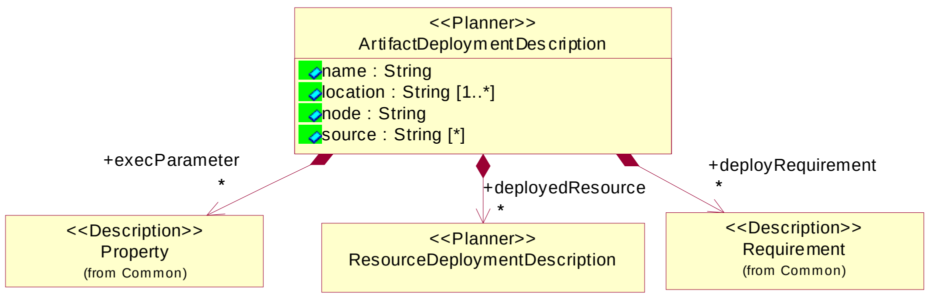
The deployment engine that is part of the **ExecutionManager** or **ApplicationManager** traverses the instances; for each instance, it determines the implementation and its artifacts, which need to be installed on a target node prior to component instantiation. All artifacts used in this process are marked. The deployment engine then traverses the artifacts and processes all "leftover" **ArtifactDeploymentDescription** elements; these may be additional artifacts included by the Planner to take care of special conditions in the target environment.

The deployment engine then proceeds to create the component instances and interconnects them.

The interface information is used so that the application can present this interface to the user. (This is detailed by platform specific models.) Default values for properties (the **configProperty** elements of the **ComponentInterfaceDescription**) are not needed in the plan and ignored by the deployment engine; a Planner may decide not to copy them into the plan.

7.7.2 ArtifactDeploymentDescription

7.7.2.1 Description



ArtifactDeploymentDescription describes an artifact that is to be deployed as part of the plan. It mirrors the **ImplementationArtifactDescription** from the component data model. To avoid redundancy, this element can be shared among **InstanceDeploymentDescription** elements, should component instances use the same artifact more than once, either on the same node, or if the artifact has no node-specific resource requirements. A Planner can compare artifacts for identity using the **UUID** attribute of the **ImplementationArtifactDescription** element. **ArtifactDeploymentDescription** describes the installation of a single implementation artifact on a node as part of component instantiation. It contains a URL pointing to the **ImplementationArtifact**. Execution parameters and deployment requirements are copied from the **ImplementationArtifactDescription**.

7.7.2.2 Attributes

- `name`: **String**
  A unique identifier for this element of the **DeploymentPlan**.

- `location`: **String** [1..*]
  The location where the artifact can be loaded from. Copied from the **ImplementationArtifactDescription**.

- `node`: **String**
  The name of the node where the artifact is to be installed. If blank, the node is implied by the **InstanceDeploymentDescription** parent.

- `source`: **String** [*]
  Identifies the **ImplementationArtifactDescription** elements that caused this artifact to be part of the deployment.

7.7.2.3 Associations

- `execParameter`: **Property** [*]
  Execution parameters, copied from the **ImplementationArtifactDescription**.

- `deployRequirement`: **Requirement** [*]
  Deployment requirements, copied from the **ImplementationArtifactDescription**.

- `deployedResource`: **ResourceDeploymentDescription** [*]
  The resources chosen to satisfy the requirements of the implementation as specified in the **ImplementationArtifactDescription**.

7.7.2.4 Constraints

No constraints.

7.7.2.5 Semantics

The deployment requirements carry information about the resources used by this implementation artifact, so that they can be committed by the **TargetManager** (presumably via the **ExecutionManager**).

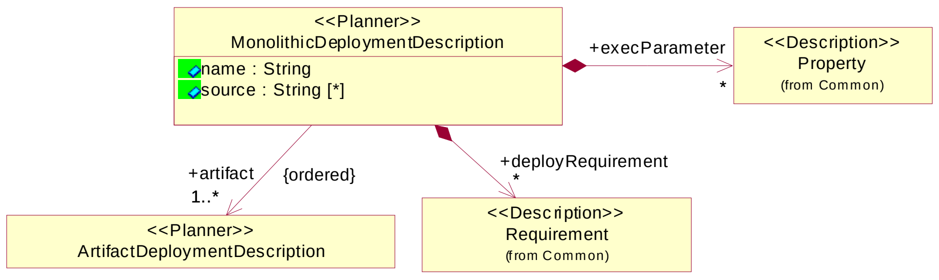
Usually, the `node` attribute is the empty string, so that artifacts will be deployed on the node where a component is to be instantiated as implied by the **InstanceDeploymentDescription**. The attributed is included here for the exotic case that special artifacts need to be installed in the target environment. In that case, the Planner would add **ArtifactDeploymentDescription** elements to the plan that are unrelated to component instances.

A Planner may compose a human readable value for the `source` attribute by combining the `name` attributes from **PackageConfiguration**, **PackagedComponentImplementation**, **SubcomponentInstantiationDescription** and **NamedImplementationArtifact** elements, describing a “path” of the artifact’s origins in the Component Data Model. The `source` attribute may have more than one element, since **ArtifactDeploymentDescription** elements may be shared among instance deployments, if the same implementation artifact is part of multiple component implementations. In case of an error, a user can use this information to track the problem.

A Planner must generate a `name` that is unique among the top-level elements in a **DeploymentPlan**.

7.7.3 MonolithicDeploymentDescription

7.7.3.1 Description



MonolithicDeploymentDescription describes the deployment of a component as part of the plan. It mirrors the **MonolithicImplementationDescription** from the component data model. If the same component instance is deployed more than once, either on the same node, or using only artifacts with no node-specific resource requirements, a **MonolithicDeploymentDescription** can be shared by multiple **InstanceDeploymentDescription** elements. A Planner can compare monolithic implementations for identity using the **UUID** attribute of the **ComponentImplementationDescription**. The **MonolithicDeploymentDescription** references **ArtifactDeploymentDescription** elements for all artifacts that are part of the deployment. The execution parameters and deployment requirements are copied from the **MonolithicImplementationDescription**.

7.7.3.2 Attributes

- `name`: **String**
  A unique identifier for this element of the **DeploymentPlan**.

- `source`: **String** [*]
  Identifies the **MonolithicImplementationDescription** elements that caused this component to be part of the deployment.

7.7.3.3 Associations

- `artifact`: **ArtifactDeploymentDescription** [*]
  The implementation artifacts that are part of this monolithic component implementation.

- `execParameter`: **Property** [*]
  Execution parameters, copied from the **MonolithicImplementationDescription**.

- `deployRequirement`: **Requirement** [*]
  Deployment requirements, copied from the **MonolithicImplementationDescription**.

7.7.3.4 Constraints

No constraints.

7.7.3.5 Semantics

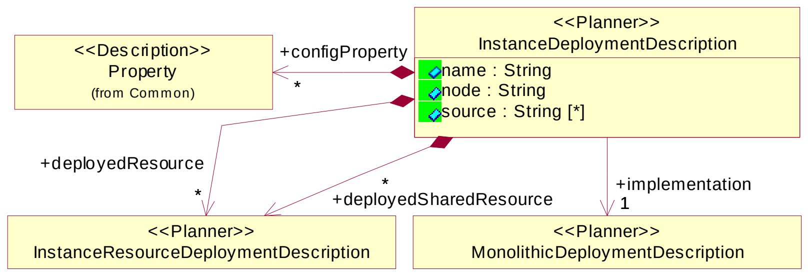
The artifacts referenced here represent a depth-first traversal of the primary artifacts from the **MonolithicImplementationDescription** and their dependency. A depth-first traversal ensures that all dependees can be installed before the dependent artifacts.

A Planner may compose a human readable value for the `source` attribute by combining the `name` attributes from **PackageConfiguration**, **PackagedComponentImplementation** and **SubcomponentInstantiationDescription** elements, describing a "path" of the component implementation's origins in the Component Data Model. The `source` attribute may have more than one element, since **MonolithicImplementationDescription** elements may be shared among instance deployments, if the same component implementation is deployed more than once. In case of an error, a user can use this information to track the problem.

A Planner must generate a `name` that is unique among the top-level elements in a **DeploymentPlan**.

7.7.4 InstanceDeploymentDescription

7.7.4.1 Description



InstanceDeploymentDescription contains the information that is necessary in order to deploy a single component instance. It references a **MonolithicDeploymentDescription** and includes the name of the node where the component is to be instantiated. It then contains properties that are used to configure the component instance.

7.7.4.2 Attributes

- `name`: **String**
  A unique identifier for this element of the **DeploymentPlan**.

- `node`: **String**
  The name of the node where the component is to be instantiated.

- `source`: **String**
  Identifies the **MonolithicImplementationDescription** element that caused this component to be part of the deployment.

7.7.4.3 Associations

- `implementation`: **MonolithicDeploymentDescription** [1]
  The component that is to be instantiated.

- `configProperty`: **Property** [*]
  Properties to configure the component instance after instantiation.

- `deployedResource`: **InstanceResourceDeploymentDescription** [*]
  The resources chosen to satisfy the requirements of the implementation as specified in the **MonolithicImplementationDescription**, which were satisfied by a node's own (not shared) resources.

- `deployedSharedResource`: **InstanceResourceDeploymentDescription** [*]
  The resources chosen to satisfy the requirements of the implementation as specified in the **MonolithicImplementationDescription**, which were satisfied by shared resources that are available to the node.

7.7.4.4 Constraints

No constraints.

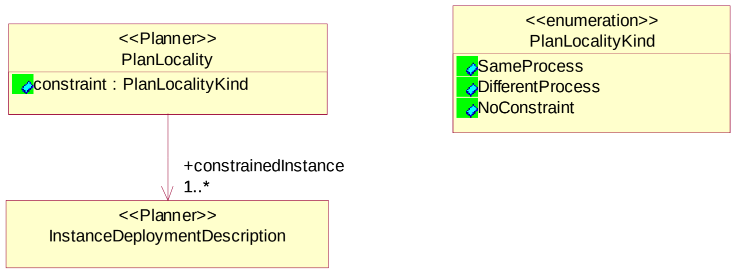
7.7.4.5 Semantics

A Planner may compose a human readable value for the `source` attribute by combining the `name` attributes from **PackageConfiguration**, **PackagedComponentImplementation**, and **SubcomponentInstantiationDescription** elements, describing a "path" of the instance's origins in the Component Data Model. In case of an error, a user can use this information to track the problem.

A Planner must generate a `name` that is unique among the top-level elements in a **DeploymentPlan**.

7.7.5 PlanLocality

7.7.5.1 Description



The **PlanLocality** element allows for defining process collocation or process separation of two or more component instances specified by a **DeploymentPlan**. The node manager must consider these constraints when creating component instances.

The values of the **PlanLocality** element are derived from the **Locality** element of the **ComponentAssemblyDescription** at planning time.

7.7.5.2 Attributes

- `constraint`: **PlanLocalityKind**
  The value of this attribute identifies the kind of the **PlanLocality** constraint.

7.7.5.3 Associations

- `constrainedInstance`: **InstanceDeploymentDescription** [1..*]
  References the component instances that the locality constraint applies to.

7.7.5.4 Constraints

No constraints.

7.7.5.5 Semantics

See the description of the **PlanLocalityKind** class for the semantics of locality constraint options.

At execution time, the node manager evaluates process locality constraints, and assigns constrained instances to the same process (`SameProcess`) or to different processes (`DifferentProcess`).

7.7.6 PlanLocalityKind

7.7.6.1 Description

See above.

7.7.6.2 Attributes

No attributes.

7.7.6.3 Associations

No associations.

7.7.6.4 Constraints

No constraints.

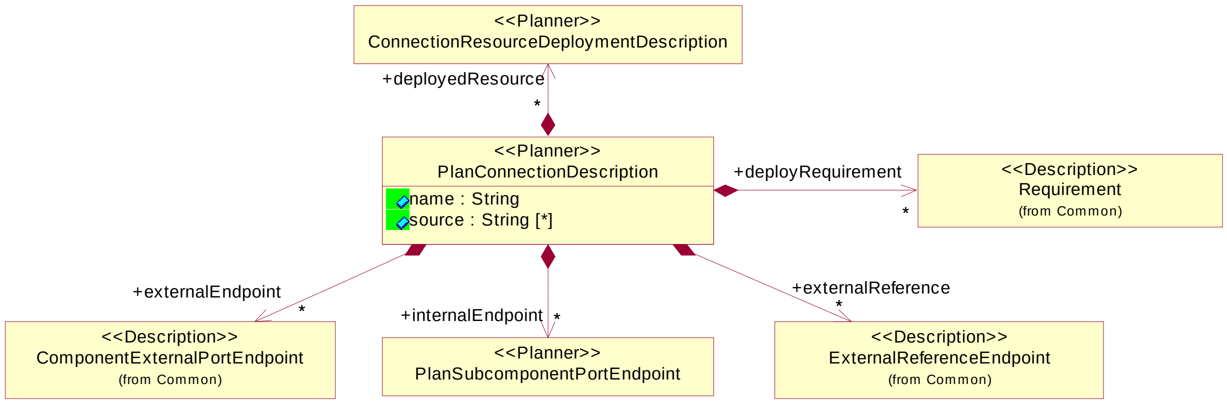
7.7.6.5 Semantics

The choices for locality constraints are:

- `SameProcess`: This value specifies that the component instances that the **PlanLocality** element refers to shall be started in the same process by the deployment engine.
- `DifferentProcess`: This value specifies that the component instances that the **PlanLocality** element refers to shall be started in different processes by the deployment engine.
- `NoConstraint`: This value specifies that there is no locality constraint for the component instances that the **PlanLocality** element refers to. The purpose of this special value is to enable to switch locality constraints temporarily off without losing the structure of the **DeploymentPlan**, i.e., the **PlanLocality** class instance with its associations to **InstanceDeploymentDescription** can be kept for later reuse but has currently no impact on the execution of the **DeploymentPlan**.

7.7.7 PlanConnectionDescription

7.7.7.1 Description



The **PlanConnectionDescription** describes a connection that is to be made among ports within the application that is being deployed. It is analogous to the **AssemblyConnectionDescription** that describes a connection within an assembly. The **ComponentExternalPortEndpoint** and **ExternalReferenceEndpoint** elements are reused from the Component Data Model.

7.7.7.2 Attributes

- `name`: **String**
  A unique identifier for this element of the **DeploymentPlan**.
- `source`: **String** [*]
  Identifies the **AssemblyConnectionDescription** elements that were combined into this **PlanConnectionDescription**.

7.7.7.3 Associations

- `deployRequirement`: **Requirement** [*]
  Connection requirements; the sum of all deployment requirements of all **AssemblyConnectionDescription** elements that are involved in this connection.

- `externalEndpoint`: **ComponentExternalPortEndpoint** [*]
  Identifies a port of the component that is implemented by the application as an endpoint of this connection.

- `internalEndpoint`: **PlanSubcomponentPortEndpoint** [*]
  Identifies a port of a component within the application as an endpoint of this connection.

- `externalReference`: **ExternalReferenceEndpoint** [*]
  Identifies a location outside the application as an endpoint of this connection.

- `deployedResource`: **ConnectionResourceDeploymentDescription** [*]
  The resources chosen to satisfy the requirements of the connection as specified in the **AssemblyConnectionDescription**.

7.7.7.4 Constraints

The number of endpoints must be larger than one.

7.7.7.5 Semantics

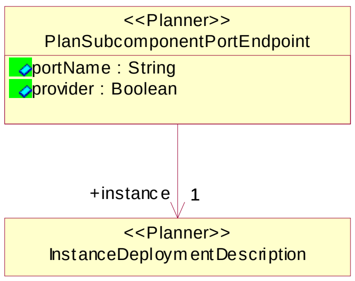
During application launch, a connection between all endpoints will be established.

A Planner may compose a human readable value for the `source` attribute by combining the `name` attributes from **PackageConfiguration**, **PackagedComponentImplementation**, **SubcomponentInstantiationDescription**, and **AssemblyConnectionDescription** elements, describing a "path" of the connection's origins in the Component Data Model. The `source` attribute may have more than one element, since a connection in the "flattened" plan might be a combination of multiple connection segments on different levels of the assembly hierarchy. In case of an error, a user can use this information to track the problem.

A Planner must generate a `name` that is unique among the top-level elements in a **DeploymentPlan**.

7.7.8 PlanSubcomponentPortEndpoint

7.7.8.1 Description



Identifies a port of a component within the application as an endpoint of the connection described by the **PlanConnectionDescription** that this element is contained in.

7.7.8.2 Attributes

- `portName`: **String**
  The name of the port of the associated component instance that is to be an endpoint of this connection.

- `provider`: **String**
  Identifies whether the port is a provider or user port.

7.7.8.3 Associations

- `instance`: **InstanceDeploymentDescription** [1]
  The associated component instance.

7.7.8.4 Constraints

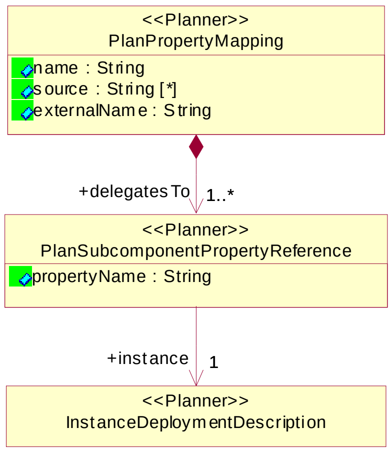
The port name must be valid for the referenced component.

7.7.8.5 Semantics

See above.

7.7.9 PlanPropertyMapping

7.7.9.1 Description



PlanPropertyMapping is part of the **DeploymentPlan**. It identifies a property of the component that this application is implementing and the subcomponents' properties that it delegates to.

7.7.9.2 Attributes

- `name`: **String**
  A unique identifier for this element of the **DeploymentPlan**.

- `source`: **String** [*]
  Identifies the **AssemblyPropertyMapping** elements that were combined into this **PlanPropertyMapping**.

- `externalName`: **String**
  The name of a property of the component that the application is implementing.

7.7.9.3 Associations

- `delegatesTo`: **PlanSubcomponentPropertyReference** [1..*]
  References ports of subcomponents within the application that the property is delegated (or propagated) to.

7.7.9.4 Constraints

The `externalName` must match the name of a property of the component that the assembly is implementing.

7.7.9.5 Semantics

A Planner may compose a human readable value for the `source` attribute by combining the `name` attributes from **PackageConfiguration**, **PackagedComponentImplementation**, **SubcomponentInstantiationDescription**, and **AssemblyPropertyMapping** elements, describing a "path" of the mapping's origins in the Component Data Model. The `source` attribute may have more than one element, since a mapping in the "flattened" plan might be a combination of multiple mapping "segments" on different levels of the assembly hierarchy. In case of an error, a user can use this information to track the problem.

A Planner must generate a `name` that is unique among the top-level elements in a **DeploymentPlan**.

7.7.10 PlanSubcomponentPropertyReference

7.7.10.1 Description

Identifies a property of a subcomponent within the deployment plan that an external property of the component that the application implements delegates to.

7.7.10.2 Attributes

- `propertyName`: **String**
  The name of the property of the associated component instance that the external property is delegated to.

7.7.10.3 Associations

- `instance`: **InstanceDeploymentDescription** [1]
  The associated component instance.

7.7.10.4 Constraints

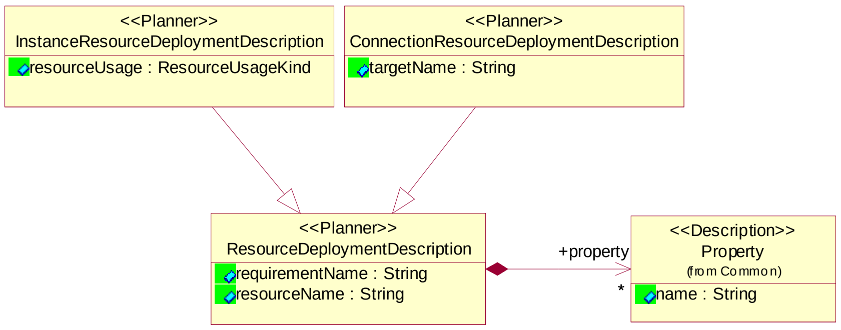
The `propertyName` must match the name of a property of the associated component.

7.7.10.5 Semantics

No semantics.

7.7.11 ResourceDeploymentDescription

7.7.11.1 Description



ResourceDeploymentDescription contains information about how a requirement of a monolithic implementation instance, artifact, or connection was satisfied by indicating the requirement, the resource, and how the resource will be used to satisfy the requirement.

7.7.11.2 Attributes

- `requirementName`: **String**
  The name of the requirement being satisfied. This is not a model association with the **Requirement** class because that information does not necessarily need to be in the plan. This attribute will enable the node, containe, and/or implementation instance to know which resource was used to satisfy each of its specified requirements.

- `resourceName`: **String**
  The name of the target domain entity's resource chosen to satisfy the requirement.

7.7.11.3 Associations

- `property`: **Property** [*]
  The aspects of the resource actually allocated, if any. The property values are of the appropriate types according to their matched **SatisfierProperty**'s **SatisfierPropertyKind** attribute. For `Quantity`, it is the ordinal allocated. For `Allocation`, it is the allocated capacity, for `Selection`, it is the matched string. For others, it is the value of the matched property.

7.7.11.4 Constraints

None.

7.7.11.5 Semantics

None.

7.7.12 InstanceResourceDeploymentDescription

7.7.12.1 Description

InstanceResourceDeploymentDescription specializes **ResourceDeploymentDescription** to describe resources allocated for instances. Associated with and contained by an **InstanceDeploymentDescription**.

7.7.12.2 Attributes

- `resourceUsage`: **ResourceUsageKind**
  How the resource will be used to satisfy the requirement (copied from the original **ImplementationRequirement**).

7.7.12.3 Associations

None.

7.7.12.4 Constraints

None.

7.7.12.5 Semantics

An **InstanceResourceDeploymentDescription** is always used in the context of an **InstanceDeploymentDescription**, which identifies the node that a component instance is deployed on. This node provides scope for the `resourceName`.

7.7.13 ConnectionResourceDeploymentDescription

7.7.13.1 Description

ConnectionResourceDeploymentDescription specializes **ResourceDeploymentDescription** to describe resources allocated for connections. Associated with and contained by a **PlanConnectionDescription**.

7.7.13.2 Attributes

- `targetName`: **String**
  The name of the target domain entity from which the resource was allocated (i.e., the name of an **Interconnect** or **Bridge**), to provide scope for the `resourceName`. This attribute is required because connections may traverse multiple bridges and interconnects.

7.7.13.3 Associations

None.

7.7.13.4 Constraints

None.

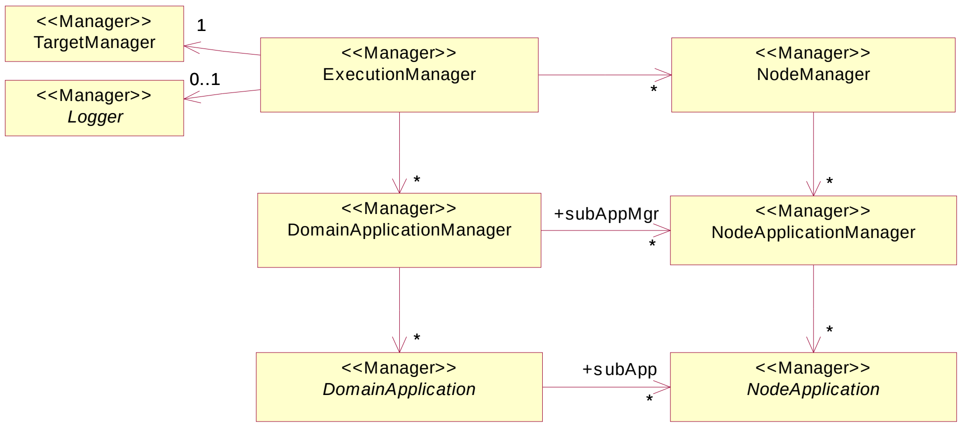
7.7.13.5 Semantics

None.

7.8 Execution Management Model

The following classes are part of the Execution Management Model. They are placed in the Execution subpackage of the Deployment and Configuration package.

7.8.1 Execution Management Model Overview



<<Manager>> TargetManager
<<Manager>> Logger
<<Manager>> ExecutionManager
<<Manager>> NodeManager
<<Manager>> DomainApplicationManager
<<Manager>> NodeApplicationManager
<<Manager>> DomainApplication
<<Manager>> NodeApplication
1
0..1
*
+subAppMgr
+subApp

Figure 7.5 - Execution Management Model Overview

After planning, application execution happens in two phases, in a total of three steps. The first phase is the preparation of the plan for execution using the `preparePlan` operation of the **ExecutionManager**, resulting in an **ApplicationManager** factory object, which can be used to put the plan into action, potentially more than once. The second phase, launching the application, is divided into two steps. The first step of launching is calling the `startLaunch` operation on the **ApplicationManager**. This causes the **Application** to be executed, but not to be started yet. The second step of launching is calling the `finishLaunch` operation on the **Application**. The reason for splitting application launch into two steps is launch-time configuration and interconnection. The first step returns references to ports that are provided by the application, the second step supplies references to ports that are used by the application.

Application execution involves the “domain” level and the “node” level. On the domain level, the **ExecutionManager** manages the execution of an application into the domain. The **ExecutionManager** separates the “global” application into “local” sub-applications that execute within a node. This essentially creates “virtual components” to run entirely within a node, including intra-node connections. The deployment of virtual components onto a node can be described the same way as the deployment of the original application, using a **DeploymentPlan**, with the limitation that all component instances will be located on the same node.

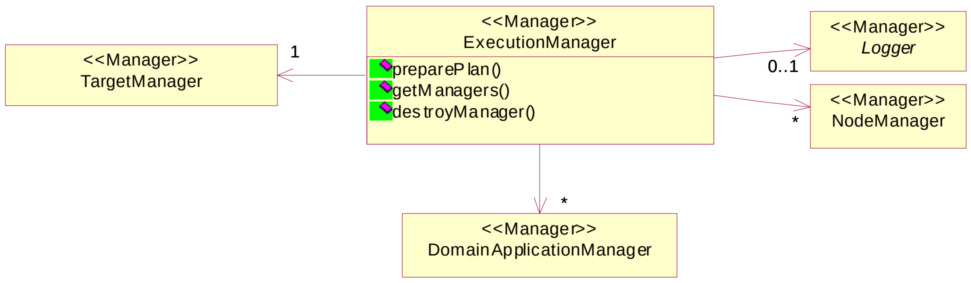
The **ExecutionManager** creates deployment plans for virtual components to run on each node, so that the complete application is covered. It then passes each **DeploymentPlan** to the **NodeManager** that is responsible for instantiating components on that node.

Just as the **DeploymentPlan** structure is the same for the deployment of both the global application and the local applications, the interfaces for managing them, **ApplicationManager** and **Application**, are the same. To keep the semantics separate, global and local versions of both interfaces are introduced with the Domain and Node prefixes. During launch and shutdown, global **DomainApplicationManager** and **DomainApplication** instances delegate management to the local, node-specific **NodeApplicationManager** and **NodeApplication** managers with the same interface.

The separation between **ExecutionManager** and **NodeManager** serves the purpose of creating a vendor boundary. It uncouples deployment (implemented by the vendor of the deployment engine) from the execution of components (implemented by the vendor of the hardware or development environment). This allows hardware vendors to supply a node-specific **NodeManager**, **NodeApplicationManager**, and **NodeApplication** implementations that can then interact with any deployment engine.

7.8.2 ExecutionManager

7.8.2.1 Description



The **ExecutionManager** manages the execution of applications from a **DeploymentPlan**. It has knowledge of **NodeManager** instances that manage nodes within the domain, and will delegate execution of component instances to relevant **NodeManager** instances as described by the plan. The **ExecutionManager** is also associated with a **TargetManager** for resource management, and, optionally, a centralized logging facility.

Application execution is initiated by preparing a **DeploymentPlan** using the **preparePlan** operation. This creates a new **DomainApplicationManager** that can later be used to launch one or more application instances.

7.8.2.2 Operations

- `preparePlan` (`plan`: **DeploymentPlan**, `resourceCommitment`: **ResourceCommitmentManager** [0..1]): **DomainApplicationManager**
  Creates an application manager (factory) from a deployment plan. If the `resourceCommitment` parameter is present and non-nil, then it is assumed that resources were already committed by an online planner, and the **ResourceCommitmentManager** object is adopted. Otherwise, the resources used by the plan are committed via the **TargetManager**, creating a new **ResourceCommitmentManager** object in the process. Raises the **ResourceNotAvailable** exception if `commitResources` is false, and one of the requested resources is not available. Raises the **StartError** exception if a deployment-related error occurs during preparation. Raises the **PlanError** exception if there is a problem with the plan.

- `destroyManager` (`manager`: **DomainApplicationManager**)
  Terminates an application manager and frees all associated resources by destroying the **ResourceCommitmentManager** object. All running applications are terminated as well. Raises the **StopError** exception if a problem occurs terminating or unpreparing any application. Raises the **InvalidReference** exception if the manager is unknown.

- `getManagers` (): **DomainApplicationManager** [*]
  Returns a list of all active application managers.

7.8.2.3 Associations

- `domainApplicationManager`: **DomainApplicationManager** [*]
  An **ExecutionManager** instantiates **DomainApplicationManager** instances.

- `targetManager`: **TargetManager** [1]
  The **TargetManager** that will be used for resource commitments.

- `logger`: **Logger** [0..1]
  An optional logging faciltiy.

- `nodeManager`: **NodeManager** [*]
  NodeManager references for all nodes that are part of the domain.

7.8.2.4 Constraints

No constraints.

7.8.2.5 Semantics

The semantics of preparation are undefined. Preparation usually involves the distribution of artifacts to the nodes. However, implementations might decide to delay this distribution until application launch — or they might, on the other hand, preload artifacts into memory so that launch can happen as fast as possible.

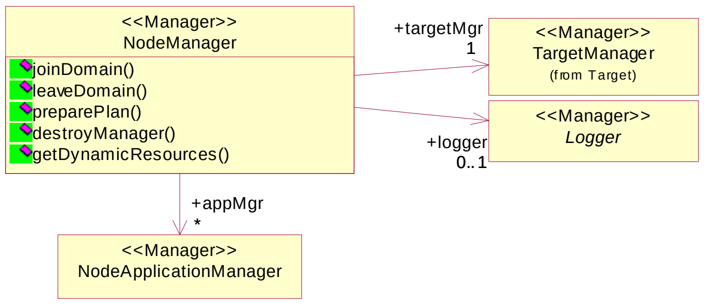
It is also undefined whether resource commitment (in case the `commitResources` parameter to the `preparePlan` operation is true) happens at preparation or launch time. Implementations should document their behavior in this respect.

The `preparePlan` operation takes the deployment plan and prepares “virtual components” with the subset of the application that is to be executed on each node. The **ExecutionManager** then contacts the **NodeManager** instances that are responsible for each node, and passes their piece of the application to their **preparePlan** operation, using the same **DeploymentPlan** format. This results in a “global” level **DomainApplicationManager** that holds references to “local,” node-specific **NodeApplicationManager** instances for each piece of the application.

The `destroyManager` operation releases all resources that were allocated during preparation and launch.

7.8.3 NodeManager

7.8.3.1 Description



The **NodeManager** is responsible for managing partial applications that are limited to its node. It mirrors the **ExecutionManager**, but is limited to one node only.

7.8.3.2 Operations

- `joinDomain` (`domainSubset`: **Domain**, `manager`: **TargetManager**, `log`: **Logger**, `updateInterval`: **Integer**)
  Informs the **NodeManager** that it is now part of a **Domain**. The **domainSubset** contains the resource availability information that is currently known within the domain. **manager** is a reference to the **TargetManager** to (optionally) send domain updates to. `log` is an abstract (PSM defined) class to send log messages to.

- `leaveDomain` () Informs the **NodeManager** that it is being removed from the domain, e.g., because of domain shutdown.

- `preparePlan` (`plan`: **DeploymentPlan**, `resourceCommitment`: **ResourceCommitmentManager**): **NodeApplicationManager**
  Prepares a partial application. The part of the application that is to be executed on this node is expressed as a **DeploymentPlan** that implements a "virtual component" with the subcomponents, connections, external ports, and properties. Raises the **StartError** exception if a deployment-related error occurs during preparation. Raises the **PlanError** exception if there is a problem with the plan.

- `destroyManager` (`manager`: **NodeApplicationManager**)
  Terminates a **NodeApplicationManager** and frees all associated resources. All running applications are terminated. Raises the **StopError** exception if an error occurs during termination. Raises the **InvalidReference** exception if the manager reference is unknown.

- `getDynamicResources` (): **Resource** [*]
  Retrieves the values of dynamic **SatisfierProperty** elements associated with the node.

7.8.3.3 Associations

- `targetManager`: **TargetManager** [1]
  The **TargetManager** that **Domain** updates are sent to if necessary. This is the reference passed as a parameter to the `joinDomain` operation.

- `logger`: **Logger** [0..1]
  The **Logger** to send log messages to. If the **NodeManager** wants to produce log messages, it keeps the reference passed as a parameter to the `joinDomain` operation.

- `nodeApplicationManager`: **NodeApplicationManager** [*]
  The node-specific application managers instantiated by this **NodeManager** via the `preparePlan` operation.

7.8.3.4 Constraints

No constraints.

7.8.3.5 Semantics

The `joinDomain` operation is called by the **ExecutionManager** at startup time or when it is informed of a new node via the **updateDomain** operation. Both the `joinDomain` and `leaveDomain` operations are called by the **ExecutionManager** on user request to add or remove nodes from a domain.

If the `joinDomain` operation is called, the **NodeManager** may optionally examine the `domainSubset`, and send an update message to the **TargetManager** if discrepancies are found.

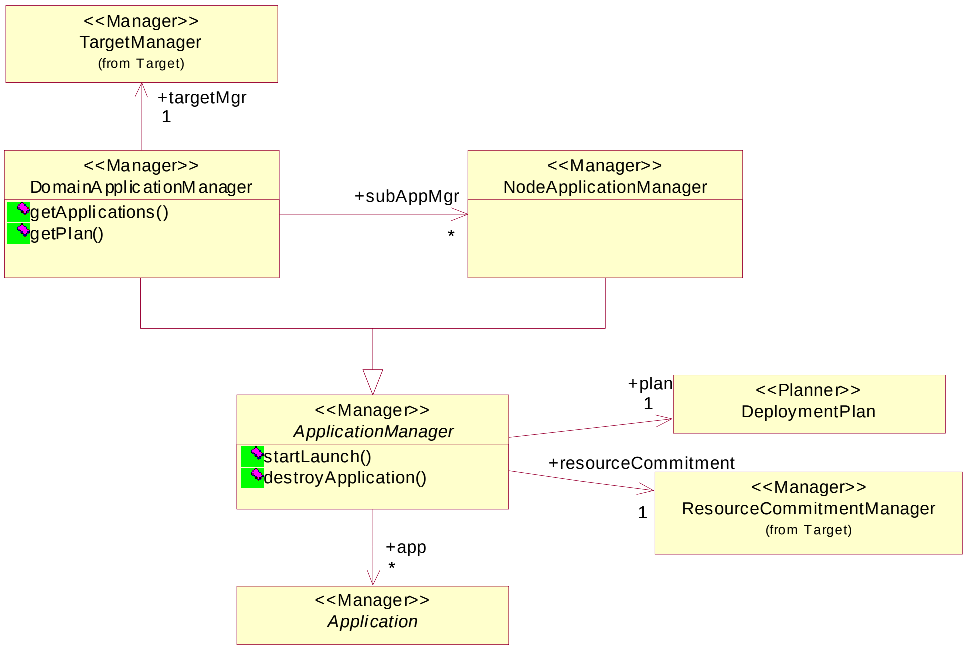
The `updateInterval` argument to the `joinDomain` operation specifies the maximum time interval between updates of changing dynamic **SatisfierProperty** values by the **NodeManager** using updateDomain operations to the **TargetManager**, in microseconds. The **NodeManager** should make best effort to not exceed this time interval, while getting as close to it as possible. A value of zero indicates that no such updates are requested.

The semantics of the `leaveDomain` operation are undefined. A **NodeManager** might shutdown or reset. In particular, the effect on running applications is also undefined. Behavior of a **NodeManager** implementation should be well documented. A **NodeManager** should not log any messages after returning from the `leaveDomain` operation.

The `preparePlan` operation and `destroyApplication` operations are called by the **ExecutionManager** as a result of a user demand for application preparation or destruction. The **DeploymentPlan** that is passed to the `preparePlan` operation describes a virtual component that is composed of all subcomponents and connections that are to be made within the node, plus mappings for connections and properties that are external to that node.

7.8.4 ApplicationManager

7.8.4.1 Description



An **ApplicationManager** is used to first launch and later to terminate an application according to a concrete **DeploymentPlan**. **ApplicationManager** is an abstract class that is specialized by the **DomainApplicationManager**, which handles deployment of a "global" application, and the **NodeApplicationManager**, which handles deployment of a locality constrained application onto a single node.

7.8.4.2 Operations

- `startLaunch` (`configProperty`: **Property** [*], `out providedReference`: **Connection** [*]): **Application**
  Executes the application, but does not start it yet. Users can optionally provide launch-time configuration properties to override properties that are part of the plan. A handle to the application is returned, as well as connections for the component's external provider ports. Raises the **InvalidProperty** exception if a configuration property is invalid. Raises the **StartError** exception if an error occurs during launching. If the **commitResources** parameter to the prepare operation of the **ExecutionManager** was true when this **ApplicationManager** was created, then this operation must, directly or indirectly, perform the resource commitment with the **TargetManager** before returning, and raise the **ResourceNotAvailable** exception if any of the requested resources are not available. Raises the **InvalidNodeExecParameter** exception if a parameter in the `nodeExecParameter` list is not valid for the execution environment. Raises the **InvalidComponentExecParameter** if a parameter in the `componentExecParameter` list is not valid for the primary artifact (implementation code).

- `destroyApplication` (`app`: **Application**)
  Terminates a running application. Raises the **StopError** exception if an error occurs during termination. Raises the **InvalidReference** exception if the appliction reference is unknown.

7.8.4.3 Associations

- `runningApp`: **Application** [*]
  The applications that were launched but not terminated yet.

- `deploymentPlan`: **DeploymentPlan** [1]
  The **DeploymentPlan** that this **ApplicationManager** is based on, a copy of the plan that was passed to the `preparePlan` operation of the **ExecutionManager** or **NodeManager**.

7.8.4.4 Constraints

Depending on the plan and whether it was based on static or online resource data, launching multiple applications from the same **ApplicationManager** in parallel might fail because of resource constraints.

7.8.4.5 Semantics

The behavior of an **ApplicationManager** is different depending on whether it is used as a **DomainApplicationManager** on the “global” level (if instantiated from an **ExecutionManager**) or a **NodeApplicationManager** on the “local” level (if instantiated from a **NodeManager**). Implementations for these two cases are usually separate. An **ExecutionManager** implementation has access to **DomainApplicationManager** and **DomainApplication** implementations, a **NodeManager** has access to **NodeApplicationManager** and **NodeApplication** implementations.

7.8.5 DomainApplicationManager

7.8.5.1 Description

The **DomainApplicationManager** is responsible for deploying an application on the domain level, i.e., across nodes. It specializes the **ApplicationManager** interface.

7.8.5.2 Operations

- `getApplications` (): **Application** [*]
  Returns a list of all applications that have been launched from this **ApplicationManager** and that are still executing.

- `getPlan` (): **DeploymentPlan**
  Returns the **DeploymentPlan** associated with this **ApplicationManager**.

7.8.5.3 Associations

- `subAppMgr`: **NodeApplicationManager** [*]
  The manager for the pieces of the application that run on each node.

- `targetManager`: **TargetManager** [1]
  The **TargetManager** that is used to commit resources if necessary.

7.8.5.4 Constraints

The targets of the `runingApp` association (inherited from **ApplicationManager**) are instances of **DomainApplication**.

7.8.5.5 Semantics

A **DomainApplicationManager** has references to node-specific **NodeApplicationManager** elements as created by the `preparePlan` operation of the **ExecutionManager**. The `startLaunch` operation then calls `startLaunch` on the **NodeApplicationManager** instances, passing the relevant properties and collecting the returned connections as determined by the separation of the "global" **DeploymentPlan** into node-specific plans. The same applies to the `destroyApplication` operation.

7.8.6 NodeApplicationManager

7.8.6.1 Description

The **NodeApplicationManager** is responsible for deploying a locality constrained application onto a node. It specializes the **ApplicationManager** interface.

7.8.6.2 Operations

No additional operations.

7.8.6.3 Associations

No additional associations.

7.8.6.4 Constraints

The targets of the **runingApp** association (inherited from **ApplicationManager**) are instances of **NodeApplication**.

The associated **DeploymentPlan** (inherited from **ApplicationManager**) only contains instance deployments onto the node that is represented by the **NodeManager** parent.

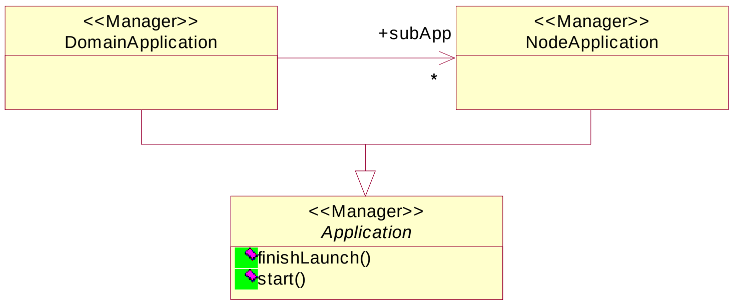
7.8.6.5 Semantics

A **NodeApplicationManager** is responsible for executing and terminating component instances on the node that it is part of (as defined by the **NodeManager** parent, usually but not necessarily implying co-location).

A **NodeApplicationManager** can use the **ResourceCommitmentManager** object to release resources that were allocated for the execution of this application, when it is certain that these resources will not be used throughout the remainder of the application's lifetime, e.g., when a **NodeApplication** terminates before `destroyApplication` is called.

7.8.7 Application

7.8.7.1 Description



Application is an abstract class represents a running application. The **Application** class may be mapped to different classes in a platform specific models, potentially allowing navigation to an application's ports, configuration or introspection at runtime. **Application** is specialized by **DomainApplication**, which represents a "global" application (i.e., across nodes), and **NodeApplication**, which represents a locality constrained application that is running on a single node.

7.8.7.2 Operations

- `finishLaunch` (`providedReference`: **Connection** [*], `start`: **Boolean**)
  The second step in launching an application. External references may be provided to connect to the component's external user ports. If the start parameter is true, the application is started as well. Raises the **InvalidConnection** if one of the provided references is invalid. Raises the **StartError** exception if launching or starting the application fails.

- `start` ()
  Starts the application. Raises the **StartError** exception if starting the application fails.

7.8.7.3 Associations

No associations.

7.8.7.4 Constraints

No constraints.

7.8.7.5 Semantics

The `finishLaunch` operation must be called in order to complete the component's configuration.

If clients want to start multiple applications simultaneously, they can set the start parameter of the `finishLaunch` operation to false and then call the `start` operation separately. If clients want to avoid the additional round-trip, they can set the start parameter of the **finishLaunch** operation to true; in that case, the start operation need not be called.

The behavior of an **Application** is different depending on whether it is used on a "global" level (if its parent is a **DomainApplicationManager**) or on a "local" level (if its parent is a **NodeApplicationManager**). Implementations for these two cases are usually separate. A **DomainApplicationManager** only creates **DomainApplication** instances, a **NodeApplicationManager** only creates **NodeApplication** instances.

A node-specific **Application** represents running component instances on the node that it is part of (as defined by the **NodeManager** parent, usually but not necessarily implying co-location).

7.8.8 DomainApplication

7.8.8.1 Description

A **DomainApplication** represents a “global” application that was deployed across nodes. It has the same interface as **Application**, but has different semantics.

7.8.8.2 Operations

No additional operations.

7.8.8.3 Associations

- `subApp`: **NodeApplication** [*]
  The pieces of the application that run on each node.

7.8.8.4 Constraints

No constraints.

7.8.8.5 Semantics

A “global” **DomainApplication** has references to node-specific **NodeApplication** elements as created by the `startLaunch` operation of the **DomainApplicationManager**. The `finishLaunch` operation then calls `finishLaunch` on the node-specific **NodeApplication** instances, passing the relevant connections as determined by the separation of the “global” **DeploymentPlan** into node-specific plans. The same applies to the `destroyApplication` operation.

7.8.9 NodeApplication

7.8.9.1 Description

NodeApplication represents a piece of an application that is executing within a single domain.

7.8.9.2 Operations

No additional operations.

7.8.9.3 Associations

No additional associations.

7.8.9.4 Constraints

No constraints.

7.8.9.5 Semantics

NodeApplication has the same semantics as the **Application** base class. It interconnects and starts the piece of the application that is being launched on the node that is represented by the **NodeManager** parent.

7.8.10 Logger

7.8.10.1 Operations

No operations.

7.8.10.2 Associations

No associations.

7.8.10.3 Constraints

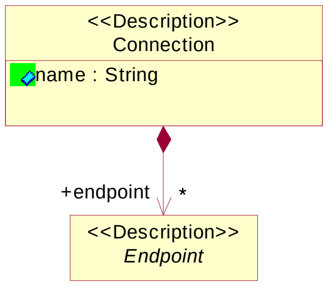
No constraints.

7.8.10.4 Semantics

Logger is an abstract runtime class to facilitate logging within the domain. It has to be mapped to a concrete type by platform specific models.

7.8.11 Connection

7.8.11.1 Description



A **Connection** is used to describe connections from or to a component port at runtime.

7.8.11.2 Attributes

- `name`: **String**
  The name of the component's port.

7.8.11.3 Associations

- `endpoint`: **Endpoint** [*]
  The endpoints that are part of the connection.

7.8.11.4 Constraints

No constraints.

7.8.11.5 Semantics

No additional semantics.

7.8.12 Endpoint

7.8.12.1 Attributes

No attributes.

7.8.12.2 Associations

No associations.

7.8.12.3 Constraints

No constraints.

7.8.12.4 Semantics

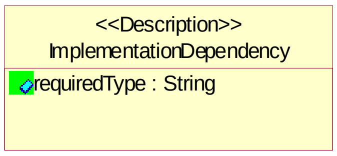
Endpoint is an abstract class that contains the “address” of an endpoint. This class needs to be mapped into a concrete platform specific type.

7.9 Common Elements

This section contains common model elements that are shared between multiple segments. They are placed in the Common subpackage of the Deployment and Configuration package.

7.9.1 ImplementationDependency

7.9.1.1 Description



Expresses a dependency that an implementation has on the target environment. Before this implementation can be deployed, an application of the required type must exist (it must have finished launching) in the target environment.

7.9.1.2 Attributes

- `requiredType`: **String**
  The interface type of which an application must exist.

7.9.1.3 Associations

No associations.

7.9.1.4 Constraints

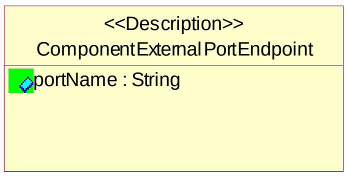
No constraints.

7.9.1.5 Semantics

When launching an application, the **ExecutionManager** and **DomainApplicationManager** verify that applications of the required type are already executing.

7.9.2 ComponentExternalPortEndpoint

7.9.2.1 Description



Identifies a port of the external component as an endpoint of the connection described by the **AssemblyConnectionDescription** that this element is contained in.

7.9.2.2 Attributes

- `portName`: **String**
  The name of the port of the external component.

7.9.2.3 Associations

No associations.

7.9.2.4 Constraints

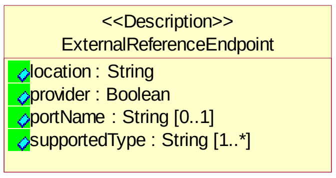
No constraints.

7.9.2.5 Semantics

See above.

7.9.3 ExternalReferenceEndpoint

7.9.3.1 Description



Identifies a location outside the assembly as an endpoint of the connection described by an **AssemblyConnectionDescription**.

7.9.3.2 Attributes

- `location`: **String**
  References a port outside of the assembly that is to be an endpoint of this connection, which is resolved at execution time.

- `provider`: **Boolean**
  Indicates whether the intention is to connect to a provider of service (reference is to either an object or a component) or user of service (reference is to a component).

- `portName`: **String** [0..1]
  This optional attribute indicates, if present, that the external entity referenced by the location attribute is a component, and further indicates which port of that component is to be connected.

- `supportedType`: **String** [1..*]
  The interface type of the external endpoint being connected. For user endpoints, the set of types required by the using entity: connect a user requiring any one of these types. For provider endpoints, the set of types supported by the provider.

7.9.3.3 Associations

No associations.

7.9.3.4 Constraints

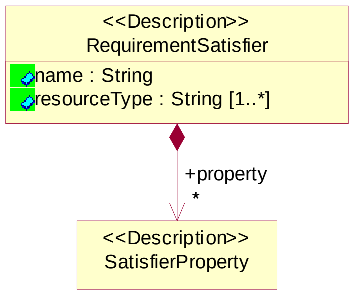
No constraints.

7.9.3.5 Semantics

The location is to be an endpoint to this connection in the assembly. Whether the endpoint is a provider or user port is implied by the URL, and its type is assumed to be compatible with the connection.

7.9.4 RequirementSatisfier

7.9.4.1 Description



RequirementSatisfier describes a resource or capability that can satisfy a requirement.

7.9.4.2 Attributes

- `name`: **String**
  The requirement satisfier’s name, which uniquely identifies this requirement satisfier within its container.

- `resourceType`: **String** [1..*]
  The resource types that can be satisfied by this satisfier.

7.9.4.3 Associations

- `property`: **SatisfierProperty** [*]
  Properties associated with this satisfier.

7.9.4.4 Constraints

There must be at least one element in the **resourceType** sequence attribute.

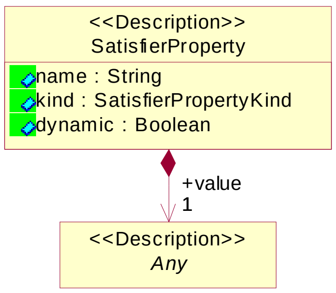
context RequirementSatisfier inv:
self.resourceType->size() >= 1

7.9.4.5 Semantics

The type of a **Requirement** is must match one of the elements in the `resourceType` attribute. The requirement's properties will then be matched against the satisfier's properties.

7.9.5 SatisfierProperty

7.9.5.1 Description



Describes a specific property of a **Resource** or **SharedResource**. It contains a **SatisfierPropertyKind** that classifies the **SatisfierProperty** and has implications on the type of the value and the comparison between the **SatisfierProperty** and a required **Property**.

7.9.5.2 Attributes

- `name`: **String**
  The name of the property.

- `kind`: **SatisfierPropertyKind**
  The kind of the property.

- `dynamic`: **Boolean**
  Whether the value of this property can change during runtime.

7.9.5.3 Associations

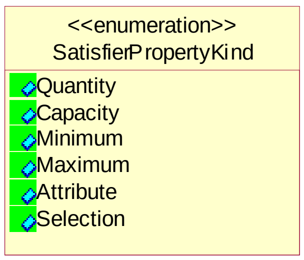
- `value`: **Any** [1]
  The value of the property.

7.9.5.4 Semantics

SatisfierProperty elements are matched against the **Property** elements within a **Requirement** at planning time. They describe attributes and capacities of hardware or software. The `name` attribute of the **SatisfierProperty** must match the `name` attribute of the **Property** it is compared against. Matching the values will be discussed as part of the **SatisfierPropertyKind** semantics. The type of the value may be fully or partially implied by the kind.

7.9.6 SatisfierPropertyKind

7.9.6.1 Description



Classifies a **SatisfierProperty**. Each **SatisfierPropertyKind** identifies a specific way to match requirements against resources. The kind of **SatisfierPropertyKind** implies the types of the values contained in **SatisfierProperty** and **Property**, and the algorithm to check their compatibility.

7.9.6.2 Attributes

No attributes.

7.9.6.3 Associations

No associations.

7.9.6.4 Semantics

The value of this enumeration implies how to check for compatibility between a required property and a resource's property, and how to keep track of capacities. In the following text, "property" refers to the property element of the **SatisfierProperty**, and "requirement" refers to the property element of the **Requirement**. Both must have matching names.

- `Quantity`
  This property exists in a certain quantity, but its capacity is not considered. The value of the property is of integer type. The value of the requirement is ignored, but each time this property is used, the quantity is decreased by one until zero. To match the requirement, the property must have a value of at least one. Example: a sound card with 4 output channels.

- `Capacity`
  This property has a certain capacity that can be consumed. The value of the property and the requirement property are both of numerical type. The value of the requirement is subtracted from the value of the property. To match the requirement, the property must have a value that equals or exceeds the value of the requirement. Example: memory size.

- `Minimum`
  The property describes a capability with a lower bound. The value of the property and the requirement are both of a type that supports ordering. To match, the value of the requirement must equal or exceed the value of the property. Example: latency – e.g., the resource can guarantee 30ms latency, the property requires at least 40ms.

- `Maximum`
  The property describes a capability with an upper bound. The value of the property and the requirement are both of a type that supports ordering. To match, the value of the requirement must be equal or lesser than the value of the property. Example: CPU speed – e.g., the property has 700MHz, and there is a requirement on at least 500MHz.

- `Attribute`
  The value of the property and the requirement are both of a type that supports equality comparison. To match, the requirement must compare equal to the property. Example: OS type.

- `Selection`
  The type of the property is a sequence of a type that supports equality comparison, the requirement is a single value of the same type. To match, the value of the requirement must compare equal to one element of the property values.

Platforms have to specify concrete types to be used for the comparison of the `Minimum`, `Maximum`, `Attribute`, and `Selection` kinds, and define how to order and compare them.

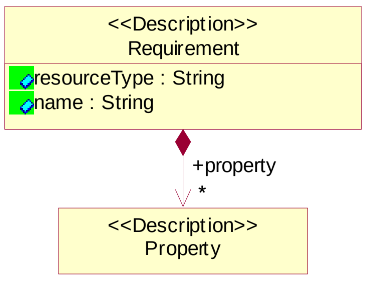
Domains have to define resource types, their properties, and the kinds to use for each property.

The `Quantity` and `Attribute` kinds are redundant, but included here to account for these common use cases. (`Quantity` is equivalent to a `Capacity` that is required in amounts of one, and `Attribute` is a subset of `Selection`.)

The above list of resource kinds is expected to cover the most common use cases. Platform specific models and domain specific profiles are allowed to add more kinds if necessary.

7.9.7 Requirement

7.9.7.1 Description



Requirement is used in the **MonolithicImplementationDescription**, **ImplementationArtifactDescription**, and the **AssemblyConnectionDescription** to express that the implementation artifact or connection has requirements that must be fulfilled by resources in the target environment. The resource type must match the type of a resource.

7.9.7.2 Attributes

- `name`: **String**
  The name of this requirement, used in the **DeploymentPlan** to link resources to the requirements they are intended to satisfy.

- `resourceType`: **String**
  Identifies the resource type.

7.9.7.3 Associations

- `property`: **Property** [*]
  Properties associated with the resource.

7.9.7.4 Constraints

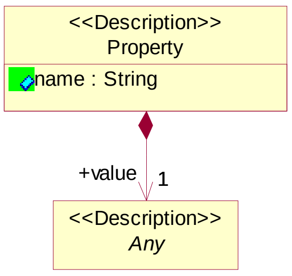
No constraints.

7.9.7.5 Semantics

No semantics.

7.9.8 Property

7.9.8.1 Description



A **Property** has a name and a value. It is used to carry names and values in various places.

7.9.8.2 Attributes

- `name`: **String**
  The name of the property.

7.9.8.3 Associations

- `value`: **Any** [1]
  Contains the value.

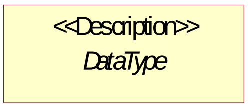
7.9.8.4 Constraints

No constraints.

7.9.8.5 Semantics

No semantics.

7.9.9 DataType



7.9.9.1 Attributes

No attributes.

7.9.9.2 Associations

No associations.

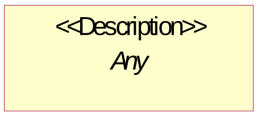
7.9.9.3 Constraints

No constraints.

7.9.9.4 Semantics

DataType is an abstract class that describes a data type. This class needs to be mapped into a concrete platform specific type.

7.9.10 Any



7.9.10.1 Attributes

No attributes.

7.9.10.2 Associations

No associations.

7.9.10.3 Constraints

No constraints.

7.9.10.4 Semantics

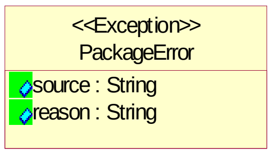
Any is an abstract class that contains a typed value. This class needs to be mapped into a concrete platform specific type.

7.10 Exceptions

All exceptions are placed in the Exception subpackage of the Deployment and Configuration package.

7.10.1 PackageError

7.10.1.1 Description



The **PackageError** exception is raised by the **installPackage** operation of the **RepositoryManager** if an internal error is detected in the package. (Potential reasons include the non-existence of a referenced file, or unresolved subcomponent references in an assembly.)

7.10.1.2 Attributes

- `source`: **String**
  Identifies a location in the package where the error occured.

- `reason`: **String**
  A human-readable description of the problem.

7.10.1.3 Associations

No associations.

7.10.1.4 Constraints

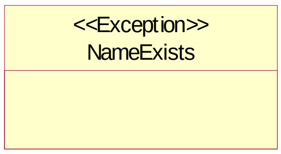
No constraints.

7.10.1.5 Semantics

The **RepositoryManager** implementation should compose a human readable value for the `source` attribute from the `name` attributes of elements in the hierarchy defined by the **PackagedComponentImplementation**, **SubcomponentInstantiationDescription**, **AssemblyConnectionDescription**, **AssemblyPropertyMapping**, and **NamedImplementationArtifact** elements so that a user can locate the problem as precisely as possible.

7.10.2 NameExists

7.10.2.1 Description



The **NameExists** exception is raised by the `installPackage` and `createConfiguration` operations of the **RepositoryManager** if a **PackageConfiguration** with the to-be-created name already exists in the repository.

7.10.2.2 Attributes

No attributes.

7.10.2.3 Associations

No associations.

7.10.2.4 Constraints

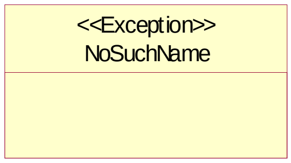
No constraints.

7.10.2.5 Semantics

No semantics.

7.10.3 NoSuchName

7.10.3.1 Description



The **NoSuchName** exception is raised by the `findConfigurationByLabel`, `createConfiguration`, `updateConfiguration`, and `deleteConfiguration` operations of the **RepositoryManager** if there is no **PackageConfiguration** with the requested name in the repository.

7.10.3.2 Attributes

No attributes.

7.10.3.3 Associations

No associations.

7.10.3.4 Constraints

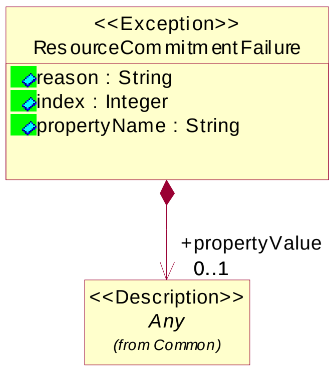
No constraints.

7.10.3.5 Semantics

No semantics.

7.10.4 ResourceCommitmentFailure

7.10.4.1 Description



The **ResourceCommitmentFailure** exception is raised by the **TargetManager** or **ResourceCommitmentManager**, if a resource allocation or deallocation cannot be satisfied.

7.10.4.2 Attributes

- `reason`: **String**
  A human-readable message explaining why the allocation failed.

- `index`: **Integer**
  Identifies the **ResourceAllocation** element that could not be satisfied, in a list of resource allocations. This identifies the top-level element in the **Domain** and the **Resource** that a property was to be committed or released of.

- `propertyName`: **String**
  Identifies the specific property that could not be satisfied. Matches the name of a **Property** associated with the **ResourceAllocation** element, and the name of a **SatisfierProperty** element associated with the **Resource**, if it exists.

7.10.4.3 Associations

- `propertyValue`: **Any** [0..1]
  The available value remaining for the property. If missing, then the **Resource** does not exist, or does not have a property by that name.

7.10.4.4 Constraints

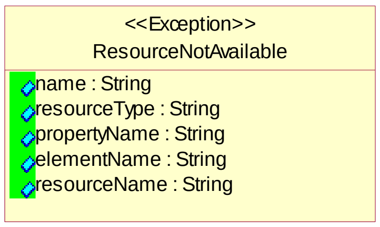
No constraints.

7.10.4.5 Semantics

No semantics.

7.10.5 ResourceNotAvailable

7.10.5.1 Description



The **ResourceNotAvailable** exception is raised by the `preparePlan` operation of the **ExecutionManager** or by the `startLaunch` operation of the **ApplicationManager** if a resource required by the plan is not available.

7.10.5.2 Attributes

- `name`: **String**
  Identifies the element in the plan whose resource requirement could not be satisfied.
- `resourceType`: **String**
  The type of resource that was requested using a **Requirement** element.
- `propertyName`: **String**
  The name of the property that could not be satisfied.
- `elementName`: **String**
  Identifies a **Node**, **Interconnect**, or **Bridge** within the **Domain**.
- `resourceName`: **String**
  The name of a **Resource** or **SharedResource** within the **Node**, **Interconnect**, or **Bridge** that was considered for matching the requirement.

7.10.5.3 Associations

No associations.

7.10.5.4 Constraints

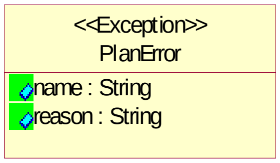
No constraints.

7.10.5.5 Semantics

The `name`, `resourceType`, and `propertyName` uniquely identify a requirement in the plan. The `elementName`, `resourceName`, and `propertyName` uniquely identify a requirement satisfier in the domain that failed to match the requirement. Note that `resourceName` can be the empty string if no **RequirementSatisfier** was found to match the `resourceType`.

7.10.6 PlanError

7.10.6.1 Description



The **PlanError** exception is raised by the `preparePlan` operation of the **ExecutionManager** if an inconsistency is detected in the plan (for example, an unresolved reference to a non-existent component instance).

7.10.6.2 Attributes

- `name`: **String**
  Identifies an element of the **DeploymentPlan** where the error occured.

- `reason`: **String**
  A human-readable reason that describes the error.

7.10.6.3 Associations

No associations.

7.10.6.4 Constraints

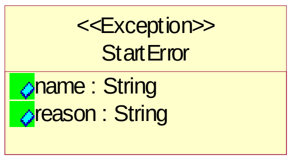
No constraints.

7.10.6.5 Semantics

This exception indicates that the plan is erroneous or inconsistent (i.e., the error is unrelated to the actual deployment).

7.10.7 StartError

7.10.7.1 Description



The **StartError** exception is raised if a problem occurred during deployment, either during preparation by the `preparePlan` operation of the **ExecutionManager** or during launch by the `startLaunch` operation of the **ApplicationManager**.

7.10.7.2 Attributes

- `name`: **String**
  Identifies an element of the **DeploymentPlan** where the error occured.

- `reason`: **String**
  A human-readable reason that describes the error.

7.10.7.3 Associations

No associations.

7.10.7.4 Constraints

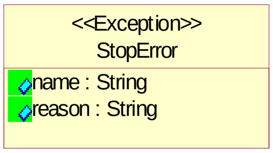
No constraints.

7.10.7.5 Semantics

Potential reasons include the inability to upload an artifact to a node or a failure during component instantiation.

7.10.8 StopError

7.10.8.1 Description



The **StopError** exception is raised if a problem occurred while terminating an application, either during the **terminate** operation of the **ApplicationManager** or during the **destroyManager** operation of the **ExecutionManager**.

7.10.8.2 Attributes

- `name`: **String**
  Identifies an element of the **DeploymentPlan** where the error occured.

- `reason`: **String**
  A human-readable reason that describes the error.

7.10.8.3 Associations

No associations.

7.10.8.4 Constraints

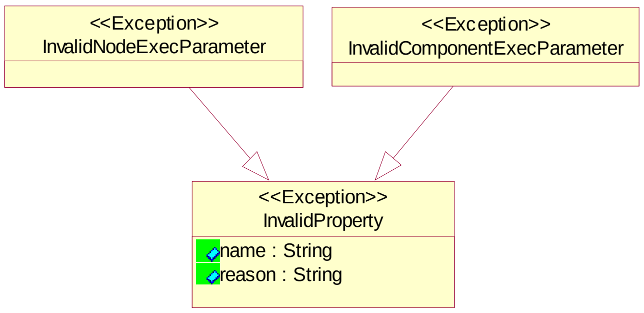
No constraints.

7.10.8.5 Semantics

This exception is raised if the problem is related to the “undeployment.” Potential reasons include the failure to stop a component instance.

7.10.9 InvalidProperty

7.10.9.1 Description



7.10.9.2 Attributes

- `name`: **String**
  The name of the property among the `configProperty` elements that caused the problem.

- `reason`: **String**
  A human-readable reason that describes the error.

7.10.9.3 Associations

No associations.

7.10.9.4 Constraints

No constraints.

7.10.9.5 Semantics

The **InvalidProperty** exception is raised if an invalid property is passed to the `startLaunch` operation of the **ApplicationManager**. The problem can be that either the name does not match any of the component's properties, or a type mismatch.

7.10.10 InvalidNodeExecParameter

7.10.10.1 Description

See below.

7.10.10.2 Attributes

No additional attributes.

7.10.10.3 Associations

No additional associations.

7.10.10.4 Constraints

No constraints.

7.10.10.5 Semantics

InvalidNodeExecParameter is raised when instantiating a component when one of the supplied parameters is invalid for the execution environment.

7.10.11 InvalidComponentExecParameter

7.10.11.1 Description

See below.

7.10.11.2 Attributes

No additional attributes.

7.10.11.3 Associations

No additional associations.

7.10.11.4 Constraints

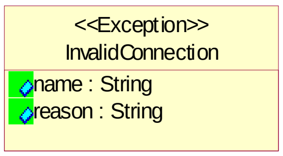
No constraints.

7.10.11.5 Semantics

InvalidComponentExecParameter is raised when instantiating a component when one of the supplied parameters is invalid for the implementation code.

7.10.12 InvalidConnection

7.10.12.1 Description



7.10.12.2 Attributes

- `name`: **String**
  The name of the property among the `configProperty` elements that caused the problem.
- `reason`: **String**
  A human-readable reason that describes the error.

7.10.12.3 Associations

No associations.

7.10.12.4 Constraints

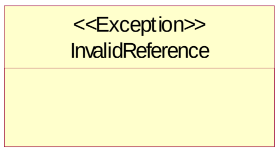
No constraints.

7.10.12.5 Semantics

The **InvalidConnection** exception is raised if an invalid connection is passed to the `finishLaunch` operation of the **Application**. The problem can be that the name does not match any of the component's ports, a type mismatch, or a direction mismatch (i.e., an attempt to connect a provider port to another provider port).

7.10.13 InvalidReference

7.10.13.1 Description



7.10.13.2 Attributes

No attributes.

7.10.13.3 Associations

No associations.

7.10.13.4 Constraints

No constraints.

7.10.13.5 Semantics

The **InvalidReference** exception is raised by the `destroyManager` operations of the **ExecutionManager** and **NodeManager** and the `destroyApplication` operation of the **ApplicationManager** if the **ApplicationManager** or **Application** reference is not known in this context. This may be because the reference was created by a different context, or because of prior destruction.

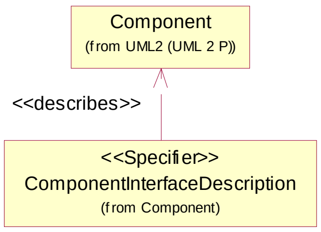
7.11 Relations to Other Standards

This section relates some classes in this platform independent model to classes from other packages. This section is explanatory and non-normative.

Both for **Artifact** and **Component**, the relation to the UML 2 Partners submission to the UML 2 RFP is weak; in both cases, it is through a dependency relationship (**ImplementationArtifact** is only referenced by a dependency with the «**describes**» stereotype from **ImplementationArtifactDescription**). **Artifact** and **Component** will therefore not show up in any code that is generated from the model.

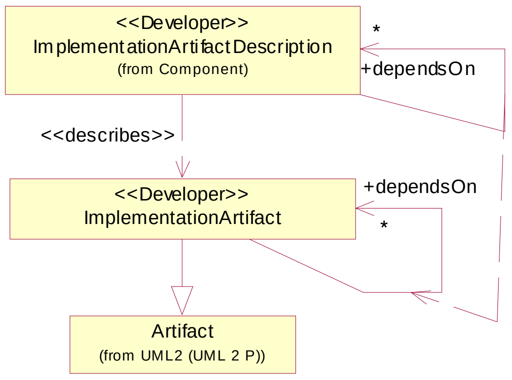
Since UML 2 is not an adopted standard yet, and since neither **Artifact** nor **Component** exist in UML 1.4, the dependencies might need to be updated or removed in sync with future iterations of UML 2 submissions. Because of the weak dependencies, changes in UML 2 do not have any impact on the models this document.

7.11.1 Component



ComponentInterfaceDescription describes the features of a **Component** that are relevant to the deployment process, such as property names and types and port names and types.

7.11.2 ImplementationArtifact



An **ImplementationArtifact** is a (potentially complete) piece of a concrete component implementation. An **ImplementationArtifact** is opaque to the deployment process and can only be evaluated in the context of a target environment (e.g., for execution). The **ImplementationArtifactDescription** captures the properties of an **ImplementationArtifact** that are relevant to the deployment process.

The dependency relationship between **ImplementationArtifactDescription** elements reflects the dependency between implementation artifacts (e.g., executables depending on shared libraries) in the data model.

ImplementationArtifact is a specialization of the **Artifact** class in the UML2.0 specification. It adds a self-relationship to describe dependencies between **Artifact** instances.

8 UML Profile for D+C Tool Support

This chapter defines the UML Profile for D&C Tool Support. The main objectives of the UML Profile for D&C Tool Support are:

- to define the notation necessary to support the component-based application development process and target environment description, as described in "Scope" on page 1.
- to enable the automatic generation of D&C descriptors from component assembly and target models.

The UML Profile for D&C Tool Support provides tool vendors with the foundation they need to develop UML tools that support the deployment and configuration of component-based distributed applications. The current D&C specification is composed of three main parts: Component, Target, and Deployment. This profile addresses the first two. The description of the deployment infrastructure is outside the scope of the current UML Profile for D&C Tool Support, and will need to be addressed separately. Currently UML allows deployment planners to define an explicit deployment plan by statically associating component with nodes.

The concepts and notation defined by this profile allows application developers to use UML to completely model the configuration of components, the assembly of components from other components, and the target environments into which components can be deployed.

The development of tools to support the D&C specification, based on the UML Profile for D&C Tool Support, offers many important advantages:

- Enables the integration of model validation techniques that will allow eliminating errors at the application design stage.
- Eliminates errors in the production of descriptors.
- Makes the component and target models independent of the specific format of the descriptors, which allow changing the format without having to change the models.
- Enables the integation with existing UML-based MDA tools.

8.1 Structure of the Profile

The UML Profile for D&C Tool Support is defined using the profiles mechanism defined in UML 2.0.

The UML Profile for D&C Tool Support is composed of a set of stereotypes that are defined as extensions of UML 2.0 metaclasses. In particular, the D&C Profile for Tool Support extends metaclasses defined in the UML 2.0 Component, Composite Structures, and Deployment packages. The dependencies between the D&C Profile for Tool Support and UML 2.0 packages is illustrated in Figure 8.1.

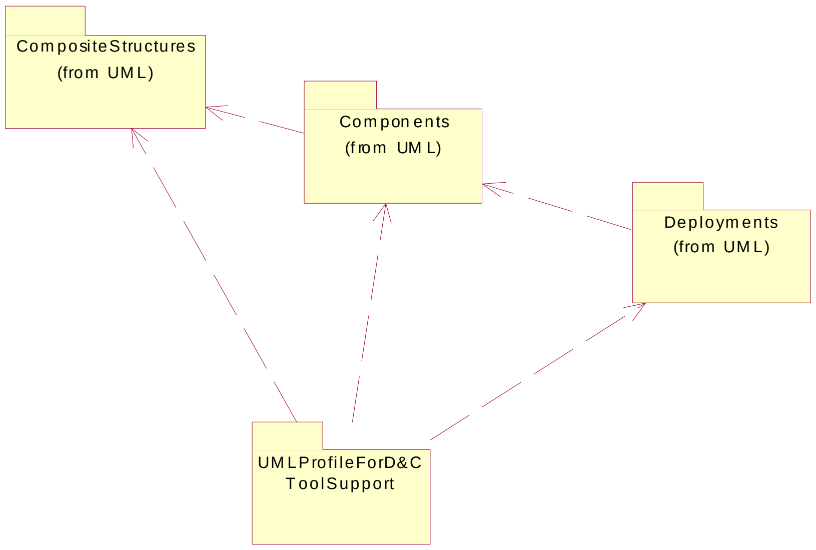


Figure 8.1 - Dependencies between the UML Profile for D&C Tool Support and UML 2.0 packages

The set of stereotypes that compose the UML Profile for D&C Tool Support are grouped in two distinct packages: Component and Target. The Component package defines the set of stereotypes that are used to model a component-based distributed application. The Target package defines the set of stereotypes that are used to model a distributed deployment target.

The content of these packages is defined in the next two sections (Section 8.2, “Package Components,” on page 99 and Section 8.3, “Package Targets,” on page 106). The dependencies between the Component and Target packages and the UML 2.0 packages are illustrated in Figure 8.2.

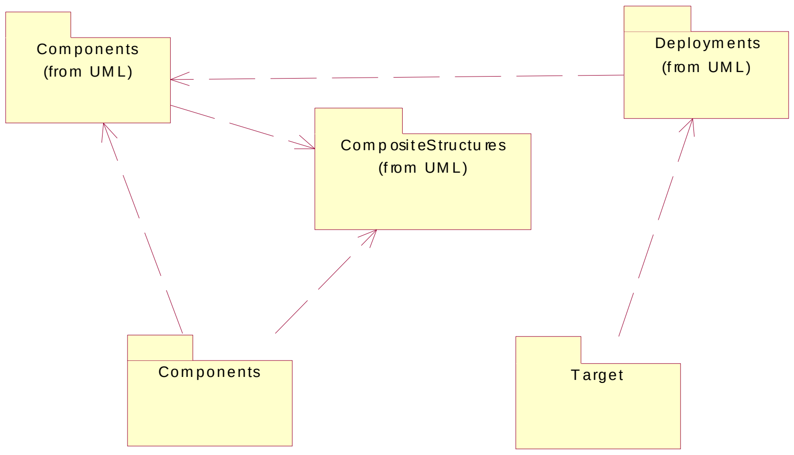


Figure 8.2 - Dependencies between the Component and Target packages and UML 2.0 packages

8.2 Package Components

The Component package defines the set of stereotypes that are used to model a component-based distributed application. The list of stereotypes currently defined in the Component package includes: Component, ComponentAssembly, Connection, Port, and Artifact.

This section defines the set of stereotypes contained in the package Components.

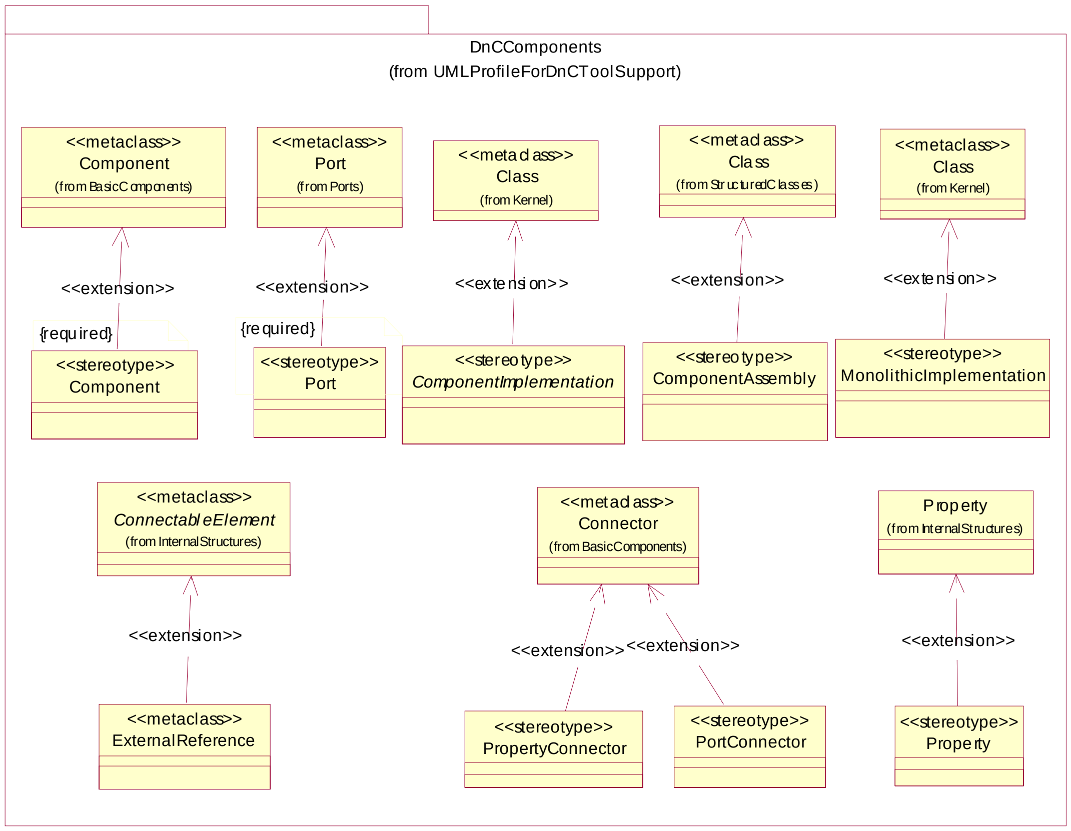


Figure 8.3 - Components package

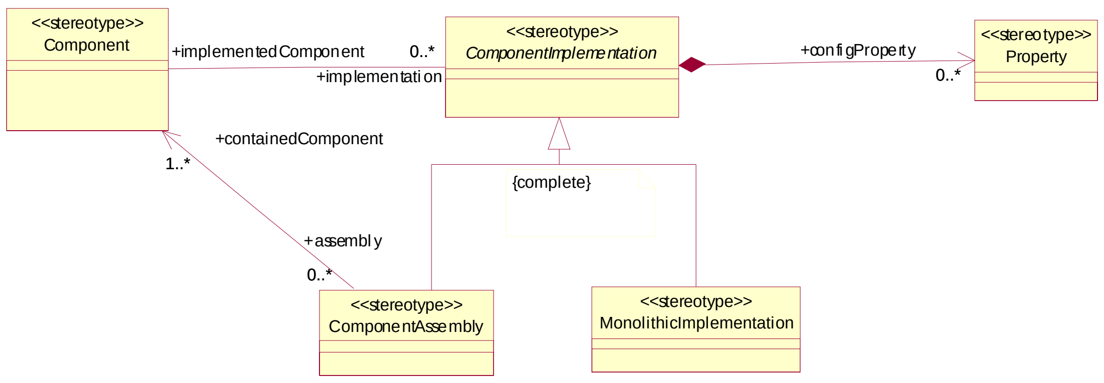
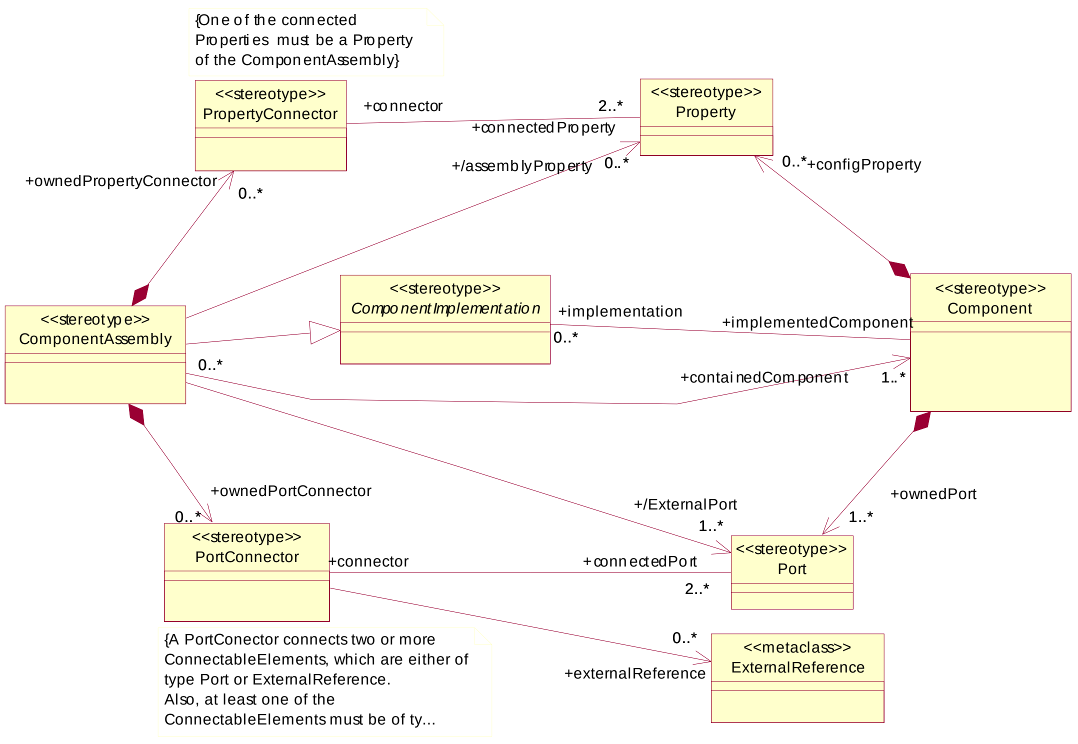


Figure 8.4 - Component implementation relationships



{One of the connected Properties must be a Property of the ComponentAssembly}
<<stereotype>> PropertyConnector
+connector
2..*
<<stereotype>> Property
+connectedProperty
+/assemblyProperty
0..*
0..*+configProperty
+ownedPropertyConnector
0..*
<<stereotype>> ComponentImplementation
+implementation
0..*
+implementedComponent
<<stereotype>> Component
<<stereotype>> ComponentAssembly
0..*
+containedComponent
1..*
+ownedPortConnector
0..*
+/ExternalPort
1..*
+ownedPort
1..*
<<stereotype>> PortConnector
+connector
+connectedPort
<<stereotype>> Port
2..*
0..*
<<metaclass>> ExternalReference
+externalReference
{A PortConector connects two or more ConnectableElements, which are either of type Port or ExternalReference. Also, at least one of the ConnectableElements must be of ty...

Figure 8.5 - ComponentAssembly Stereotype

8.2.1 Capability

8.2.1.1 Description

Capability is used to describe an implementation’s capabilities, which are matched against selection requirements.

8.2.1.2 Attributes

- **name**: **String**
  An optional name for the requirement satisfier.

- **resourceType**: **Sequence(String)**
  The resource types that can be satisfied by this satisfier.

8.2.1.3 Associations

No associations

8.2.2 Component (Stereotype)

8.2.2.1 Description

The Component metaclass extends the UML Component metaclass (from UML2.0::Components). In UML 2.0, a component is defined in terms of its set of ports and has references to its realizations.

The Component stereotype is defined as "required," which means that every instance of the Component metaclass must be associated with an instance of the Component stereotype.

8.2.2.2 Attributes

- **label**: **String**
  An optional human-readable label for the component.

- **UUID**: **String**
  An optional unique identifier for this component.

8.2.2.3 Associations

- **implementation:** ComponentImplementation [0..*]
  References the Classifiers of which the Component is an abstraction, i.e., that realize its behavior. This association renames the "realization" association owned by Component (from UML2.0::Components::Component).

- **configProperty**: **Property** [*]
  Contains the set of configurable properties of the component. These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen. configProperty is a subset of the ownedAttribute association of Component (inherited from UML2.0::CompositeStructures::InternalStructures::StructuredClassifier).

- **ownedPort: Port** [*]
  Contains the set of ports of the component.These configuration properties are used to configure the component once instantiated. This allows the definition of configuration properties in a package regardless of which implementation is chosen. ownedPort is a renaming of the ownedPort association of Component (inherited from UML2.0::CompositeStructures::Ports::EncapsulatedClassifier).

Note: Definition. Component (from UML2.0::Components): A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces. A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parameterize the component's execution.

8.2.3 ComponentAssembly (Stereotype)

8.2.3.1 Description

In spite of the fact that UML 2.0 allows for the recursive definition of components in terms of subcomponents (based on the fact that a UML 2.0 Component is a specialization of UML2.0::StructuredClass::Class), the concept of component assembly is not explicitly defined in UML 2.0. The ComponentAssembly stereotype specializes the UML 2.0 Class metaclass from StructuredClasses (UML2.0::CompositeStructures::StructuredClasses). It is a subclass of the ComponentImplementation stereotype.

A ComponentAssembly is a classifier whose behavior is fully described by the collaboration of a set of components. A ComponentAssembly is defined in terms of a set of components (subcomponents) and the set of connections that connect components.

A ComponentAssembly is defined as an implementation of a Component.

A ComponentAssembly also has a two derived attributes: ports, that contains the set of external ports of the assembly implements, and properties, that contains the set of properties of the assembly. These two attributes are derived from the component the assembly implements. The ports and properties of the implemented component must be allocated to ports and properties of sub-components contained in the ComponentAssembly.

8.2.3.2 Attributes

No additional attributes.

8.2.3.3 Associations

- **/assemblyProperty: Property** [0..*]
  Contains the set of properties of the assembly. This association is derived from the Component the assembly implements.
- **/externalPort: Port** [0..*]
  Contains the set of external ports of the assembly. This association is derived from the Component the assembly implements.
- **containedComponent: Component** [1..*]
  Describes the set of Components contained in the ComponentAssembly (i.e., subcomponents). This association is a subset of the "role" association owned by the StructuredClassifier (UML2.0::CompositeStructures::InternalStructures::StructuredClassifier).
- **ownedPortConnector: PortConnector** [*]
  Describes the set of PortConnectors owned by the ComponentAssembly. This association is a subset of the ownedConnector association owned by UML2.0::CompositeStructures::InternalStructures::StructuredClassifier.
- **ownedPropertyConnector: PropertyConnector** [*]
  Maps the external properties of the component that is implemented by the assembly to properties of subcomponent instances. Describes the set of PropertyConnectors owned by the ComponentAssembly. This association is a subset of the ownedConnector association owned by UML2.0::CompositeStructures::InternalStructures::StructuredClassifier.

8.2.4 ComponentImplementation (Stereotype)

8.2.4.1 Description

The ComponentImplementation stereotype is an extension of the UML 2.0 Class metaclass (from UML2.0::Kernel). A ComponentImplementation is an abstract class that contains the attributes and associations that are common to the different types of component implementations (MonolithicImplementation and ComponentAssembly).

A ComponentImplementation describes a specific implementation of a component interface. This implementation can be either assembly based or monolithic. The ComponentImplementation may contain configuration properties that are used to configure each component instance ("default values"). Implementations may be tagged with user-defined capabilities. Administrators can then select among implementations using selection requirements; Assemblers can place requirements on implementations.

8.2.4.2 Attributes

- **capacity: Sequence(Capacity)**
  Tags that can be used to discriminate between implementations.

8.2.4.3 Associations

- **deployRequirement: Requirement** [1..*]
  Requirements that are matched against node resources at deployment time.

8.2.5 ExternalReference (Stereotype)

8.2.5.1 Description

The ExternalReference stereotype is an extension of the UML 2.0 ConnectableElement metaclass (from UML2.0::CompositeStructures::InternalStructures). It identifies a location outside the assembly as an endpoint of a PortConnector. Whether the endpoint is a provider or user port is implied by the URL, and its type is assumed to be compatible with the connection.

8.2.5.2 Attributes

- location: **URL**
  References a port outside of the assembly that is to be an endpoint of this connection, which is resolved at execution time.

8.2.5.3 Associations

No associations.

8.2.6 PortConnector (Stereotype)

8.2.6.1 Description

The PortConnector stereotype is an extension of the UML 2.0 Connector metaclass (from UML2.0::Components::BasicComponents). A PortConnector connects a set of compatible ports.

8.2.6.2 Attributes

- **label: String**
  Optionally identifies this connection within its assembly. May be used or generated by visual design tools.

8.2.6.3 Associations

- **connectedPort: Port** [1..*]
  The set of Ports connected by the PortConnector. This association is a subset of the "end" association owned by UML2.0::CompositeStructures::InternalStructures::Connector.

- **externalReference**: **externalReference** [*]
  The set of ExternalReferences connected by the PortConnector. This association is a subset of the "end" association owned by UML2.0::CompositeStructures::InternalStructures::Connector.

8.2.6.4 Constraints

- A PortConnector connects two or more ConnectableElements, which are either of type Port or ExternalReference.

- Also, at least one of the ConnectableElements must be of type Port.

Note: Definition. Connector (from UML2.0::Components::BasicComponents): The connector concept is extended in the Components package to include interface based constraints and notation. A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events) : a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling. An assembly connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Note: One of the issues in the D&C is that a single connector can at the same time connect ports of peer components in an assembly and ports of internal components to external ports, i.e., delegation ports. So according to the UML 2.0 spec, we have connectors that have both a delegation connector capability and an assembly connector capability. The D&C concept of PortConnector is based on the ECAD (circuit design, netlist) model. It fully expresses the idea that a set of ports can be connected together just like a "signal" (say "the reset signal") can be connected to many "pins" of the components (chips) of a circuit. This allows the expression of connections that are point to point (one provider and one user) as well as those with multiple users (like many clients for one server, many event producers for one consumer), multiple providers (like a multicast channel), or multiple of both (like a multicast event channel with multiple listeners). Also, in network systems, you want to talk about a flow that represents the traffic between a set of users and providers so you can plan, manage, and configure it as a whole. If the only means of expression is point to point connections, there is no way to talk about the aggregate "connection." This "richness" has been used in network, circuit, and chip design systems for decades.

8.2.7 PropertyConnector (Stereotype)

8.2.7.1 Description

The PropertyConnector stereotype is an extension of the UML 2.0 Connector metaclass (from UML2.0::Components::BasicComponents). A PropertyConnector connects properties of a ComponentAssembly to properties of sub-Components.

8.2.7.2 Attributes

- **label: String**
  Optionally identifies this connection within its assembly. May be used or generated by visual design tools.

8.2.7.3 Associations

- **connectedProperty**: Property [2..*]
  The set of Properties connected by the PropertyConnector. This association is a subset of the “end” association owned by UML2.0::CompositeStructures::InternalStructures::Connector.

8.2.7.4 Constraints

- One of the connected Properties must be a Property of the ComponentAssembly.

8.2.8 MonolithicImplementation (Stereotype)

8.2.8.1 Description

The MonolothicImplementation stereotype is an extension of the UML 2.0 Class metaclass (from UML2.0::Kernel). It is a subclass of the ComponentImplementation stereotype. A MonolithicImplementation is a class that contains the implementation of a component.

8.2.8.2 Attributes

- **deployRequirement: Requirement [1..*]**
  Requirements that are matched against node resources at deployment time.

8.2.8.3 Associations

No additional associations.

8.2.9 Port (Stereotype)

8.2.9.1 Description

The Port stereotype is an extension of the UML 2.0 Port metaclass (from UML2.0::CompositeStructure::Ports).

The Port stereotype is defined as “required,” which means that every instance of the Port metaclass must be associated with an instance of the Port stereotype.

8.2.9.2 Attributes

- **name: String**
  The name of the port.

- **UID: String**
  The primary type of the port.

- **supportedType: Sequence(String)**
  All types supported by this port, including the primary and inherited types. All of the types listed in this attribute are acceptable for a connection.

- **provider: Boolean**
  Identifies whether the port acts in the role of provider or user, for any connection attached to it.

- **exclusiveProvider: Boolean**
  If set to true, then this port expects that there is at most one provider on the connection that it is an endpoint to.

- **exclusiveUser: Boolean**
  If set to true, then this port expects that there is at most one user on the connection that it is an endpoint to.

- **optional: Boolean**
  Identifies whether connecting this port is optional or mandatory.

8.2.9.3 Associations

No additional associations.

Note: Restriction. In UML 2.0, a Port can be associated with both required and provided interfaces. In this specification, a Port is restricted to be associated with either required interfaces (user Port) or provided interfaces (provider Port). An OCL constraint could be added to formally express this restriction.

8.2.10 Property (Stereotype)

8.2.10.1 Description

The Property stereotype is an extension of the UML 2.0 Property metaclass (from UML2.0::CompositeStructures::InternalStructures). A Property has a name and a typed value. It is used to carry named and typed values in various places. In the context of D&C, components have configuration properties.

8.2.10.2 Attributes

No additional attributes.

8.2.10.3 Associations

No additional associations.

8.2.11 Requirement

8.2.11.1 Description

Requirements are used to express the fact that an implementation artifact or connection has requirements that must be fulfilled by resources in the target environment. The resource type must match the type of a resource.

8.2.11.2 Attributes

- **resourceType**: **String**
  Identifies the resource type.

8.2.11.3 Associations

- **properties**: **Property** [*]
  Properties associated with the resource.

8.3 Package Targets

The Target package defines the set of stereotypes that are used to model a distributed deployment target. The list of stereotypes currently defined includes: Bridge, CommunicationPath, Domain, Interconnect, Node, Resource, and SharedResource.

This section defines the set of stereotypes contained in the package Targets.

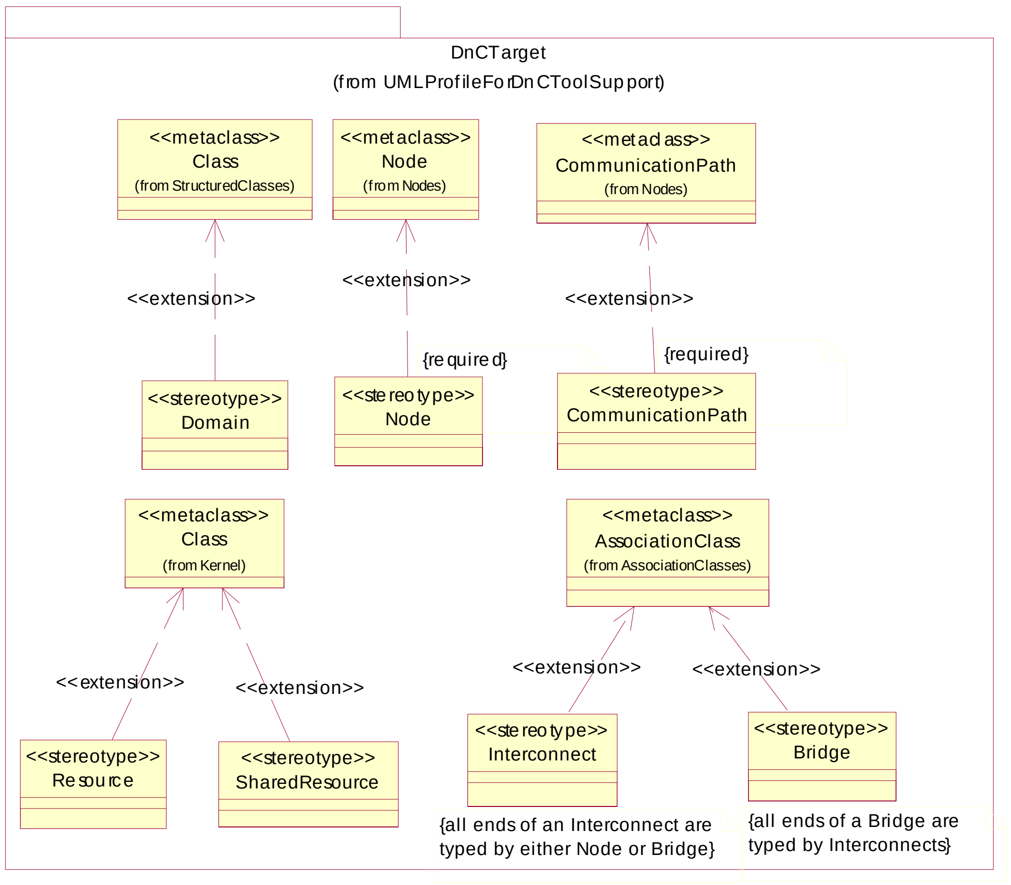
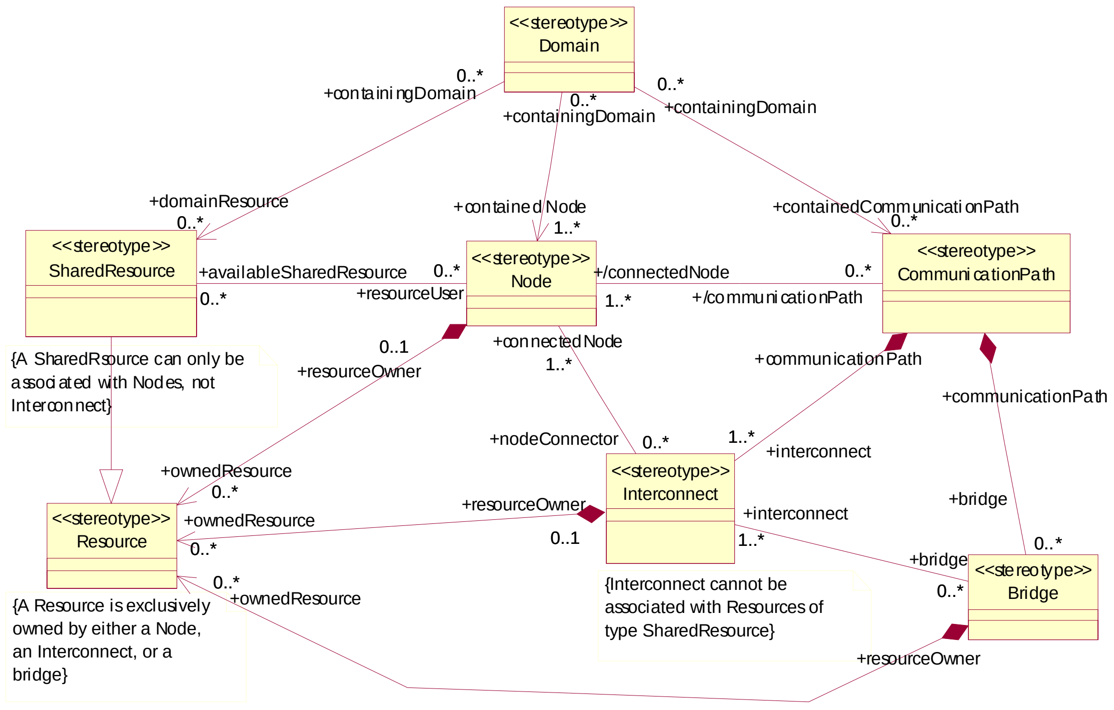

DnCTarget
(from UMLProfileForDnCToolSupport)
<<metaclass>> Class (from StructuredClasses)
<<metaclass>> Node (from Nodes)
<<metaclass>> CommunicationPath (from Nodes)
<<extension>>
<<extension>>
<<extension>>
{required}
{required}
<<stereotype>> Domain
<<stereotype>> Node
<<stereotype>> CommunicationPath
<<metaclass>> Class (from Kernel)
<<metaclass>> AssociationClass (from AssociationClasses)
<<extension>>
<<extension>>
<<extension>>
<<extension>>
<<stereotype>> Resource
<<stereotype>> SharedResource
<<stereotype>> Interconnect
<<stereotype>> Bridge
{all ends of an Interconnect are typed by either Node or Bridge}
{all ends of a Bridge are typed by Interconnects}

Figure 8.6 - Targets package



<<stereotype>> Domain
<<stereotype>> SharedResource
<<stereotype>> Node
<<stereotype>> CommunicationPath
<<stereotype>> Interconnect
<<stereotype>> Resource
<<stereotype>> Bridge
+containingDomain
+domainResource
+containedNode
+containedCommunicationPath
+availableSharedResource
+resourceUser
+/connectedNode
+/communicationPath
+resourceOwner
+connectedNode
+communicationPath
+nodeConnector
+interconnect
+ownedResource
+bridge
0..*
1..*
0..1
{A SharedRsource can only be associated with Nodes, not Interconnect}
{A Resource is exclusively owned by either a Node, an Interconnect, or a bridge}
{Interconnect cannot be associated with Resources of type SharedResource}

Figure 8.7 - Domain stereotype definition

8.3.1 Bridge (Stereotype)

8.3.1.1 Description

The Bridge stereotype extends the UML 2.0 AssociationClass metaclass (from UML2.0::AssociationClasses). A Bridge is a special type of association that connects two or more interconnects.

A Bridge exists between Interconnects to describe an indirect communication path between nodes. If a connection is to be deployed between components that are instantiated on nodes that are not directly connected, therefore requiring bridging, the connection's requirements must be satisfied by the resources of each interconnect and bridge in between.

8.3.1.2 Attributes

- **name**: **String**
  The bridge's name.

- **label**: **String**
  An optional human-readable label for this bridge.

8.3.1.3 Associations

- **interconnect**: **Interconnect** [1..*]
  The Interconnects that this Bridge provides connectivity to.

- **ownedResource**: **Resource** [*]
  Set of Resources owned by the Bridge.

- **communicationPath**: **CommunicationPath** [1]
  Reference the CommunicationPath the Interconnect belongs to.

8.3.1.4 Constraints

The name must be unique within the domain.

8.3.2 CommunicationPath (Stereotype)

8.3.2.1 Description

The CommunicationPath stereotype extends the UML 2.0 CommunicationPath metaclass (from UML2.0::Deployments::Nodes). A CommunicationPath connects two or more Nodes (as opposed to only two nodes for UML 2.0 Node). A CommunicationPath may be composed of one or more Interconnects and zero or more Bridges.

8.3.2.2 Attributes

No additional attributes.

8.3.2.3 Associations

- **interconnect**: **Interconnect** [1..*]
  Set of Interconnect contained in the CommunicationPath.

- **bridge**: **Bridge** [*]
  Set of Bridges contained in the CommunicationPath.

- **/connectedNode**: **Node** [*]
  Set of Nodes that uses the sharedResource. This association is derived from the Interconnect::connectedNode association.

8.3.3 Domain (Stereotype)

8.3.3.1 Description

The Domain stereotype extends the UML 2.0 Class metaclass (from UML2.0::CompositeStructures::StructuredClasses). A Domain is defined as a set of Nodes, CommunicationPaths, and SharedResources. In a Domain, Nodes are connected using CommunicationPaths. It represents the entire target environment.

8.3.3.2 Attributes

- **label**: **String**
  An optional human-readable label for the domain.

- **UUID**: **String**
  An optional unique identifier for this domain.

8.3.3.3 Associations

- **containedNode**: **Node** [1..*]
  Node elements that belong to the domain.

- **containedCommunicationPath**: **CommunicationPath** [*]
  CommunicationPaths that provide connections between nodes.
- **domainResource**: **SharedResource** [*]
  Shared resources that belong to the domain.

8.3.3.4 Constraints

- The top-level elements in a domain all have name attributes. These names must be unique within the domain.

8.3.4 Interconnect (Stereotype)

8.3.4.1 Description

The Interconnect stereotype extends the UML 2.0 AssociationClass metaclass (from UML2.0::AssociationClasses). It establishes connection between a set of Nodes and Bridges.

An Interconnect provides a shared direct connection between one or more nodes. It can have resources, but no shared resources. Resources are matched against a connection's requirements at deployment time.

An Interconnect that is attached to only a single node can be used to describe the loopback connection. A loopback connection is implicit; components can always be interconnected locally. Sometimes, it may be useful or necessary to describe the type(s) of available loopback connections (e.g., “shared memory”), or their resources or capabilities (e.g., latency).

8.3.4.2 Attributes

- **name**: **String**
  The interconnect’s name.
- **label**: **String**
  An optional human-readable label for the interconnect.

8.3.4.3 Associations

- **connectedNode**: **Node** [1..*]
  Set of nodes that the Interconnect is connected to.
- **bridge**: **Bridge** [*]
  The bridges that provide connectivity to other interconnects.
- **ownedResource**: **Resource** [*]
  Set of Resources owned by the Interconnect.
- **communicationPath**: **CommunicationPath** [1]
  Reference the CommunicationPath the Interconnect belongs to.

8.3.4.4 Constraints

- The name must be unique within the domain

8.3.5 Node (Stereotype)

8.3.5.1 Description

The Node stereotype extends the UML 2.0 Node metaclass (from UML2.0::Deployments::Nodes).

Nodes are connected to zero or more CommunicationPaths that enable components that are instantiated on this node to communicate with components on other nodes. Nodes may own resources and may have access to shared resources that are shared between nodes.

8.3.5.2 Attributes

- **name**: **String**
  The node's name.

- **label**: **String**
  An optional human readable label for the node.

8.3.5.3 Associations

- **nodeConnector: Interconnect [*]**
  Set of Interconnect to which the node is connected.

- **/communicationPath: CommunicationPath [*]**
  Set of CommunicationPath to which the node is connected. This association is derived from the Interconnect::communicationPath association.

- **ownedResource**: **Resource [*]**
  Set of resources owned by the Node.

- **availableSharedResource**: **SharedResource [*]**
  Set of SharedResources that the Node has access to.

8.3.5.4 Constraints

- The name of the **Node** must be unique within the **Domain** (see above).

8.3.6 Resource (Stereotype)

8.3.6.1 Description

The Resources stereotype extends the UML 2.0 Class metaclass (from UML2.0::Kernel).

Resources represent features within the target environment. They are matched against implementation requirements at deployment planning time.

8.3.6.2 Attributes

- **name**: **String**
  An optional name for the requirement satisfier.

- **resourceType**: **Sequence(String)**
  The resource types that can be satisfied by this resource.

8.3.6.3 Associations

No additional associations.

8.3.6.4 Constraints

- The name of a Resource must be unique within the container.
- A Resource is exclusively owned by either a Node, an Interconnect, or a bridge.

8.3.7 SharedResource (Stereotype)

8.3.7.1 Description

The SharedResources stereotype extends the UML 2.0 Class metaclass (from UML2.0::Kernel). It is a specialization of the Resource stereotype.

Shared resources are resources that are shared between nodes. They are semantically equivalent to "normal" resources; however, the planner must make sure that a shared resource is not exhausted by using it from multiple nodes in parallel.

8.3.7.2 Attributes

No additional attributes.

8.3.7.3 Associations

- **resourceUser**: **Node** [1..*]
  Set of nodes that have access to the SharedResource.

8.3.7.4 Constraints

- The name of the SharedResource must be unique within the domain.
- A SharedResource is a type of Resource that can only be associated with Nodes.

9 Actors

The previous chapter defined the platform independent model for deployment and configuration. The data models are used by the management interfaces for data interchange, but all model elements are passive entities. Actors manipulate the data, are clients to the interfaces, and enact the various phases of deployment. Usually, part of the actor will be implemented in software tools, aiding a (human) user in development and deployment of an application.

All actors defined by this specification are abstract. Some behavior is regulated, e.g., how data is to be processed by them, but the implementation of actors is left undefined. Some implementations of this specification might combine all actors into a single GUI, others could provide separate scripts. Some actors might be implicit parts of derived actors, others might be split across multiple sub-actors. While the deployment system described by the PIM requires actors acting as clients to perform the work of deployment and configuration, the descriptions in this section are not normative, but rather express the expected usage of the capabilities offered by the PIM. In particular, run time errors can be expected if this anticipated actor behavior is not followed. Since any bundling or communication or modularity between actors is completely undefined, constraints cannot be described that insist on the behavior described in this section.

There are three categories for actors; development, target, and deployment, mirroring the model segmentation presented earlier. Actors in the first category are concerned with the various phases of implementing a component, starting with an interface design and eventually creating a component package. Actors in the deployment category take existing component packages, and deploy them into a target environment in order to create running applications. The only actor in the target category is the Domain Administrator.

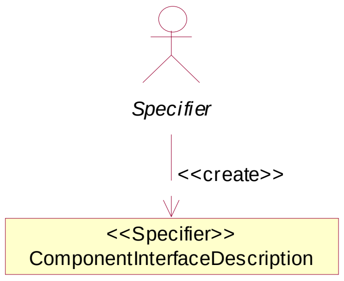
9.1 Development Actors Overview

The development of a component implementation involves the roles of Specifier, Developer, Assembler, and Packager. The Specifier creates an interface specification. Developers create a monolithic implementation of that specification, or an Assembler creates an assembly based implementation from existing subcomponents. The Packager then wraps up one or more implementations of the component interface into a component package.

This process is circular, as component packages and/or interface specifications of subcomponents are inputs to the Assembler.

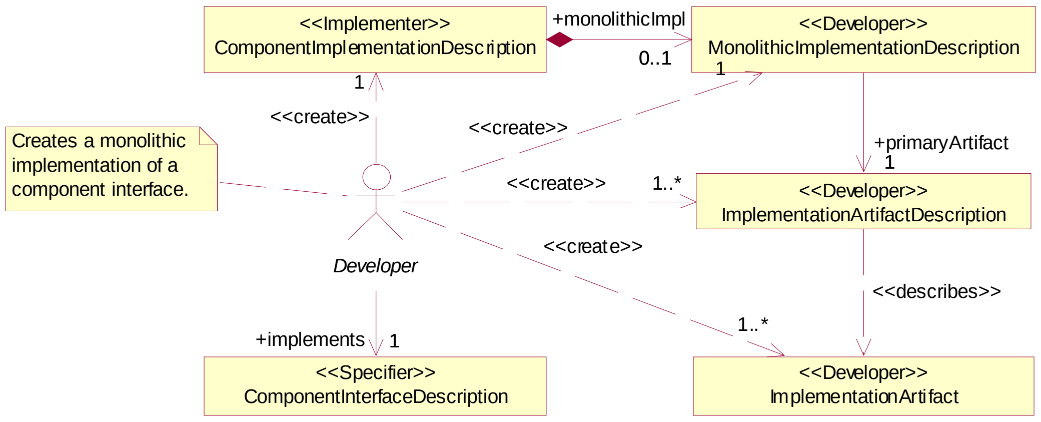
The above paragraph implies a bottom-up approach to component development, but that is not necessarily true, the flow of information can be reversed. An Implementer or Assembler can also work “downwards” from an existing component package in order to add new implementations to the package. An Assembler might then involve the Specifier in defining interface specifications for subcomponents.

9.2 Specifier



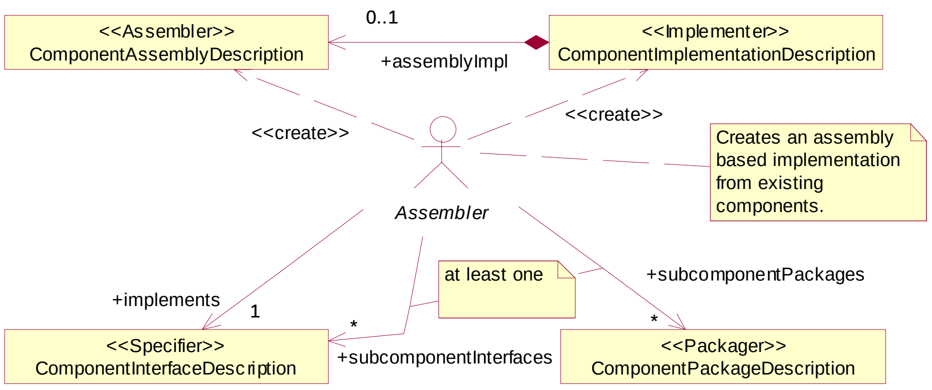
The Specifier creates an interface specification and generates a **ComponentInterfaceDescription** to describe the component interface, including its ports. Specifiers usually create other documents as well, such as PSM-specific interface descriptions (e.g., IDL files), behavioral models, and system specifications, but the **ComponentInterfaceDescription** is the only piece that is captured in this model.

9.3 Developer



The Developer creates a monolithic implementation that satisfies a specific component interface. The Developer reads the Specifier's **ComponentInterfaceDescription** and creates an implementation contained in one or more implementation artifacts. For each **ImplementationArtifactDescription**, the Developer then creates a matching **ImplementationArtifactDescription** that describes the artifact and its requirements on the target environment. The Developer then describes the component implementation as a whole by creating one **MonolithicImplementationDescription** and one **ComponentImplementationDescription** element.

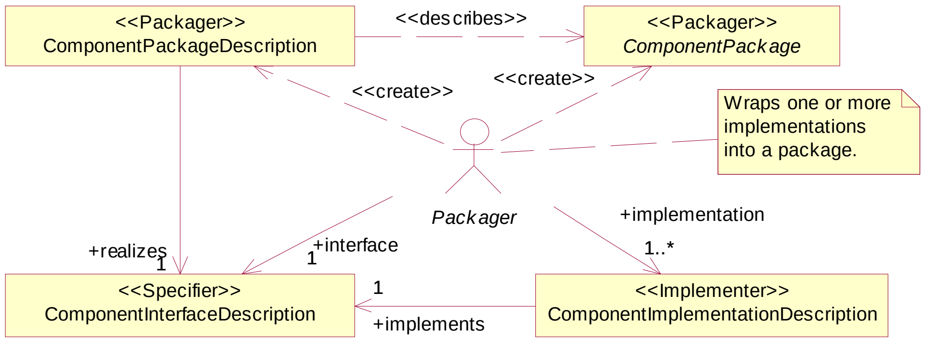
9.4 Assembler



<<Assembler>> ComponentAssemblyDescription
0..1
+assemblyImpl
<<Implementer>> ComponentImplementationDescription
<<create>>
<<create>>
Creates an assembly based implementation from existing components.
Assembler
+implements
1
at least one
+subcomponentPackages
*
*
<<Specifier>> ComponentInterfaceDescription
+subcomponentInterfaces
<<Packager>> ComponentPackageDescription

The Assembler creates an assembly based implementation of a specific component interface, using existing components as building blocks. The Assembler uses either interface descriptions for subcomponents from **ComponentInterfaceDescription** elements (expecting implementations for such interfaces to exist in the repository associated with the target domain) or concrete implementations for subcomponents from a **ComponentPackageDescription** (which implies an interface description). The Assembler configures subcomponents, interconnects them, and maps external ports and properties to ports and properties of subcomponents. The Assembler then creates a **ComponentAssemblyDescription** element to describe the assembly and a **ComponentImplementationDescription** to describe this component implementation.

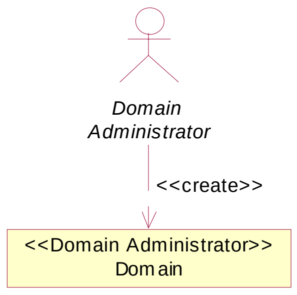
9.5 Packager



<<Packager>> ComponentPackageDescription
<<describes>>
<<Packager>> ComponentPackage
<<create>>
<<create>>
Wraps one or more implementations into a package.
Packager
+implementation
+realizes
1
+interface
1
1..*
<<Specifier>> ComponentInterfaceDescription
1
+implements
<<Implementer>> ComponentImplementationDescription

The Packager wraps multiple implementations of the same component interface into a component package. The **ComponentInterfaceDescription** and one or more **ComponentImplementationDescription** elements are input to the packaging process. The Packager ensures that the implementations' component interfaces are compatible with the desired interface. The Packager then creates a **ComponentPackageDescription**, potentially assigning default values to properties. The Packager then creates a component package that wraps all relevant descriptors and implementation artifacts. This component package is then distributed to Repository Administrators.

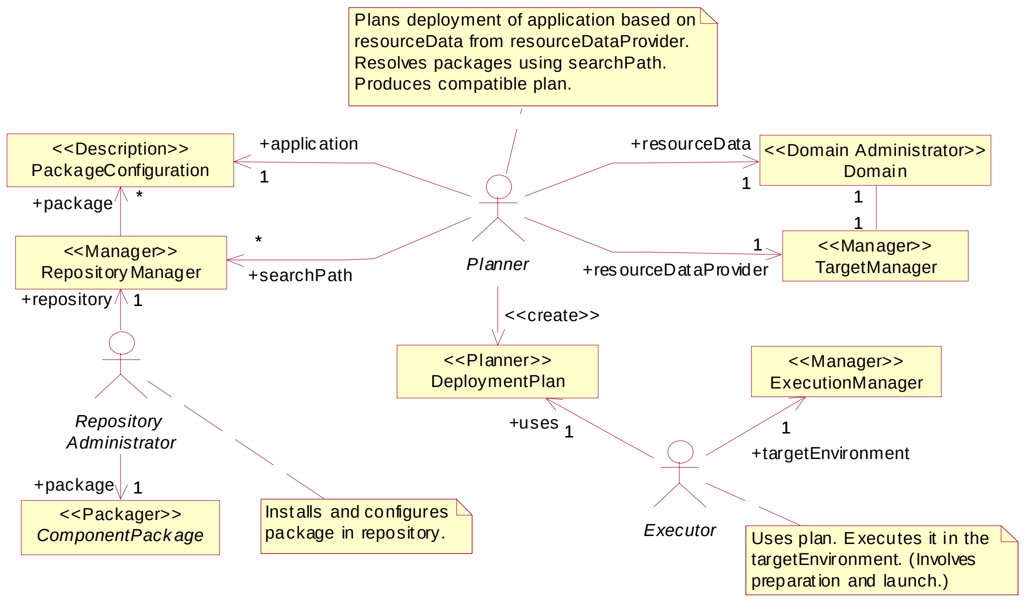
9.6 Domain Administrator



The Domain Administrator describes the local target environment and all its resources by creating a **Domain** element and then initializing a **TargetManager** with that information.

Note: In the future, the Domain Administrator role could be refined. Ideally, hardware providers would deliver descriptions for all pieces of a domain: nodes, interconnects, bridges, hardware devices, etc. The Domain Administrator would then collect that information and create a specific domain configuration. For the moment, it is safe to assume that the job of describing a domain's resources ends up with the Domain Administrator.

9.7 Deployment Actors Overview



The overview diagram above shows the three actors that are involved in the deployment of an application, the Repository Administrator, the Planner, and the Executor. The Repository Administrator receives component packages from the Packager and installs them in the local repository using the **RepositoryManager** interface. The Planner matches an implementation's requirements against available resources and creates a specific **DeploymentPlan**. The Executor uses the **DeploymentPlan** and contacts the **ExecutionManager** in order to execute the deployment and to instantiate the application. More detail is provided in the upcoming sections.

A proprietary implementation of the deployment system could merge planning and execution functionality in a single actor, immediately executing components, based on online planning, by directly using the **NodeManager** interface, without creating a **DeploymentPlan**. Such an implementation would not expose the **ExecutionManager** interface but could still use off the shelf implementations of the other compliance points.

9.8 Repository Administrator

The Repository Administrator installs a component package into a repository, and then configures the component packages within the repository.

The Repository Administrator has access to a component package via URL, and to a **RepositoryManager** via reference. The Repository Administrator calls the **installPackage** operation of the **RepositoryManager**, passing the URL of the component package. A user may provide a label for the new **PackageConfiguration**.

After installing a package in the repository, the configuration for that package may optionally be updated, or new configurations can be created. In order to update or create a configuration, the user provides configuration and selection properties, and the Repository Administrator can then use the `createConfiguration` or `updateConfiguration` operation of the **RepositoryManager** to effect the update or creation of a **PackageConfiguration**.

9.9 Planner

The Planner supports planning the deployment of an application.

The Planner has access to a specific **PackageConfiguration** via a repository reference and a name: the Planner uses the `findConfigurationByName` operation of the **RepositoryManager** to retrieve the description of the application that is to be deployed. A user might provide zero or more references to **RepositoryManager** instances as a search path to resolve **ComponentPackageReference** references in the component package. To resolve such a reference, the Planner passes the `specificType` from the **ComponentPackageReference** to the `findLabelsByUID` operation of each **RepositoryManager** in the search path and selects an appropriate configuration among all available configurations using implementation defined means. The Planner then retrieves resource data from a **TargetManager** using either the `getAllResources` or `getAvailableResources` operation. From this information, the Planner produces a **DeploymentPlan** that details a valid deployment of the application into the domain.

The Planner selects a valid **DeploymentPlan** using implementation defined means. Usually, there will be many possibilities to deploy an application into a domain, some of them equivalent – e.g., permutations of distributing component instances among homogeneous nodes, – some of them can be considered better than others – e.g., distributing computation-intensive component instances across multiple nodes rather than executing them on a single node. Selecting plans that are more appropriate than others in a given context is a quality of implementation issue, possibly influenced by user input and feedback.

A valid **DeploymentPlan** describes a deployment of an application using concrete implementations that match requested selection properties, and an assignment of these implementations to nodes so that node and interconnection resources match or exceed the requirements of component and connection instances that are deployed on them.

9.9.1 Finding Valid Deployments

To find a valid deployment, the Planner may have to consider all potential decompositions of an application, and all potential distributions. One possible algorithm is to consider a decision tree where inner nodes mark selections of specific implementations within a component package. The leaves of the tree then represent decompositions of the application into monolithic implementations. For each decomposition, the Planner then has to consider all possibilities for distributing component instances among all nodes until a valid deployment is found. Pseudo code for this algorithm follows.

1. Initialize a "decision queue" with the top-level package that is to be deployed. This queue will contain packages for which we still have to decide on an implementation. Recurse into the algorithm, initializing it with the one-element decision queue, starting at step 2. If the recursion fails, there is no valid deployment.
2. Remove the first element from the queue, which identifies a **ComponentPackageDescription**.
3. For each concrete implementation in the package, go to step 4 to find a valid deployment. If that fails, backtrack.
4. Match the capabilities of this **ComponentImplementationDescription** against the relevant selection requirements (see below). On the top level, i.e., for the implementations of the top-level component, selection requirements are found in the **PackageConfiguration**. On other levels, i.e., for implementations of subcomponents in an assembly, the selection requirements are found in the **SubcomponentInstantiationDescription**. If they are not compatible, return to step 3 and continue iterating over other implementations in this package.
5. If the implementation is assembly-based, then add the packages that provide implementations for its subcomponents to the decision queue.
6. If the decision queue is not empty, then the application is not fully decomposed yet. Recurse to step 2. If recursion fails, return to step 3.
7. If the decision queue is empty, then the application has been fully decomposed into monolithic implementations by the decisions made in step 3. The Planner now has to consider potential instantiations.
8. Iterate over all permutations of assigning component instances to nodes. For each permutation, go to step 9 to see whether it identifies a valid deployment. If that fails, backtrack.
9. For each component instance, consider the node it has been assigned to. Match the requirements defined by its monolithic implementation against the node's resources (see below). If that fails, return to step 8 to consider other permutations.
10. For each connection between component instances, match its connection requirements against the interconnect and bridge resources that provide the connection between the nodes that the component instances have been assigned to (see below). If there is no path between the nodes, or if the interconnects and bridges are not capable of hosting the connection, return to step 8.
11. Otherwise, the deployment is valid.

This specification does not impose any requirements on the Planner implementation. The algorithm above is designed to find a valid deployment if one exists. It has been included for informative purposes and is not normative. Obviously, there are many techniques for narrowing the search space and for considering more likely implementations and permutations first, but still, the number of possibilities might be too large to be practical. Planners are not required to traverse the full search space – that's a quality of implementation issue. Planners are also free to either stop after finding a first valid deployment or to continue searching and to select among valid deployments – possibly with user feedback.

Steps 4, 9 and 10, the matching of selection properties and the matching of requirements against resources, are defined in the following sections.

Note: Steps 2, 3 and 5 assume that in order to find a concrete implementation for a component, only a single package is considered. However, Planner implementations might consider multiple packages when resolving **ComponentPackageReference** elements. Again, this is implementation specific.

9.9.2 Matching Selection Requirements

Both **PackageConfiguration** and **SubcomponentInstantiationDescription** define selection requirements that are matched against implementation capabilities in the **ComponentImplementationDescription** for all implementations in the referenced **ComponentPackageDescription**.

For each **Requirement**, the Planner checks whether the **ComponentImplementationDescription** has a **Capability** whose `resourceType` attribute includes the `resourceType` attribute of the **Requirement**. If not, then the implementation cannot satisfy the requirements.

The **Requirement** is then matched against the **Resource** as described below.

9.9.3 Matching Implementation Requirements

A component instance's requirements are defined as the sum of all deployment requirements in its **MonolithicImplementationDescription**, the **ImplementationArtifactDescription** of its primary artifacts and all directly or indirectly dependent **ImplementationArtifactDescription** elements (excluding duplicates). The "sum" of all requirements is the concatenation of all **Requirement** elements into a single list.

For each **Requirement**, the Planner checks whether the **Node** has a **Resource** (or **SharedResource** – resources and shared resources are treated the same) whose `resourceType` attribute includes the `resourceType` attribute of the **Requirement**. If not, then the **Node** is not capable of hosting the component implementation.

The **Requirement** is then matched against the **Resource** as described below.

9.9.4 Matching Connection Requirements

Connection requirements are described as part of an assembly in the **deployRequirement** attribute of the **AssemblyConnectionDescription**. Connections between two component ports can be made up of multiple segments if the two components belong to different assemblies, e.g., two segments to connect the components to external ports of their respective assemblies, and another segment to connect the two components (that are implemented by the assemblies) in the assembly-based implementation of a supercomponent. In that case, the requirements for the connection is the sum of all deployment properties of all its segments. The "sum" of all requirements is the concatenation of all **Requirement** elements into a single list.

Note: Considering point-to-point connections between two ports is the worst-case scenario. In some domains, if a connection has more than two endpoints, part or all of the communication path could be shared – e.g., if events are broadcast using UDP. Planners that are aware of this situation can account for capacities appropriately. Connection requirements must be matched against the resources of the interconnects and bridges that the connection is routed over, as defined by the communication path between the nodes that the components that are the endpoints to the connection are instantiated on.

Note: This specification assumes that a single communication path is implied by its two endpoints.
For each **Requirement**, the Planner checks whether all **Interconnect** and **Bridge** elements in the communication path have a **Resource** whose `resourceType` attribute includes the `resourceType` attribute of the **Requirement**. If not, then routing the connection is not possible.

The **Requirement** is then matched against all these **Resource** elements as described below. If any match fails, then routing the connection is not possible.

9.9.5 Matching a Resource against a Requirement

For every **Property** that is part of the **Requirement**, there must be a **SatisfierProperty** among the property elements of the **Resource** whose **name** attribute equals the **name** attribute of the requirement's property. If there is no **SatisfierProperty** of matching name, then the **Resource** cannot satisfy the **Requirement**.

Each **Property** is then matched against the **SatisfierProperty** according to the rules set forth for the kind of **SatisfierProperty**, as described in the documentation for **SatisfierPropertyKind**, to determine if the meets this specific requirement.

The **Resource** meets the **Requirement** if and only if the **Resource** above test succeeds for all **Property** elements that are part of the **Requirement**.

9.10 Executor

The Executor supports preparation of a **DeploymentPlan** and the launch of the application, possibly, but not necessarily, in a single step.

For preparation, the Executor reads the **DeploymentPlan** and passes it to the `preparePlan` operation of the **ExecutionManager**. The Executor stores the **DomainApplicationManager** reference that is returned.

To launch an application, the Executor remembers the **DomainApplicationManager** reference that was the result of preparation, and calls the `startLaunch` operation, passing configuration properties if desired. The **DomainApplicationManager** returns a **DomainApplication** reference and the connections that are provided by the application on external ports.

The Executor then calls the `finishLaunch` operation on the **DomainApplication**, passing connections to the application's external ports if desired.

The Executor can either set the start parameter to the `finishLaunch` operation to true in order to start the **DomainApplication**, or it can later call the `start` operation separately.

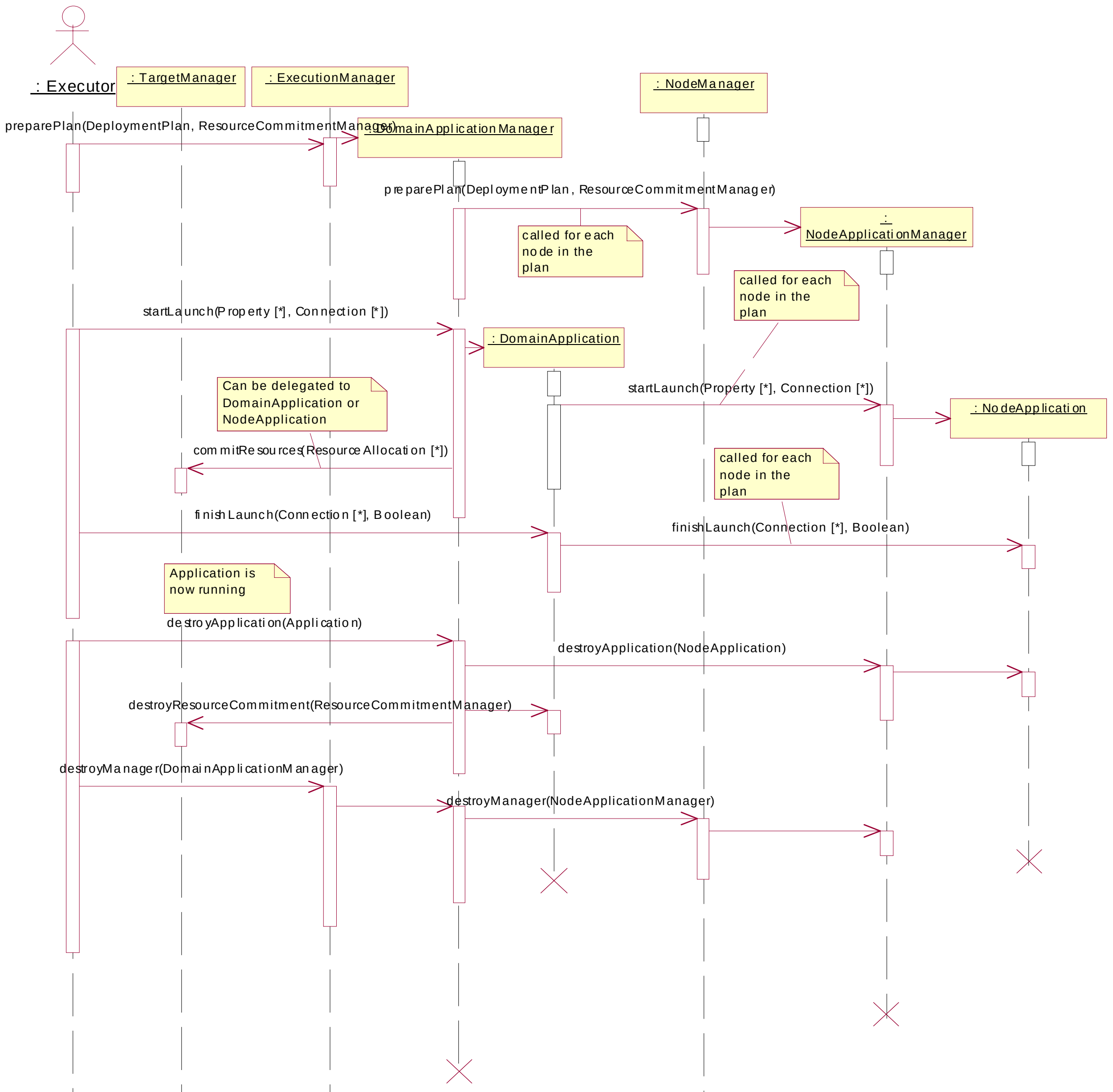

: Executor
: TargetManager
: ExecutionManager
: NodeManager
: DomainApplicationManager
: NodeApplicationManager
: DomainApplication
: NodeApplication
preparePlan(DeploymentPlan, ResourceCommitmentManager)
preparePlan(DeploymentPlan, ResourceCommitmentManager)
called for each node in the plan
called for each node in the plan
startLaunch(Property [*], Connection [*])
startLaunch(Property [*], Connection [*])
Can be delegated to DomainApplication or NodeApplication
commitResources(ResourceAllocation [*])
called for each node in the plan
finishLaunch(Connection [*], Boolean)
finishLaunch(Connection [*], Boolean)
Application is now running
destroyApplication(Application)
destroyApplication(NodeApplication)
destroyResourceCommitment(ResourceCommitmentManager)
destroyManager(DomainApplicationManager)
destroyManager(NodeApplicationManager)

Figure 9.1 - Executor in Action

Figure 9.1 shows the sequence of events that are exchanged between the Executor and the deployment system as well as events within the domain.

10 PSM for CCM

10.1 Introduction

This chapter describes the mapping of the platform-independent model for Deployment and Configuration to the CORBA Component Model platform **[CCM]**. It is intended to be a replacement for the Packaging and Deployment chapter of the CCM specification in CORBA 3.0 as well as the XML DTD chapter of **[CCM]**. Issues of migration and compatibility to this previous CCM deployment specification are addressed in Section 10.8, “Migration Issues,” on page 140, ““Migration Issues”” on page 140.

The D&C data models are used in two different ways, first for persistent storage and distribution of information, and second for representing data at runtime. For persistent storage and distribution, the data models are mapped to XML schemas **[XSD]**, so that information can be stored in XML files **[XML]** according to the model. We frequently use the term (and stereotype) *description* for the classes that define the data model. We use the term “*descriptor*” to refer to the XML file that contains the data. For runtime, the data models are mapped to IDL data structures.

The management classes are runtime entities and mapped to IDL interfaces only.

This section does not include XML schema and IDL files, since both are generated according to rules. However, these files are supplied with this specification to show the results of this rule-based file generation. The rules that will be used to auto-generate these files from the platform independent model use stereotype classes and associations appropriately and then use rules set forth in the UML profile for CORBA.

This chapter defines three transformations and two mappings.

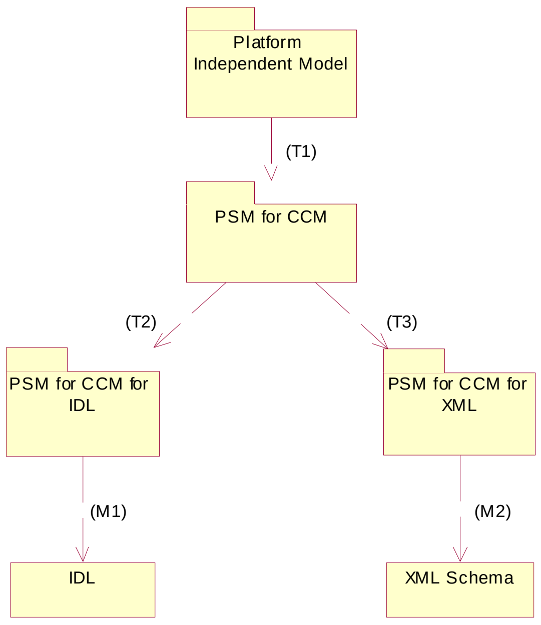


Figure 10.1 - Model Transformations and Mappings for CCM

The first transformation, T1 (*PIM* to *PSM for CCM*), takes the platform-independent model, and refines it into a platform specific model for CCM. In this *PSM for CCM*, the abstract meta-concepts are concretized, and also some other classes are aligned with the CORBA Component Model.

The second transformation T2 (*PSM for CCM* to *PSM for IDL*) takes the *PSM for CCM* and transforms it into a *PSM for CCM for IDL* that can be used to generate concrete IDL from the model. The third transformation T3 (*PSM for CCM* to *PSM for CCM for XML*) creates a *PSM for CCM for XML* that can be used to generate concrete XML schemas.

The motivation for transformations T2 and T3 is to transform the PIM into PSMs so that generic, rule-based mappings M1 and M2 can be used. (Note that some classes have different representations in IDL and XML, for example the <ClassName>Any class, prohibiting IDL and XML schema generation from the same model.) The motivation for transformation T1 is that some CCM specific transformations are necessary that are independent of the mapping to IDL or XML.

The M1 mapping is realized using the UML Profile for CORBA **[UPC]**, the M2 mapping is realized using the XML Metadata Interchange (XMI) Version 2 **[XMI]** specification, Chapter 2, “XML Schema Production.”

10.2 Definition of Meta-Concepts

This section provides a concrete definition for the classes that are abstract in the PIM. This section is unrelated to the transformations, which will be described in the following sections.

10.2.1 Component

The abstraction of **Component** in the PIM is mapped to both components and homes for the CCM platform. Components in CCM have an interface, attributes and ports. Homes do not have ports, but an interface and attributes. Both components and homes have explicitly “supported” interfaces in addition to the “equivalent” interface, that inherits all supported interfaces, and includes attributes and explicit operations in the component and home interface definitions.

Viewing homes as a kind of component allows this specification’s model to deploy homes (by themselves or as part of an assembly). Applications or other components in an assembly can then use the home to create component instances at runtime. This supports the full feature set of CCM, without requiring explicit home implementations.

If a CCM home or component supports an interface, their **ComponentInterfaceDescription** has a special port named “`supports`” that can be used in connections for any of the “supported” interfaces. If, in an assembly, a connection is to be provided by any of the component’s or home’s supported interfaces, then the port name of the **ComponentExternalPortEndpoint** or **SubcomponentPortEndpoint** class is “`supports`.” For CCM homes, this port also provides their equivalent interface. The “`supports`” port for CCM components does *not* provide the equivalent interface, since this would be problematic for assembly implementations of components. Home implementations are always monolithic. (Note that in CCM 3.0, assemblies did not allow connections to a component’s equivalent interface either.)

Configuration properties of components, as described by the **ComponentPropertyDescription** class, are attributes in the component or home interface or any inherited component or home interface, but not in any supported interface.

Note: The “`supports`” magic name has been chosen because it reflects the supported interface. Because it is an IDL keyword, it has little likelihood of conflicting with other port names.

10.2.2 ImplementationArtifact

The meta-concept of **ImplementationArtifact** is mapped to a file accessible by URL. This PSM still treats files as opaque. Agreement between the author of an implementation and the **NodeManager** over the contents of an implementation artifact is assumed. This agreement, or “contract,” is expressed in terms of execution parameters and an implementation’s dependencies on resources provided by the node.

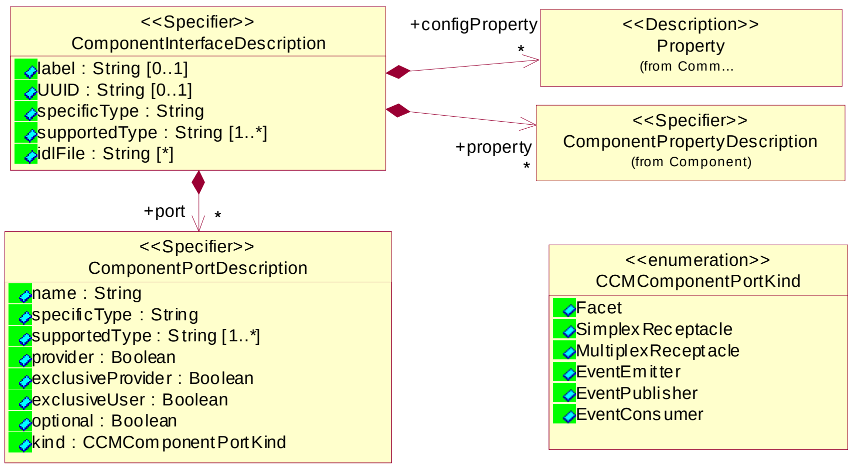
10.2.3 Package

The meta-concept of a package is mapped to a ZIP file **[ZIP]** accessible by URI **[URI]**, that includes implementation artifacts and descriptors. Packages have the “`.cpk`” extension and must contain a single Toplevel Package Descriptor containing a <ClassName>ToplevelPackageDescription element with the magic name “`package.tpd`.”

10.3 PIM to PSM for CCM Transformation

This section defines transformation T1 (as described in the introduction for this chapter). It takes the platform-independent model from Chapter 7 and aligns classes with the CORBA Component Model. This involves changes to attributes, associations, and semantics of some classes. All classes from the PIM that are not refined here are imported into the PSM for CCM without change.

10.3.1 ComponentInterfaceDescription



The **ComponentInterfaceDescription** and **ComponentPortDescription** classes are augmented to support CCM.

The `idlFile` attribute is added to the **ComponentInterfaceDescription**. The `idlFile` attribute, if present, contains alternative URIs that reference an IDL file containing the component’s (or home’s) interface definition. The IDL file is not used within the deployment infrastructure; it may be included in a package for convenience. Since deployable applications have a component interface, some tools that deploy and execute such applications might need the IDL to interact with the ports of the application’s component interface.

The `kind` attribute is added to the **ComponentPortDescription** class and specifies the concrete port kind that is used. This information is required by the **NodeManager** and by assembly tools. In CCM, `EventConsumer` and `Facet` ports are considered providers, the other ports are users.

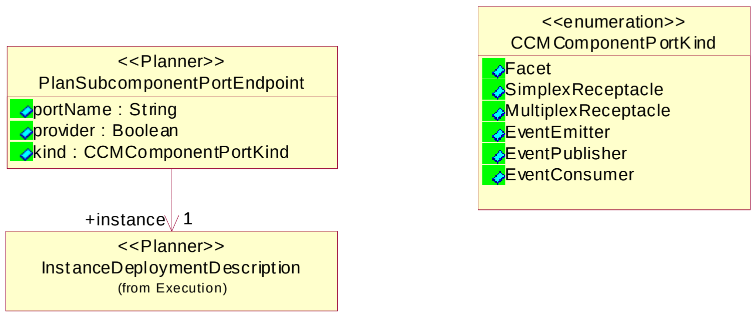
Repository Id strings are used to identify interface types, i.e., for the `specificType` and `supportedType` attributes.

For `Facet` ports, `supportedType` lists the Repository Id of the provided interface and any of its base interfaces that the developer (or tool) chooses to expose for connections. For receptacles, `supportedType` lists the Repository Id of the accepted interface. For `EventEmitter` and `EventPublisher` ports, `supportedType` lists the Repository Id of the accepted consumer interface. For `EventConsumer` ports, `supportedType` lists the Repository Id of the consumer interface and any of its base interfaces that the developer (or tool) chooses to expose for connections.

If the component or home supports one or more interfaces, this will be reflected by a **ComponentPortDescription** element of kind Facet with the magic name "`supports`." The `specificType` attribute is left empty, the `supportedType` attribute lists the Repository Id of any of its supported interfaces and base interfaces that the developer wants to expose for connections.

Initially, a **ComponentInterfaceDescription** can be generated from a component's or home's IDL description with a defined set of configuration properties (from attributes) and default values for the `exclusiveProvider`, `exclusiveUser` and `optional` attributes. If desired, a user can then adjust these three attributes for each port and also add configuration property default values to the **ComponentInterfaceDescription** by adding **Property** elements to the `configProperties` list.

10.3.2 PlanSubcomponentPortEndpoint



The `kind` attribute augments the `provider` attribute in the **PlanSubcomponentPortEndpoint** class and specifies the concrete port kind that is used. This information is required by the various managers in the Execution Management Model. The `provider` attribute still indicates a port which provides an object reference.

10.3.3 Application

The **start** operation on the **Application** class performs the `configuration_complete` operation in all component instances that are part of the application.

10.3.4 RepositoryManager

When artifact files are included in the package (as opposed to referenced via URL outside the package), the **RepositoryManager** must make its own copy of these artifacts during the `installPackage` operation. It must substitute an URL that references this copy of the artifact in the `location` attribute of **ImplementationArtifactDescription** elements delivered via its interface.

10.3.5 SatisfierProperty

This PSM has to define concrete types that are implied on the value of a **SatisfierProperty** by the **SatisfierPropertyKind**, and on the value of the **Property** that is matched against the satisfier.

- For the `Quantity` kind, the value of the **SatisfierProperty** is of type `unsigned long`. The value of the **Property** is ignored.
- For the `Capacity` kind, the value of the **SatisfierProperty** is of type `unsigned long` or `double`. The value of the **Property** must be of the same type.
- For the `Maximum` and `Minimum` kinds, the value of the **SatisfierProperty** is of type `long` or `double`. The value of the **Property** must be of the same type.
- For the `Attribute` kind, the value of the **SatisfierProperty** is of type `long`, `double`, `string`, or an enumeration type. In the case of long, double or string, the value of the **Property** must be of the same type. If the value of the **SatisfierProperty** is of enumeration type, the value of the **Property** is of type `string`, containing the enumeration value that must compare equal to the **SatisfierProperty** value.
- For the `Selection` kind, the value of the **SatisfierProperty** is a sequence of type `long`, `double`, `string`, or an enumeration type. The same rules as for the `Attribute` kind apply.

10.4 PSM for CCM to PSM for CCM for IDL Transformation

This section defines transformation T2 (as described in the introduction). It transforms the *PSM for CCM* into a *PSM for CCM for IDL* that can be used to generate concrete IDL using a rule-based mapping. Classes from the *PSM for CCM* are transformed to match the UML Profile for CORBA. Its rules are then used to generate concrete IDL.

The first subsection describes generic mapping rules that are applied to all classes that are part of the *PSM for CCM*. The second subsection defines special transformation rules for the classes that are abstract in the PIM.

All classes in the *PSM for CCM for IDL* are placed in the **Deployment** package, so that all resulting IDL structures and interfaces will be part of the `Deployment` IDL module.

10.4.1 Generic Transformation Rules

The mapping to IDL is accomplished using the rules set forth in the UML Profile for CORBA. To apply these rules, the stereotypes used in the platform-independent model are mapped to stereotypes for which a mapping is defined in the profile. The «**Description**» stereotype and all that inherit from it are mapped to the «**CORBAStruct**» stereotype; these classes are therefore mapped to CORBA structures. The «**Exception**» stereotype is mapped to the «**CORBAException**» stereotype; such classes become CORBA exceptions. The «**Enumeration**» stereotype is mapped to the «**CORBAEnum**» stereotype in order to become enum types in IDL. The «**Manager**» stereotype is mapped to the «**CORBAInterface**» stereotype so that these classes become CORBA interfaces.

To avoid redundancy and circular graphs, non-composite associations between classes with a common owner are expressed by an ordinal attribute at the source (navigating) end, with the name of the attribute being the role name plus the suffix “`Ref`,” and the type “`unsigned long`.” The value of this attribute is the index of the target element in its container, with the index of the first element being 0 (zero). To enable the usage of an index, the composition of the target element in its container is qualified with the “ordered” constraint.

Wherever the multiplicity of an attribute, parameter or return value is not exactly one (but 0..1, 1..* or *), a new class is introduced to represent a sequence of the type of the attribute, parameter or return value. The sequence class has the «**CORBASequence**» stereotype, and its name is the english plural of the name of the type. The sequence class has a composition association with the element class that is navigable from the sequence to the element. The composition is qualified with the index of the sequence. The attribute, parameter or return value is then replaced with an attribute, parameter or return value, respectively, with the same name as before, but with the type being the newly introduced sequence class and the exactly one (1..1) multiplicity.

A similar rule is applied to all navigable association or composition ends whose multiplicity is not exactly one (but 0..1, 1..* or *): a new class is introduced to represent a sequence of the class at the navigable end; this sequence class is defined as describe above. The original association or composition end is then replaced with a navigable association or composition end, with the same role name as before, at the new sequence class, with a multiplicity of exactly one (1..1). According to the rules in the UML Profile for CORBA, these associations and compositions will then map to a structure member in IDL, its type being a named sequence of the referenced type.

Excepted from the two rules above are attributes, parameters, return values or navigable association or composition ends where the type is **String**, `unsigned long` or **Endpoint**. Instead of defining new sequence types, the existing types in the CORBA package are being used; see below.

Note that in combination, these rules map non-composite associations between classes with a common owner and a multiplicity other than 1 to sequence of “`unsigned long`” type.

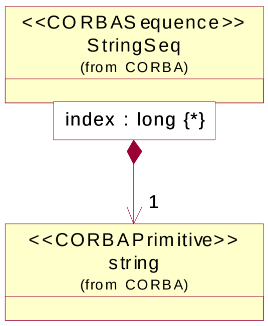
Another exception from the rule above are attributes of type **String** with the 0..1 (zero or one) multiplicity. In this case, the multiplicity is updated to 1..1 (exactly one). If the value is missing in an XML representation of the model, the empty string is used as default value.

The inheritance relationships of classes with the «**Description**» stereotype (**SharedResource**, **Resource** and **Capability**) classes are removed; all attributes and associations of the base class are attached to the derived class.

Associations of classes with the «**Manager**» stereotype are removed from the *PSM for CCM for IDL*.

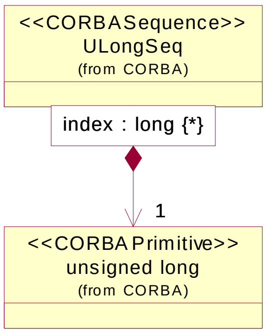
10.4.2 Special Transformation Rules

10.4.3 Sequence of String



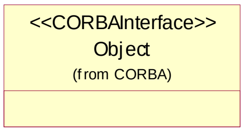
A type representing a sequence of strings already exists in the **CORBA** package and can be re-used. Wherever the **String** type is used with a multiplicity other than exactly one, it is mapped to the **StringSeq** class from the CORBA package as shown above. It then maps to the `CORBA::StringSeq` type in IDL (from the `orb.idl` file).

10.4.4 Sequence of unsigned long



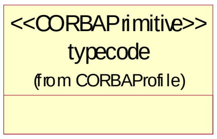
A type representing a sequence of the `unsigned long` type already exists in the **CORBA** package and can be re-used. Wherever the `unsigned long` type is used with a multiplicity other than exactly one, it is mapped to the **ULongSeq** class from the CORBA package as shown above. It then maps to the `CORBA::ULongSeq` type in IDL (from the `orb.idl` file). Sequences of the unsigned long type occur when a non-composite association between classes with a common owner with a multiplicity other than one occurs, according to the generic rule above.

10.4.5 Endpoint



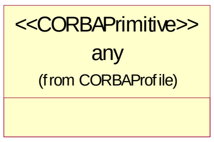
The abstract **Endpoint** class is mapped to the **Object** class from the **CORBA** package. It will therefore map to the `Object` type in IDL.

10.4.6 DataType



The abstract **DataType** class is mapped to the **typecode** class from the **CORBAProfile** package. It then maps to the `TypeCode` type in IDL.

10.4.7 Any



The abstract **Any** class is mapped to the **any** class from the **CORBAProfile** package. It will then map to the `any` type in IDL.

10.4.8 Primitive Types

The UML data types **String**, **Integer** and **Boolean** are mapped to the classes **string**, **long** and **boolean** in the **CORBAProfile** package, respectively. They will then map to the `string`, `long` and `boolean` types in IDL, respectively.

10.4.9 Mapping to IDL

After applying the transformations defined in this section, IDL is generated by applying the rules set forth in the UML Profile for CORBA specification **[UPC]**.

10.5 PSM for CCM to PSM for CCM for XML Transformation

This section defines transformation T3 (as described in the introduction). It transforms the *PSM for CCM* into a *PSM for CCM for XML* that can be used to generate a concrete XML schema using the mapping rules described in Chapter 2, “XML Schema Production” of the XML Metadata Interchange (XMI) Version 2 **[XMI]** specification.

10.5.1 Generic Transformation Rules

Data model elements, annotated with the «**Description**» or «**enumeration**» stereotype (or a stereotype that inherits from it), are used to generate an XML schema for representing metadata in XML documents for distribution, interchange or persistence. The only normative use of such XML-based metadata in this specification is for installing component packages using the **RepositoryManager**’s `installPackage` operation.

Management model elements, annotated with the «**Manager**» or «**Exception**» stereotype, are not part of the PSM for CCM for XML, they are mapped to IDL only.

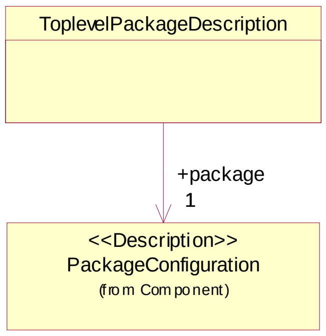
All classes in the PSM for CCM for XML are annotated with the “`org.omg.xmi.contentType`” tag set to the value “`complex`.”

All attributes are annotated with the “`org.omg.xmi.element`” tag set to “`true`.”

All packages are annotated with the “`org.omg.xmi.nsURI`” tag set to “`http://www.omg.org/Deployment`” and the “`org.omg.xmi.nsPrefix`” tag set to the value “`Deployment`.”

10.5.2 Special Transformation Rules

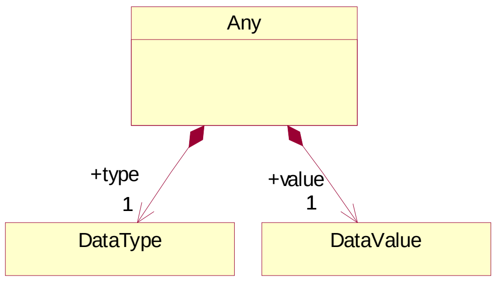
10.5.3 ToplevelPackageDescription



The <ClassName>ToplevelPackageDescription is introduced to point to the **PackageConfiguration** element for the top-level component package in a package.

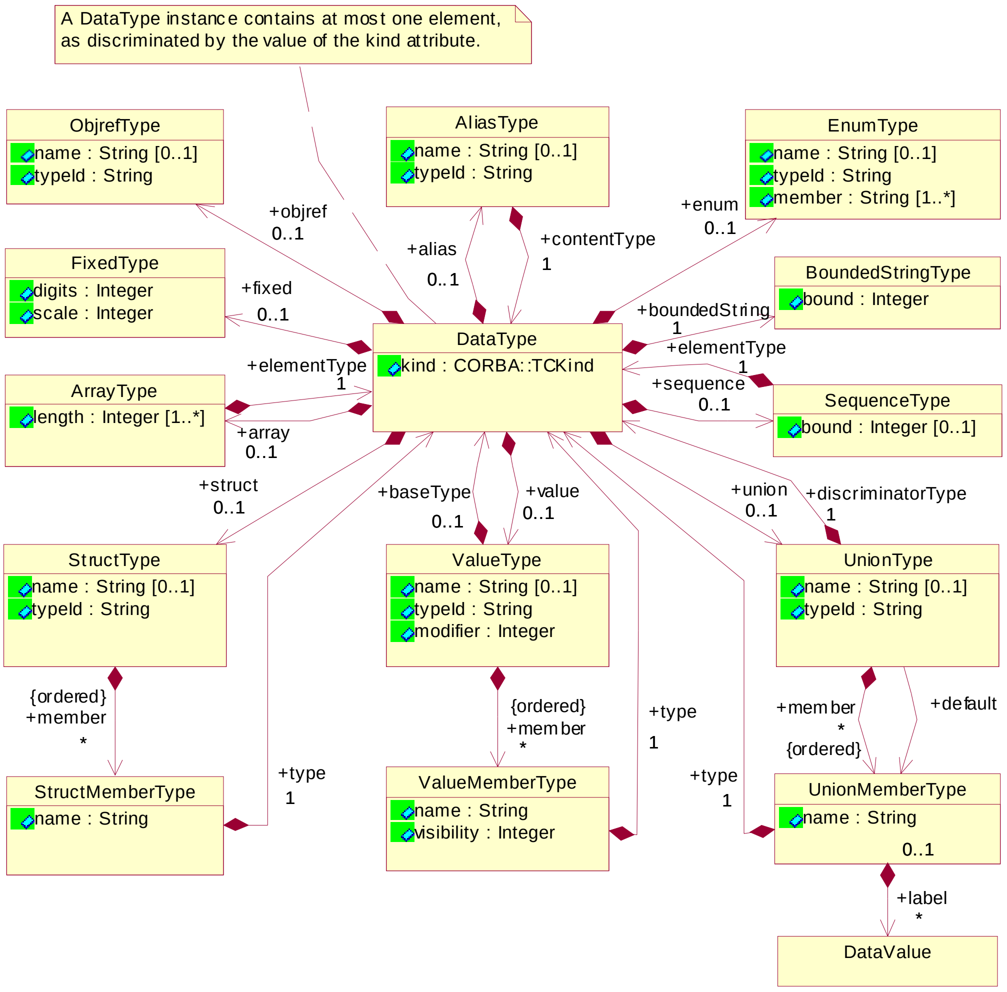
The motivation for this element is that a package may include component packages for sub-components. A selection mechanism is necessary to distinguish the top-level component package. This is accomplished by including a single Toplevel Package Descriptor with the magic name “`package.tpd`” into the package.

10.5.3.1 Any



An **Any** instance describes a typed value. It is mapped to a class that contains a <ClassName>DataType and a <ClassName>DataValue, which are elaborated below.

10.5.4 DataType



A <ClassName>DataType instance describes a type. It is mapped to a hierarchical structure as shown above, describing available types in IDL.

The **DataType** class contains a `kind` field that indicates the IDL type described by a **DataType** instance. The `kind` is of the enumeration type `CORBA::TCKind`, as defined in Section 4.11 (Type Codes) of the CORBA specification, v3.0.3 (available from http://www.omg.org/technology/documents/formal/corba_iiop.htm).

If the kind is `tk_null`, `tk_void`, `tk_short`, `tk_long`, `tk_ushort`, `tk_ulong`, `tk_float`, `tk_double`, `tk_boolean`, `tk_char`, `tk_octet`, `tk_any`, `tk_TypeCode`, `tk_longlong`, `tk_ulonglong`, `tk_longdouble` or `tk_wchar`, the **DataType** element does not contain any other elements.

If the kind is `tk_string` or `tk_wstring`, then the **DataType** may optionally contain a **BoundedStringType** element indicating the upper bound for the string length. If the **DataType** does not contain a **BoundedStringType**, an unbounded string is assumed.

If the kind is `tk_objref`, `tk_component` or `tk_home`, then the **DataType** may optionally contain an **ObjrefType** element describing the object reference's type (using its Repository Id). If the **DataType** does not contain an **ObjrefType** element, then an untyped object reference (Repository Id "`IDL:omg.org/CORBA/Object:1.0`") is assumed.

If the kind is `tk_struct` or `tk_except`, then the **DataType** contains a **StructType** element, which in turn describes a list of struct members.

If the kind is `tk_union`, then the **DataType** contains a **UnionType** element. **UnionType** contains the type of the descriminator and a number of typed elements, one of which may be the default member. Each member may be identified with multiple case labels. No label is associated with the default member.

If the kind is `tk_enum`, then the **DataType** contains an **EnumType** element describing the enumeration values.

If the kind is `tk_sequence`, then the **DataType** contains a **SequenceType** element. Its optional `bound` attribute indicates the sequence's upper bound. If the `bound` attribute is absent, the sequence is unbounded.

If the kind is `tk_array`, then the **DataType** contains an **ArrayType** element. Its `length` attribute indicates the length of the array. For multi-dimensional arrays, the multiplicity of the length attribute is greather than one, and the most significant dimension is listed first ("left to right" in IDL).

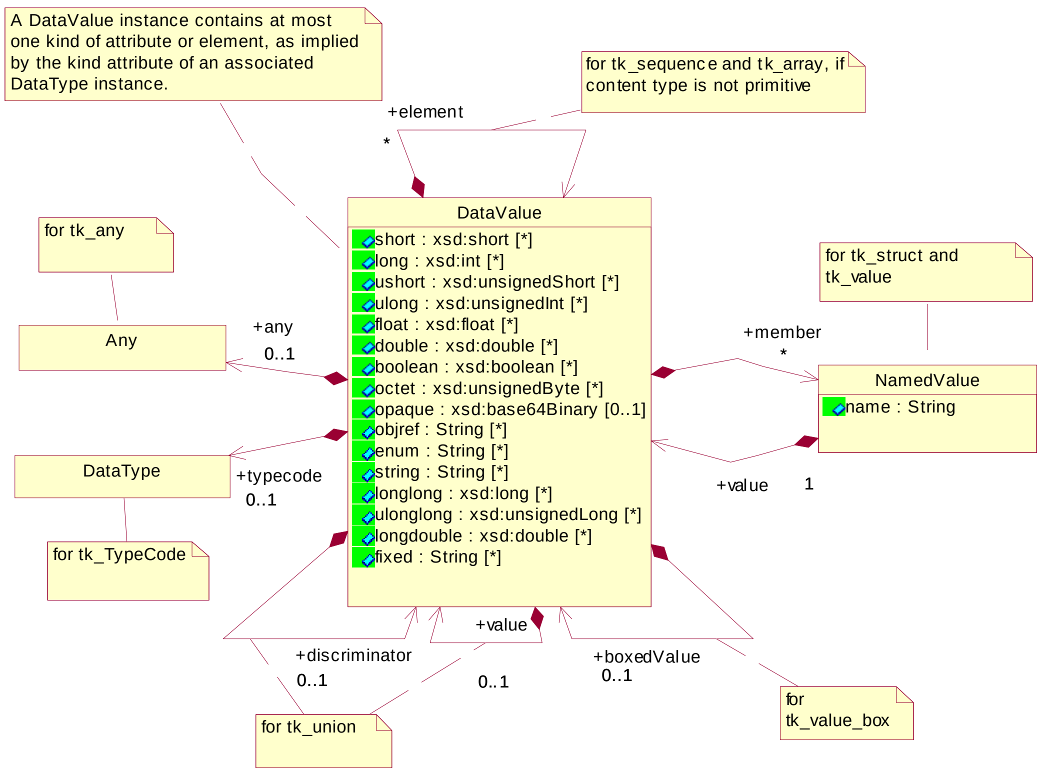
If the kind is `tk_alias` or `tk_value_box`, then the **DataType** contains an **AliasType** element.

If the kind is `tk_fixed`, then the **DataType** contains a **FixedType** element.

If the kind is `tk_value`, then the **DataType** contains a **ValueType** element. **ValueType** contains the type code of the concrete base type, if any, a type modifier (with values as defined by `CORBA::ValueModifier`), and a number of members. Each member has a name, type and visibility (with values as defined by `CORBA::Visibility`).

In **StructType**, **ValueType** and **EnumType**, the name attribute contains the name of the `struct`, `valuetype` or `enum` IDL type, and the `typeId` attribute contains its Repository Id.

10.5.5 DataValue


A DataValue instance contains at most one kind of attribute or element, as implied by the kind attribute of an associated DataType instance.
for tk_sequence and tk_array, if content type is not primitive
+element
*
DataValue
short : xsd:short [*]
long : xsd:int [*]
ushort : xsd:unsignedShort [*]
ulong : xsd:unsignedInt [*]
float : xsd:float [*]
double : xsd:double [*]
boolean : xsd:boolean [*]
octet : xsd:unsignedByte [*]
opaque : xsd:base64Binary [0..1]
objref : String [*]
enum : String [*]
string : String [*]
longlong : xsd:long [*]
ulonglong : xsd:unsignedLong [*]
longdouble : xsd:double [*]
fixed : String [*]
for tk_any
Any
+any
0..1
DataType
+typecode
0..1
for tk_TypeCode
for tk_struct and tk_value
+member
*
NamedValue
name : String
+value
1
+discriminator
0..1
+value
0..1
+boxedValue
0..1
for tk_union
for tk_value_box

The **DataValue** class describes a value. It is mapped to a hierarchichal structure as above, fully describing a value that can be described by an IDL type. A **DataValue** cannot exist by itself, it needs a matching <ClassName>DataType to describe its structure (see the <ClassName>Any class).

If the type's kind is `tk_null` or `tk_void`, **DataValue** is empty.

If the type's kind is `tk_short`, `tk_long`, `tk_ushort`, `tk_ulong`, `tk_float`, `tk_double`, `tk_boolean`, `tk_octet`, `tk_string`, `tk_longlong`, `tk_ulonglong`, or `tk_longdouble`, **DataValue** contains a single `short`, `long`, `ushort`, `ulong`, `float`, `double`, `boolean`, `octet`, `string`, `longlong`, `ulonglong` or `longdouble` attribute, respectively. If the type's kind is `tk_wstring`, then **DataValue** also contains a `string` element.

If the type's kind is `tk_char` or `tk_wchar`, the **DataValue** contains a `string` attribute containing a string of length 1.

If the type's kind is `tk_enum`, the **DataValue** contains the enumeration value in the `enum` attribute.

If the type's kind is `tk_objref`, `tk_component` or `tk_home`, the **DataValue** contains a stringified object reference in the `objref` attribute.

If the type's kind is `tk_fixed`, the **DataValue** contains a `fixed` attribute holding a fixed-point decimal literal.

If the type's kind is `tk_sequence` or `tk_array`, and the sequence's or array's element type is equivalent to (i.e., not considering aliased types) `tk_short`, `tk_long`, `tk_ushort`, `tk_ulong`, `tk_float`, `tk_double`, `tk_boolean`, `tk_octet`, `tk_objref`, `tk_enum`, `tk_string`, `tk_longlong`, `tk_longlong`, `tk_ulonglong`, `tk_longdouble`, `tk_wstring`, `tk_fixed`, `tk_component` or `tk_home`, then the respective attribute has a multiplicity equal to the length of the sequence or array. In the case of multi-dimensional arrays, the least significant dimension is enumerated first.

If the type's kind is `tk_sequence` or `tk_array`, and the sequence's or array's element type is equivalent to `tk_char` or `tk_wchar`, then the **DataValue** contains a single `string` attribute. Each character in this string is used as an element of the sequence or array.

If the type's kind is `tk_sequence` or `tk_array`, and the sequence's or array's element type is equivalent to `tk_octet`, then the **DataValue** contains a single `opaque` attribute.

If the type's kind is `tk_sequence` or `tk_array`, and the sequence's or array's element is not of the types enumerated above, then the **DataValue** contains the elements of the sequence or array as <ClassName>DataType elements, using the `element` association.

If the type's kind is `tk_TypeCode` or `tk_any`, the **DataValue** contains a <ClassName>DataType or <ClassName>Any element, respectively.

If the type's kind is `tk_struct` or `tk_value`, the **DataValue** contains a **NamedValue** for each member of the structure or valuetype.

If the type's kind is `tk_union`, the **DataValue** contains a single **DataValue** as the union's discriminator, and zero or one **DataValue** elements, using the `value` association, as the member of the union.

If the type's kind is `tk_value_box`, the **DataValue** contains zero or one **DataValue** elements using the `boxedValue` association. If the `boxedValue` element is missing, a null value is implied.

10.5.6 Others

The **PackageConfiguration**, **DomainUpdateKind**, **Connection** and **Endpoint** classes are used by the runtime models only and are not part of the PSM for XML.

10.5.7 Transformation Exceptions and Extensions

Metadata for a component package is usually spread out across several XML files, which are called descriptors. Certain associations are expected to be expressed by links within the same document, others are expected to link across documents. XMI takes care of both patterns by way of “proxies,” which do not contain nested elements but a link attribute (either “`href`” or “`xlink:href`”) referencing the target element by URI. A schema produced using the XMI rules for schema production allows proxies to appear anywhere.

Composition associations in UML express that the class at the composite end (the containing class) owns and contains the class at the part end (the contained class). It is typical, in XML documents, for instances of contained classes to be embedded within the instance of the containing class. However, it is also possible to store contained instances by themselves in a separate file by using a proxy (using "`href`" or "`xlink:href`") to reference the contained instance in a separate file. Since the multiplicity on the composite end of a composite association is always one to one in this specification, contained instances can only have a single such proxy reference.

For non-composite associations between classes with a common owner (composite end of composition), the definition of the class at the source end of the association must contain a proxy linking to the element at the target end of the association. The definition of the class at the source end cannot contain the definition of the element at the target end, because it is owned by the common owner, and its identity cannot be duplicated.

Non-composite associations between classes that do not have a common owner are usually expressed by the element definining the class at the source end containing a proxy that links to an external document. Direct containment is possible but may result in duplicated data.

While tools can decide to either combine information into a single XML document or to place information into arbitrary files, using XMI proxies to link to that information, it is expected that some model elements usually appear as the outermost document element of a standalone XML file. These commonly used descriptors are assigned descriptive terms and standard file extensions.

- A Component Package Descriptor contains a **ComponentPackageDescription** document element; it has the "`.cpd`" file extension.
- A Component Implementation Descriptor contains a **ComponentImplementationDescription** document element; it has the "`.cid`" file extension.
- An Implementation Artifact Descriptor contains an **ImplementationArtifactDescription** document element; it has the "`.iad`" file extension.
- A Component Interface Descriptor contains a <ClassName>ComponentInterfaceDescription document element; it has the "`.ccd`" (CORBA Component Descriptor) file extension.
- A Domain Descriptor contains a **Domain** document element; it has the "`.cdd`" (Component Domain Descriptor) file extension.
- A Deployment Plan Descriptor contains a **DeploymentPlan** document element; it has the "`.cdp`" (Component Deployment Plan) file extension.
- A Package Configuration Descriptor contains a **PackageConfiguration** document element; it has the "`.pcd`" file extension.
- A Toplevel Package Descriptor contains a <ClassName>ToplevelPackageDescription document element; it has the "`package.tpd`" file name.
- Package files use the "`.cpk`" extension.

Spreading information across files according to these patterns allow better reuse, for example by extracting an implementation from a package.

Proxies follow the linking semantics specified by XMI **[XMI]**. If a URI reference **[URI]** does not contain a fragment identifier (the "`#id_value`" part), then the target of the reference is the outermost document element of an descriptor file.

10.5.8 Interpretation of Relative References

URI references **[URI]** are used by proxies and appear in the `location` attribute of the **ImplementationArtifactDescription** and **ArtifactDeploymentDescription** classes and the `idlFile` attribute of the <ClassName>ComponentInterfaceDescription class.

XML documents that are part of a Component Package can use relative-path references (i.e., URIs that do not begin with a scheme name or a slash character) to reference documents and other artifacts within the same package.

The interpretation of relative URIs that are not relative-path references (i.e., network-path references that start with two slash characters, or absolute-path references that start with a single slash character), the interpretation of relative-path references that reference documents outside the package (by way of "`..`" path segments), and the interpretation of relative-path references in documents that are not contained in a Component Package (e.g., a Deployment Plan Descriptor) is implementation-specific. (Note: this allows XML processors to supply arbitrary Base URIs that do not necessarily relate to any file system but that must expose the Component Package's hierarchical structure.)

10.5.9 Mapping to XML

After applying the transformations defined in this section, an XML schema is generated by applying the rules set forth in the XML Metadata Interchange specification, Chapter 2, "XML Schema Production." **[XMI]**

10.6 Miscellaneous

10.6.1 Entry Points

CCM's Packaging and Deployment chapter in CORBA 3.0 **[CCM]** defines a home factory entry point that enables a container to create a user-defined home using a user-defined factory.

This specification defines the interaction between an implementation artifact and the execution manager as implementation-dependent, in order to not restrict the forms that an implementation artifact might have – executable files, loadable libraries, source files or scripts, for example.

However, to ensure source code compatibility in the common case without restricting implementation choice, entry points are defined here if the language is C++ and the implementation artifact is a shared library, or if the language is Java and the implementation artifact is a class file. In these two cases, there must be a specific execution parameter associated with the Monolithic Implementation Description.

If the instance to be deployed is a component, then the name of the execution parameter shall be "`component factory`." The parameter is of type **String**, and its name is the name of an entry point that has no parameters and that returns a pointer of type `Components::EnterpriseComponent`.

If the instance to be deployed is a home, then the name of the execution parameter shall be "home factory." The parameter is of type **String**, and its name is the name of an entry point that has no parameters and that returns a pointer of type `Components::HomeExecutorBase`.

For backwards compatibility, it is recommended that the name of the entry point should be the name of the component or home, prefixed with "`create_`" (e.g., "`create_Account`" for an Account component).

If the language is C++, then the entry points shall be qualified as '`extern "C"`'.

These well-defined entry points ensure that the user code for the entry point does not need to be changed when building components for different target environments. These definitions do not enable interoperability between containers and DLLs (even assuming the same compiler and ORB), thus additional interfaces are still required that are specific to container implementations. This implies that, as in CCM 3.0, component and home implementation DLLs are specific to the container implementation (and the code generation tools). Since there was and is no normative interoperability interfaces within a node, thus further implies that there is no vendor segmentation boundary within a node at all.

10.6.2 Homes

Note that this specification does not depend on the existence of homes; using the entry points defined above, a container is able to create component instances directly, without the need of creating a home first, and then using it as a factory for the component instance.

This is no loss in comparison to the Packaging and Deployment chapter of CCM in CORBA 3.0. If a component instance is to be deployed as part of an assembly, the container has no way of providing a user-defined home with any parameters, and is therefore limited to keyless homes. However, a factory operation for the component instance as defined above can do its job as well as the parameter-challenged `create` operation that is part of a keyless home.

In contrast to the Packaging and Deployment chapter, this specification recognizes homes as instances that can be deployed, and therefore enables the full range of home features.

10.6.3 Valuetype Factories

If an **ImplementationArtifact** contains valuetype factories, then its list of execution parameters shall include an element with the name “valuetype factories” and of type **ValuetypeFactoryList**, which is defined as

```
module Deployment {
    struct ValuetypeFactory {
        string repid;
        string valueentrypoint;
        string factoryentrypoint;
    };
    typedef sequence<ValuetypeFactory>
        ValuetypeFactoryList;
};
```

Each element of that sequence describes a valuetype factory that needs to be registered with the ORB in order to demarshal user-defined valuetypes. The `repid` field specifies the Repository Id of the valuetype created by the valuetype factory. The `factoryentrypoint` field specifies the name of an entry point that can be be used to create an instance of the valuetype factory. If `valueentrypoint` is not the empty string, it specifies an entry point that can be used to create an instance of the valuetype.

If the language is C++, then the entry points shall be qualified as ‘`extern "C"`’.

10.6.4 Discovery and Initialization

The **ExecutionManager** must be able to find the **NodeManager** instances for all nodes in the **Domain**, so that it is able to deploy applications according to deployment plans that are based on the current contents of the Target Data Model. This is accomplished using the Naming Service.

- The user of the deployment system creates a naming context for a domain. Note that a naming context is expressible by a URL representation (e.g., a “`corbaname:`“ reference).
- Implementations of the **ExecutionManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish its own reference with the name “`ExecutionManager`” and the empty string as the id in that context.
- Implementations of the **TargetManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish its own reference with the name “`TargetManager`” and the empty string as the id in that context.
- Implementations of the **NodeManager** interface must accept the address of the naming context as a configuration parameter, and use it to publish their own reference with the node’s name as the name and the id “`NodeManager`.” The node’s name must match the name attribute of the node in the Target Data Model.

Upon startup, the **ExecutionManager** finds the **TargetManager** in the Naming Service and accesses the current **Domain** information. Based on the **Node** elements that are contained in the **Domain**, the **ExecutionManager** then calls the `joinDomain` operation of each **NodeManager**.

An **ExecutionManager** may offer functionality to "add" new nodes to the domain, or to remove nodes from the domain. In that case, the **ExecutionManager** looks up a **NodeManager** with a user-provided name in the Naming Service and then calls its `joinDomain` or `leaveDomain` operation, respectively. In addition, an **ExecutionManager** may offer to scan the Naming Service context for previously unregistered nodes, calling the `joinDomain` operation on each associated **NodeManager**.

Note that there is no direct relationship between domains and repositories. Therefore, implementations of the **RepositoryManager** interface are not registered in the Naming Service.

10.6.5 Location

URI references **[URI]** are handled by the **RepositoryManager** and **NodeManager** interfaces: the **RepositoryManager** receives URLs to packages as a parameter to the `installPackage` operation and must generate URLs pointing to itself in **ImplementationArtifactDescription** elements. The **NodeManager** receives URLs as attributes of the **ArtifactDeploymentDescription** elements that are part of the **DeploymentPlan**.

Both **RepositoryManager** and **NodeManager** shall be able to interpret URLs according to the `http` scheme. Additional schemes may optionally be supported.

Note: This requires **RepositoryManager** implementations to include both an `http` server and an `http` client **[HTTP]**. **NodeManager** need to implement `http` clients only, in order to download implementation artifacts from the repository.

The **RepositoryManager** must supply a "http" URI as part of the location attribute in the **ImplementationArtifactDescription** elements. A **RepositoryManager** may optionally include other alternative locations to provide **NodeManager** implementations with a choice of transports to use for downloading artifacts.

10.6.6 Segmentation

This specification obsoletes CCM's idea of component segmentation. In the original CCM, assemblies provided just a single level of decomposition. Segments then offered a second level to split the implementation of a component into several independent pieces of code. This specification allows composition and decomposition on any level, and therefore the ability to add another level of decomposition on the lowest level is redundant. However, no parts of this specification inhibit a component author from using this feature of the CCM Implementation Framework.

10.7 Impact on the CCM Specification

This specification is intended to replace the Packaging and Deployment chapter and the XML DTD chapter of CCM 3.0 **[CCM]**.

Note: The Packaging and Deployment chapter of CCM 3.0, in its Component Deployment section, defines interfaces that are involved in the deployment of components onto nodes. Similar interfaces might be useful in implementing the NodeManager, however, this specification does not prescribe any such node-level interfaces.

The potential ability to create component instances without homes requires that the `get_ccm_home` operation in the `CCMObject` interface is allowed to return a nil object reference.

10.8 Migration Issues

This section deals with the issues of migrating from the Packaging and Deployment model that exists in CCM 3.0 **[CCM]** to the deployment model presented in this specification.

10.8.1 Component Implementations

The portable parts of CCM component implementation source code remains untouched. The generated code to enable interactions with the containers may change, requiring recompilation and linking. The non-portable hand written code in some implementations which was written assuming a particular container implementation would likely have to change — similar to porting the component to a different CCM system.

10.8.2 Component and Assembly Packages and Metadata

The metadata is changed to be based on XML schemas, and the basic models are different. Many lower level elements are not different, and it is expected that meta-data transformation (forward migration) will be able to be automated in the common cases where all the features used are supported.

This specification is kept simple in anticipation that broad (and necessarily complex) software packaging and distribution standards do not exist, and the W3C OSD specification (by Microsoft and Marinba in 1997) referenced by the original CCM specification did not become a standard. Future RFPs may want to consider mappings from such comprehensive standards into this simpler model that focuses on CCM applications.

The component data model stays within the scope of deployment and configuration and does not bring forward all the metadata aspects in the previous CCM specification that were not relevant to deployment and configuration. Furthermore, much of the metadata for informing containers of the requirements of component instances was not defined as part of an intervendor boundary. Thus this specification assumes the use of two “private” channels of information between the development tools (and code generation) and the runtime environment (NodeManager). These are the resource requirements of the MonolithicImplementationDescription and the execParameters of the InplementationArtifactDescription. The authors believe standardizing this metadata should be part of a true effort at vendor segmentation between CCM development tools and CCM runtime environments (assuming the same compiler and ORB), which does not exist and was not the mandate of this RFP.

Beyond the necessity of validating configuration and connection among components, the one other metadata interoperability issue is to standardize the vocabulary for selection criteria, which is interoperation between users and implementers of component software. This is currently deferred due to the concurrence of the other specification for this language with this specification (see below).

10.8.3 Component Deployment Systems

Deployment systems need to be changed to support this specification. Most aspects of container implementations should be reusable.

10.9 Metadata Vocabulary

10.9.1 Implementation Selection Requirements

Selection requirements, part of both the **PackageConfiguration** and **SubcomponentInstantiationDescription** classes, express requirements that are meant to drive the selection among alternative implementations. The user of an implementation (creator of a package configuration or an assembly) is requesting services to be satisfied by a component implementation. The mechanism defined in this specification requires agreement of the vocabulary of these services on both sides, but there is no interoperable vocabulary defined. The currently active RFP entitled “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms” **[UMLQOS]** should result in, among other things, “a Definition of Individual QoS Characteristics,” which should provide an appropriate vocabulary to drive this mechanism.

When this QoS-driven vocabulary is connected to the CCM PSM, some other component metadata requirements, such as “humanlanguage” may also be added to the selection criteria language.

10.9.2 Monolithic Implementation Resource Requirements

As mentioned above, this vocabulary is a private communication channel between development tools and the NodeManager, since no other interoperability boundary exists between these two. Obviously some standardization could be easily done, based on previous CCM-defined metadata such as container supported persistence, transactions, and POA policies. If this limited scoping is not accepted by the Task Force, data model classes containing this type of information can easily be added to support both a defined resource vocabulary and even a separate container-services vocabulary for information that would never be part of a “resource finding” matching process with the target nodes, but needs to be conveyed to the runtime environment for component instances.

Index