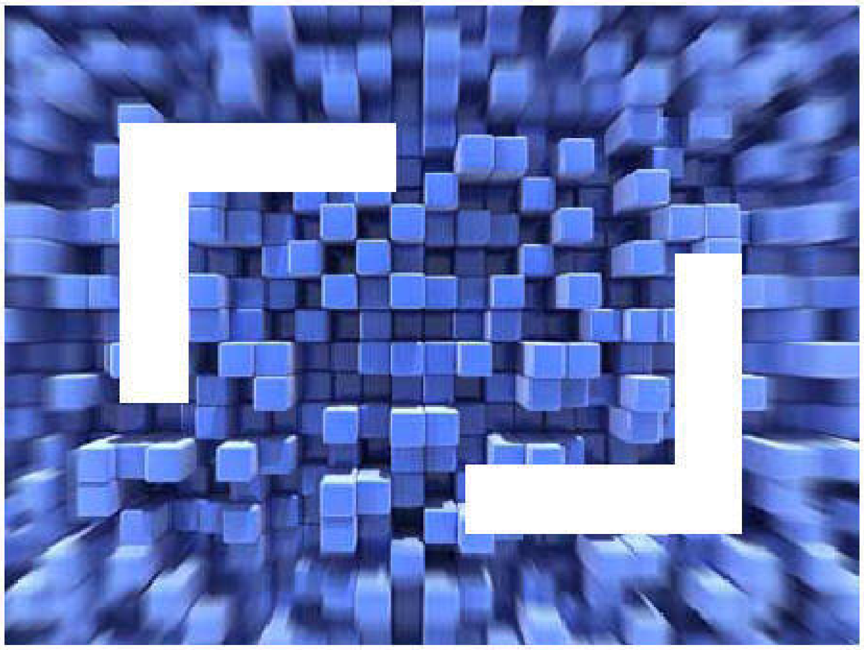


Borland Together 2008

Borland Together UML 2.1 Guide

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®
Leading the Evolution™

Borland Software Corporation
4 Hutton Centre Dr., Suite 900
Santa Ana, CA 92707

Contents

Concepts

This section contains information about UML 2.1 concepts, capabilities, and best practices.

Related Topics

UML 2.1 Overview

The Unified Modeling Language (UML) is a tool maintained by the Object Management Group (OMG) that lets you visually specify the design of software systems using standardized diagrams. With its most recent version, UML 2.1, the UML standard for defining software systems has been functionally extended to address toolsmith issues, including the implementation of redefinition and bidirectional associations in the metamodel.

UML 2.1 provides a set of Graphical Modeling Framework (GMF) editors based on the MDT UML2 metamodel. Currently, UML 2.1 supports eight of the thirteen basic diagram types. With the following diagram editors, divided into two functional sets, you can view and edit models:

- Behavior – These diagrams depict the interaction of model elements and the different states of the model as its design is applied.
  - Activity – Similar to a flowchart, these show the flow of activities used to accomplish a single task.
  - State Machine – Depict the possible states of a model element and the transitions that cause a change in the state.
  - Use Case – Describe scenarios that a system does, from the point of view of an external observer, in order to accomplish a single task or goal. These diagrams emphasize *what* a system does rather than *how*.
- Structures – These diagrams depict the static architecture of models, which include the elements that comprise the model and their relationships.
  - Class – Depict an overview of a system by showing the relationships among its classes rather than what happens when they do interact.
  - Component – Show the high-level physical analogs of a system and their structural relationships and interfaces. These diagrams are typically used to model the logical components of a system rather than its physical attributes, which are best handled by deployment diagrams.
  - Composite Structure – Show the configuration of interconnected parts that collaborate to perform the behavior and achieve the purpose of the containing classifier.
  - Deployment – Depict the hardware functionality of a software system and how it gets executed.
  - Profile Definition – Let you define all the domain model's non-UML properties and provide a way for metaclasses from existing metamodels to be extended so they can be adapted for different purposes.

The following resources offer information and assistance:

- *Eclipse Model Development Tools (MDT) – UML2Tools*
- *Eclipsepedia MDT-UML2Tools FAQ*
- *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*
- *Mailing list: mdt-uml2tools.dev*

Note: Since Together 2008 Release 3, the feature set Together UML 2.1 Diagrams is not required to be installed. When not present, the corresponding parts of the user interface and functionality are not available. Refer to the installation instructions in the Release Notes document for additional info about the product installation process.

Related Topics

Concepts on page 5

UML 2.1 Implementation in Together

Even in its latest version of UML, the OMG faces usability issues because of its patchwork contributions from varying groups of vendors and distinct legacy notations. These issues include language complexity, notational inconsistency, semantical ambiguity, and a handicapped interchange format that limits model portability between modeling tools.

The Together architecture streamlines some of these challenges. Because of Together's compatibility with XML Metadata Interchange (XMI), you can import projects or sections of projects that were created in other modeling tools. Diagrams of such projects can benefit from Together's sophisticated layout capabilities, which other modeling tools lack. In addition to its support of UML 2.x and XMI format import and export, Together modeling offers Object Constraint Language (OCL) support to describe expressions that cannot be otherwise expressed by diagrammatic notation on UML models.

Another way Together meets the challenges posed by UML is its Quality Assurance feature, which provides a wide variety of audits and metrics for measuring how well your model adheres to the UML constraints you have configured.

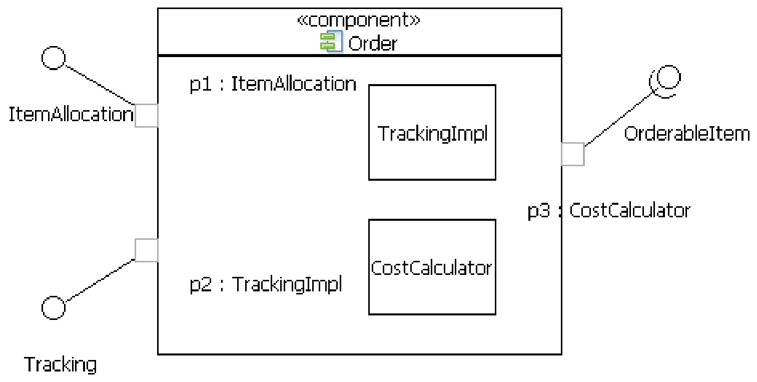
Related Topics

Concepts on page 5

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

Provided and Required Interface Links of a Port

The provided and required interfaces define behavior of a component. You can optionally organize them through ports. As demonstrated in the following image, a provided interface is displayed using the lollipop notation, and a required interface is shown using the socket notation.



In the preceding image, the **Order** component *provides* interfaces **ItemAllocation** and **Tracking** and *requires* interface **OrderableItem**. The interfaces are exposed through ports **p1**, **p2** and **p3**.

Related Topics

Concepts on page 5

Required Interface on page 7

Provided Interface on page 8

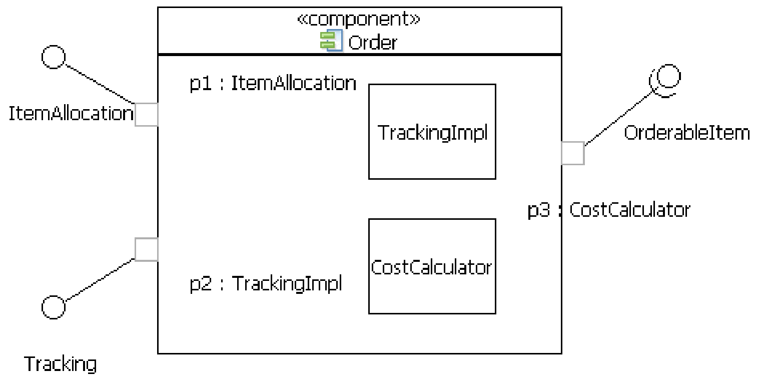
OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

Creating a Required Interface on page 54

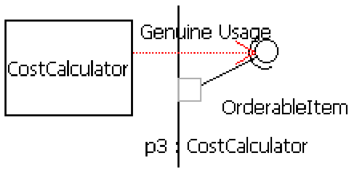
Managing the Visibility of Provided and Required Interface Links on page 55

Required Interface

Required interfaces are the interfaces that a component requires from other components in its environment so that it can offer a full set of provided functionality. The required interface link is a derived link. For a port element, the value of a Required Interface is derived from the interfaces realized by the type of the port, or by the type of the port itself if the port was typed by an interface. In the following image of the **Order** component, the **p3** port requires the **OrderableItem** interface.



In the domain editor, no link from port **p3** to the interface **OrderableItem** exists. However, you can provide a Usage link from the **CostCalculator** class to the **OrderableItem** interface, as demonstrated in the following image.



CostCalculator is a type of **p3** port, and that port requires all interfaces that **CostCalculator** uses. In the case of this component, this includes an **OrderableItem**.

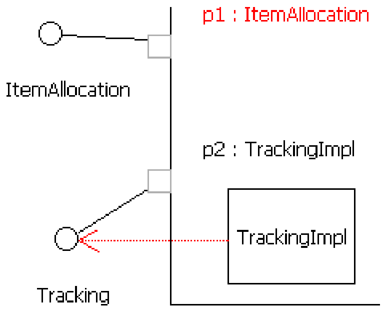
Related Topics

Provided and Required Interface Links of a Port on page 6

Creating a Required Interface on page 54

Provided Interface

Provided Interfaces are the interfaces that a component exposes to its environment. Like the Required Interface, the provided interface link is also a link whose association is derived from the interfaces realized by the type of the port, or by the type of the port itself if the port was typed by an interface. As seen in the following image of the original **Order**, port **p1** provides an **ItemAllocation** interface because the interface is a type of the port. Port **p2** provides interface **Tracking**, because there is an **InterfaceRealization** link that goes from the type of the port (**TrackingImpl**) to interface **Tracking**.



Related Topics

Provided and Required Interface Links of a Port on page 6

UML 2.1 Diagrams

Together provides support for the most frequently needed diagrams and notations defined by UML 2.1. For a thorough understanding of these diagrams, users should be familiar with the latest UML specification, available from the *Catalog of OMG Modeling and Metadata Specifications*.

Because several of the features that UML 2.0 provides (such as documentation functionality and metrics) are not yet implemented in UML 2.1, the UML 2.1 capabilities are disabled by default. To turn them on, select **Window ➤ Preferences... ➤ General ➤ Capabilities**. Click **Advanced...** and select the **UML2 Diagramming** node under the **UML Modeling** feature.

Related Topics

Concepts on page 5

UML 2.1 Common Diagram Elements and Preferences on page 9

UML 2.1 Activity Diagram on page 10

UML 2.1 Class Diagram on page 18

UML 2.1 Component Diagram Definition on page 26

UML 2.1 Composite Structure Diagram on page 30

UML 2.1 Deployment Diagram on page 36

UML 2.1 Profile Diagram on page 40

UML 2.1 State Machine Diagram on page 43

UML 2.1 Use Case Diagram on page 47

UML 2.1 Common Diagram Elements and Preferences

The different UML 2.1 diagrams each contain palette elements in Together that are diagram-specific. However, the palette elements described in the following table are common to all the UML 2.1 diagrams.

Note: To add any palette element to your diagram several times at once, press the Control key when selecting the palette element and continue to hold it down while pasting the element into the diagram.

Common Diagram Elements

Palette Element	Description
Select	Use the **Select** node (the default) to select individual elements on the diagram.
Zoom In Zoom Out	Click the **Zoom In** node and then apply it to any element on the diagram to magnify the entire diagram. Apply the **Zoom Out** node to make the diagram smaller.
Note Text Note Attachment	Use one of these nodes to optionally show relationship conditions between diagram elements. **Note:** These elements are not UML elements.

Common Diagram Preferences

All of the UML diagrams have a set of common preferences that you can set using **Window ➤ Preferences ➤ UML <diagramtype> Diagram**.

Global settings	Specifies whether connector handles and popup bars are displayed, and whether animated layout, animated zoom, and anti-aliasing features are enabled.

Appearance	Adjusts diagram colors and fonts, and the color settings of lines, fills, and highlighting. The highlighting feature alerts you to invalid elements in your diagram and is set to red by default.
Connections	Toggles diagram line styles between rectilinear and oblique.
Icon Style	Specifies whether diagram labels are displayed in the **Classic Eclipse Style** or in the Together **Cheerful Style**.
Pathmaps	Selects path variables to use in modeling artifacts. Paths that you select are a subset of the path variables set in your workspace's **Linked Resources** preferences page.
Printing	Specifies general printing settings.
Rulers And Grid	Sets ruler options (default measurement is in inches) and the display, snap, and spacing options for grids.
View Filters	Manages required interface links by letting you specify whether Genuine links (the original links going out of a classifier) and Derived links (links going out of the port) are hidden. Diagrams are updated when this option is changed.

Related Topics

UML 2.1 Diagrams on page 8

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Activity Diagram

Activity diagrams are object-oriented flowcharts that let you model the process of an activity. To demonstrate this type of modeling, this topic uses the flow of activity involved in the purchase of an item in an E-Store system.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

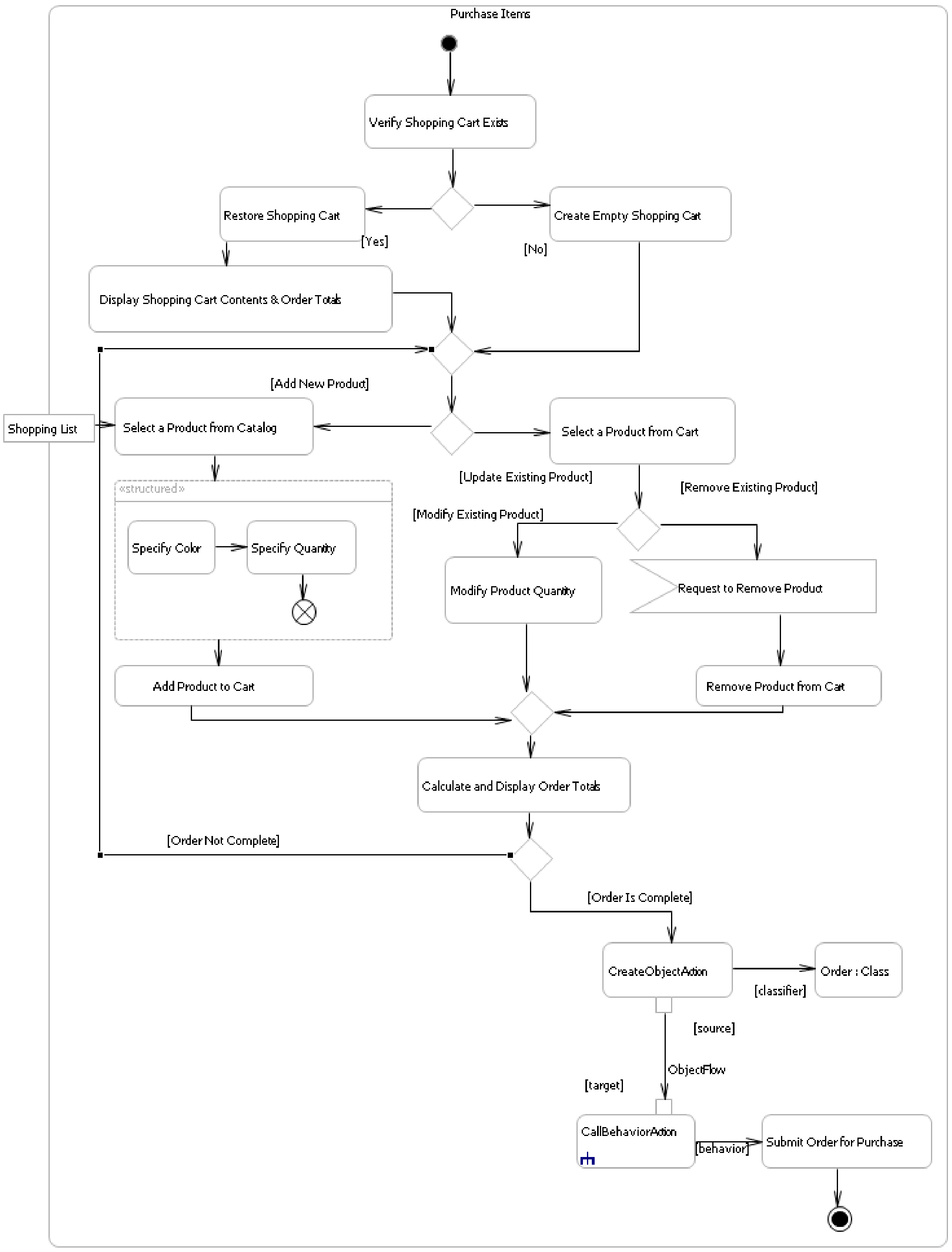
An activity diagram represents one specific aspect of a system's dynamic behavior. At a basic level, activities are made up of individual actions, which are fundamental units of executable functionality. There is control sequencing and data flow between actions, which means that the execution of one action node affects and is affected by the execution of other action nodes. Flows can have several sources and several targets.

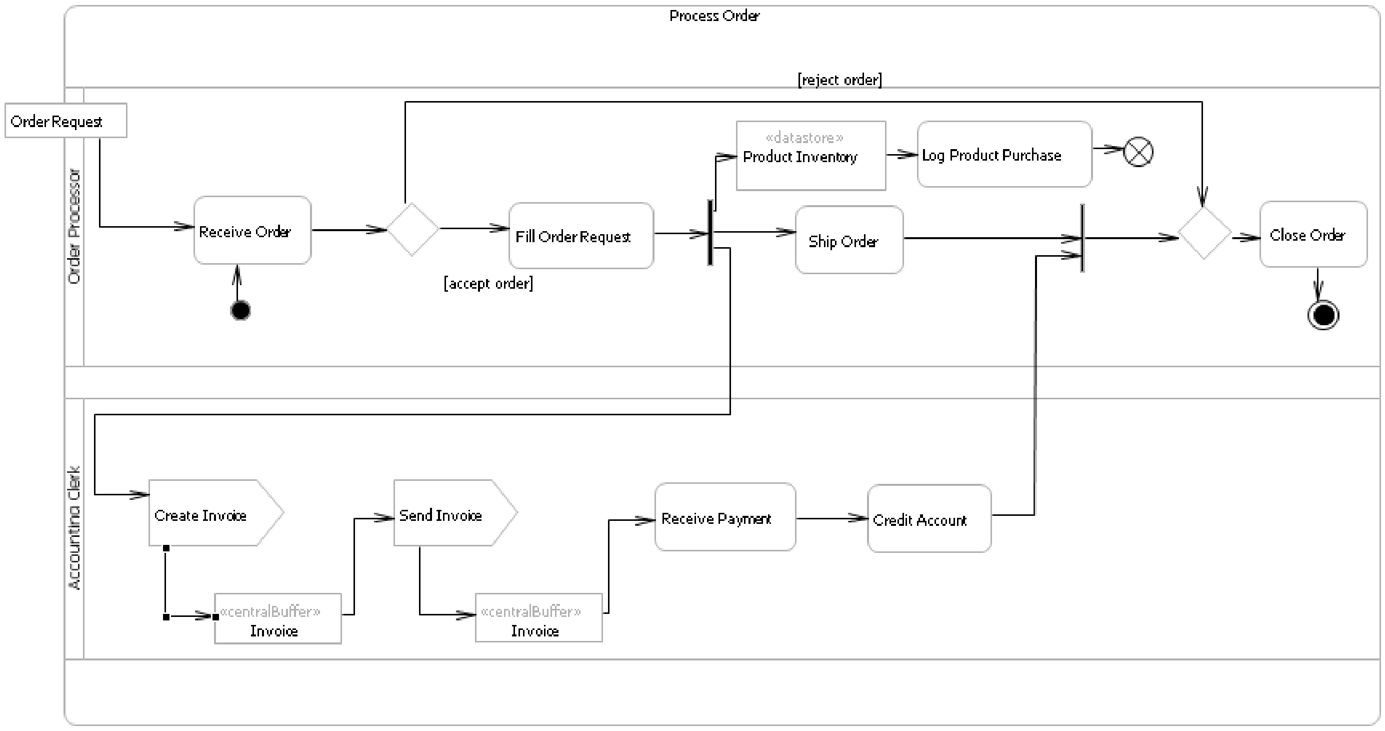
Activities can be attached to any model element in order to model the element's behavior. Activity diagrams are often used to examine complex use cases in order to capture different perspectives of them. This is possible because activity diagrams visually communicate basic and alternative flows, algorithms, extension points, loops and parallelism.

Activity diagrams focus on the sequence and conditions of system behaviors. These behaviors can be prompted by other behaviors, by objects and data becoming available, or by events occurring outside of the flow.

Sample Diagrams

The following activity diagrams show the basic purchasing activity through an E-Store system. These figures show a view of logical paths, which include both basic and alternative flows. Refactoring the purchasing activity could yield more flows (for example, specifying shipping address and rates, giftwrapping options, and discounts). Furthermore, a less in-depth perspective for this activity could also be depicted without alternative flows.


Purchase Items
Verify Shopping Cart Exists
Restore Shopping Cart
[Yes]
Create Empty Shopping Cart
[No]
Display Shopping Cart Contents & Order Totals
[Add New Product]
Shopping List
Select a Product from Catalog
Select a Product from Cart
[Update Existing Product]
[Remove Existing Product]
«structured»
Specify Color
Specify Quantity
[Modify Existing Product]
Modify Product Quantity
Request to Remove Product
Add Product to Cart
Remove Product from Cart
Calculate and Display Order Totals
[Order Not Complete]
[Order Is Complete]
CreateObjectAction
Order : Class
[classifier]
[source]
ObjectFlow
[target]
CallBehaviorAction
[behavior]
Submit Order for Purchase



The key graphical elements of an activity diagram include control flows, actions, merge/decision nodes and the activity itself. Descriptions for these and other elements used for activity diagrams can be found in the table that follows.

Palette Element	Canvas Element	Description
Activity	**Purchase Items** **Process Order**	A major task that must occur so that an operational contract can be fulfilled. Activities represent user-defined behaviors that specify how subordinate behaviors are executed via control and data flow models. Activities can be the start of an operation, a step in a business process, or an entire business process. They can originate from both inside and outside of the system.
Activity Parameter	**Shopping List** **Order Request**	Activity parameters are objects at the beginning and end of flows. Each parameter must have nodes to provide inputs to the activity and outputs from the activity.
Activity Partition	**Order Processor** **Accounting Clerk**	Activity partitions, previously called *swimlanes* in UML 1.4, group actions that have some characteristics in common often corresponding to

Palette Element	Canvas Element	Description
		organizational units in a business model. For example, they can group activities performed by the same actor or that are contained in the same flow. Although they do not affect the flow of a model, they constrain and provide a view on the behaviors of activities. **Note:** Add activity partitions to the canvas before you add individual flow elements that the partition will contain.
Accept Event Action Accept Time Event Action	**Request to Remove Product**	An action that waits for the occurrence of an event that meets a specified condition. An accept event action is displayed as a concave pentagon. When the specified condition is not yet met, an accept time event action (displayed as an hour glass) waits until the condition is met before the action can accept it. When there are no incoming edges, the action's immediate container (either the containing activity or structured node) starts after a signal is accepted.
Add Feature Value Action	N/A	Lets you add values to a feature.
Call Behavior Action	Action called to invoke the **Submit Order for Purchase** user-defined behavior	A call that starts a behavior directly instead of starting a behavior feature that subsequently starts the behavior. For asynchronous calls, the action completes immediately without a result. For synchronous calls, the call behavior action waits to execute until the invoked behavior completes, and a result is returned on its output pin.
Call Operation Action	**Select Product Attributes**	An action that requests an operation call from the target object, leading to the start of the associated behavior. Synchronous actions wait until the invoked behavior completes before executing, and then send a reply to the caller. For asynchronous actions, the invoked operation proceeds concurrently with the execution of the calling behavior.

Palette Element	Canvas Element	Description
		When the call operation action receives reply transmissions on its output pins, the action is complete.
Create Object Action	Action called to instantiate the static classifier **Order**	Creates a new object whose classifier conforms to a static classifier. The new object resides on the output pin at runtime and is returned as the value of the action.
Opaque Action	**Verify Shopping Cart Exists** **Restore Shopping Cart** **Create Empty Shopping Cart** **Display Shopping Cart Contents & Order Totals** **Select a Product from Catalog** **Specify Color** **Specify Quantity** **Select a Product from Cart** **Modify Product Quantity** **Add Product to Cart** **Remove Product from Cart** **Calculate and Display Order Totals** **Submit Order for Purchase** **Receive Order** **Log Product Purchase** **Fill Order Request** **Ship Order** **Create Invoice** **Receive Payment** **Credit Account** **Close Order**	An action whose semantics are determined by an implementation.
Send Signal Action	**Create Invoice** **Send Invoice**	Creates a signal instance that is transmitted to the target object so that an activity can execute.
OpaqueBehavior	N/A	A behavior whose semantics are determined by an implementation.

Palette Element	Canvas Element	Description
● Activity Initial Node	The starting point before the **Verify Shopping Cart Exists** action The starting point before the **Receive Order** action	The control node that starts all flows when an activity is invoked. If an activity has more than one initial node, then invoking the activity starts flows at each initial node. **Note:** Activities can also be started by an Activity Parameter node and an Accept Event Action node.
◉ Activity Final Node	The end point after the **Submit Order for Purchase** action The end point after the **Close Order** action	The final control node that stops all flows in an activity. If an activity has more than one final node, the node that is first reached stops the activity.
⊗ Flow Final Node	Element that ends the flow of product attribute selections after the **Specify Quantity** and **Log Product Purchase** actions	The final control node that stops a specific flow in an activity. It does not affect the other flows in the activity.
Merge Node Decision Node	The diamond-shaped element between the **Create Empty Shopping Cart**/**Display Shopping Cart Contents & Order Totals** actions, the **Specify Quantity and Add Product to Cart**/**Modify Product Quantity**/**Remove Product from Cart**, and the **Ship Order**/**Credit Account**/**Close Order** actions is a merge node. The other diamond-shaped elements are decision nodes.	The merge node unites multiple alternate incoming flows. Rather than synchronizing concurrent flows, it selects one flow among the alternate flows. Similarly, a decision node selects between multiple outgoing flows.
Fork Node Join Node	The vertical line between **Fill Order Request** and **Ship Order** is a Fork Node The vertical line between **Ship Order**/**Credit Account** actions and the Merge Node is a Join Node	A fork node splits one incoming flow into multiple outgoing concurrent flows. Similarly, the join node synchronizes multiple incoming flows into one outgoing flow. These nodes enable parallelism in activities.
Conditional Node	N/A	This structured activity node makes an exclusive choice between alternatives. These nodes are made up of clauses with a test section and body section. The test sections execute at the same time the conditional node does. No matter how many test sections yield a true value, only one body section will be executed. If no test section yields a

Palette Element	Canvas Element	Description
		true value, no body sections are executed.
Expansion Region	N/A	A structured, nested region of an activity that executes once for every element in the input collection. It has ExpansionNodes as its inputs and outputs. An expansion region begins executing after it accepts a token from an activity. Expansion regions let you apply behaviors to elements of a set without constraining the order of application. Expansion regions are displayed as dashed, rounded rectangles that enclose either the `parallel`, `iterative`, or `streaming` keyword. Vertically divided rectangles on its border represent its expansion nodes.
Loop Node	N/A	A structured, nested subregion of an activity that acts as a loop and provides structured iteration. Each loop node has a setup, test and body section. The setup section is executed upon entry to the loop, and the test and body sections execute (unordered) continually until the test returns False.
Structured Activity Node	Product Attributes node where color and quantity are specified	An activity node that can expand into an Activity Group via subordinate, well-nested nodes. Although nesting is allowed, none of the subnodes and edges can be contained by another structured node. Structured activity nodes are displayed as dashed, rounded rectangles that enclose nodes and edges and the keyword `<<structured>>`.
Central Buffer	**Invoice**	An object node that manages flows from multiple sources and destinations. Unlike other buffers, central buffers are not attached to actions or activities. They help manage queuing and competing object flows.

Palette Element	Canvas Element	Description
Datastore	**Product Inventory**	A central buffer node that stores all incoming tokens and distributes select copies for movement downstream.
Pin Input Pin Output Pin	The source output pin from the **CreateObjectAction** The target input pin to the **CallBehaviorAction**	An object node that permits inputs to and outputs from actions. When the output pin of one action is connected to the input pin of the same name in another action, the joint flow can be represented in a standalone pin.
Selection	N/A	Creates a selection link between elements.
Control Flow	All arrows between Opaque Actions	An edge at which one activity node starts after the previous activity node finishes. Control flows model behavior sequencing that does not involve object flow.
Object Flow	All arrows between objects and actions	An edge that passes objects and data between object nodes in an activity.
Exception Handler Link	N/A	Specifies which body to execute after a protected node executes with an exception.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Class Diagram

Class diagrams provide an overview of a target system according to the objects and classes specified in the system and the relationships between them. To demonstrate this type of diagram, this topic models subsystem interfaces of an E-Store system.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

Every high-level architecture identifies key subsystems and their interfaces. A subsystem design model specifies how interfaces of a given subsystem are to be implemented. The key element of a subsystem design model is one or more class diagrams that show how classes constitute the subsystem. In addition, class diagrams show the relationships between classes and interfaces, and the relationships among classes.

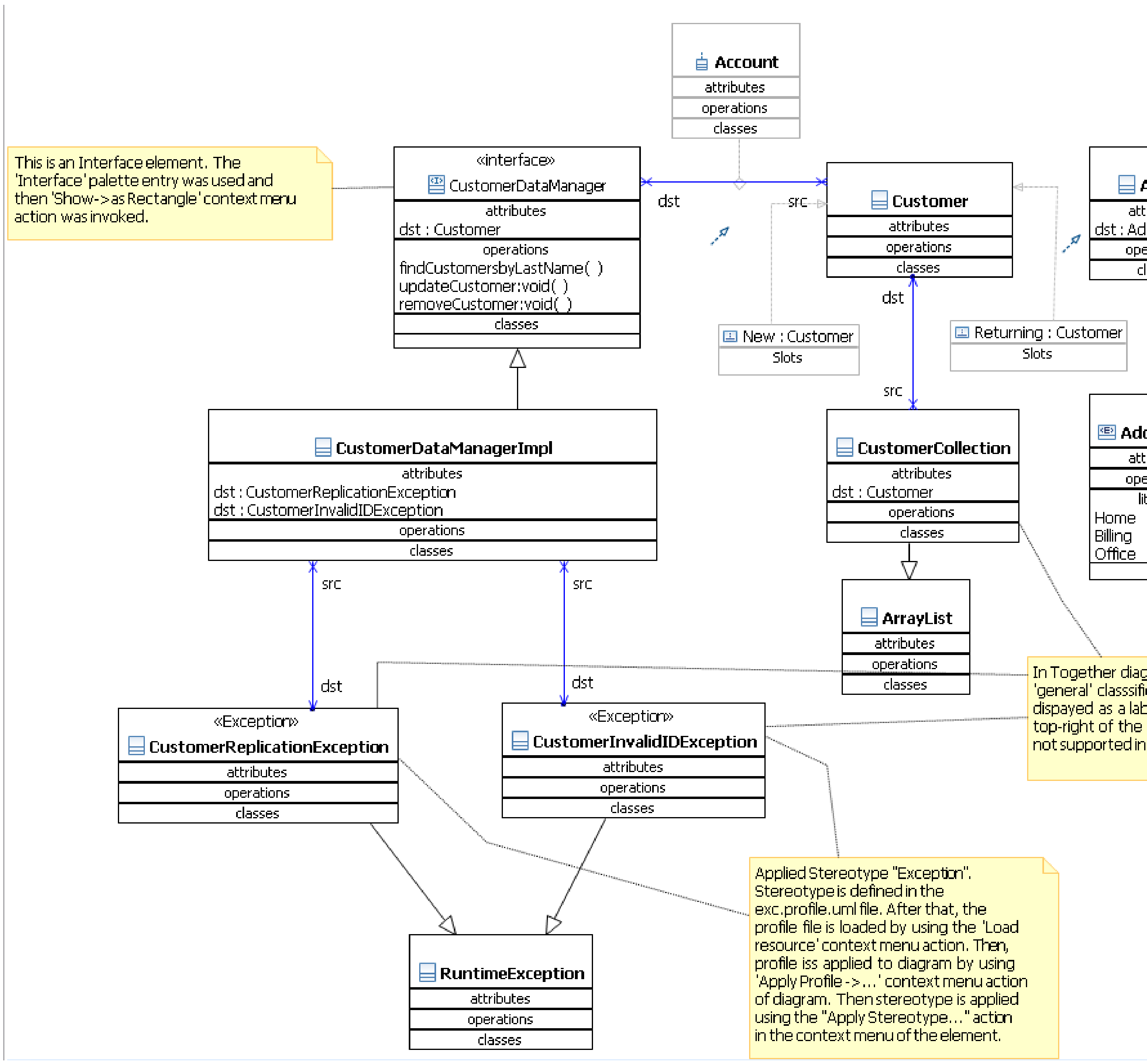
Class diagrams are static diagrams in that they display what interacts but not what happens during an interaction. Nor do they have to incorporate advanced elements or be extensively elaborate in order to be useful. Initially at least, the focus of every class diagram should be its classes and their relationships. High-level class diagrams are created in an analysis phase and attempt to depict the basics of a system. When an increased understanding of the system prompts an attempt to diagram solutions, attributes and operations can be added to classifiers.

Object diagrams are class diagrams that show specific instances of classes and their relationships. Relationships describe the different types of logical connections in class and object diagrams. They include general relationships (dependency, multiplicity), relationships between instances (link, composition, association, aggregation), and relationships between classes (generalization, inheritance, realization).

Using Together, you can create language-neutral class diagrams in design projects, or language-specific class diagrams in implementation projects. For implementation projects, all diagram elements are immediately synchronized with the source code.

Sample Diagrams

The following diagram specifies the operations and attributes of an E-Store's subsystem interfaces. The first diagram elaborates on a customer database subsystem interface.


Account
attributes
operations
classes
This is an Interface element. The 'Interface' palette entry was used and then 'Show->as Rectangle' context menu action was invoked.
«interface»
CustomerDataManager
attributes
dst : Customer
operations
findCustomersbyLastName( )
updateCustomer:void( )
removeCustomer:void( )
classes
dst
src
Customer
attributes
operations
classes
dst
New : Customer
Slots
Returning : Customer
Slots
src
CustomerDataManagerImpl
attributes
dst : CustomerReplicationException
dst : CustomerInvalidIDException
operations
classes
CustomerCollection
attributes
dst : Customer
operations
classes
ArrayList
attributes
operations
classes
Home
Billing
Office
src
src
dst
dst
«Exception»
CustomerReplicationException
attributes
operations
classes
«Exception»
CustomerInvalidIDException
attributes
operations
classes
In Together diag
'general' classifi
dispayed as a lab
top-right of the
not supported in
Applied Stereotype "Exception". Stereotype is defined in the exc.profile.uml file. After that, the profile file is loaded by using the 'Load resource' context menu action. Then, profile iss applied to diagram by using 'Apply Profile ->...' context menu action of diagram. Then stereotype is applied using the "Apply Stereotype..." action in the context menu of the element.
RuntimeException
attributes
operations
classes

The fundamental elements of most class diagrams consist of classifiers with attributes and operations, and connectors that specify relationships between classes. Descriptions for these and other elements used for class diagrams can be found in the table that follows.

Palette Element	Canvas Element	Description
Class	**Customer** **Address** **CustomerDataManagerImpl** **RuntimeException**	A classifier whose behavior is described through the interaction of its parts. Within a class you can specify attributes, operations, and other classes.

Palette Element	Canvas Element	Description
	ArrayList **CustomerCollection**	In the Class diagram editor, classes can only be displayed as classifier rectangles.
Package	N/A	A package groups elements of the diagram and provides a namespace for those grouped elements. The Package node can be used to represent a *Classifier*, and can import individual members or all members of other packages. Packages can also be displayed as frames.
Enumeration	**AddressType**	A data type with values enumerated in the model as user-defined literals.
DataType	N/A	A classifier that is very similar to a class except that its instances are identified only by their value.
PrimitiveType	N/A	Defines a predefined data type with substructure parts defined outside of UML. Instances of primitive types are data values without identities. They include Boolean, Integer, UnlimitedNatural, and String. PrimitiveTypes have a many-to-one ratio to mathematical elements (such as integers) defined outside of UML.
Constraint	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	A constraint is a Boolean expression that restricts the extension of an element. It restricts by imposing a value that specifies additional semantics beyond what is imposed by other language constructs applied to that element. The element that owns the constraint must have access to Constrained Elements. By default, the value of a constraint is an Opaque Expression element.
Association Class	**Account**	An element with both association and class properties that connects a set of classifiers. It has its own set of features that do not belong to any of the connected classifiers.

Palette Element	Canvas Element	Description
		Association classes are displayed as classes attached to association paths by a dashed line.
Interface	**CustomerDataManager**	Creates a new object whose classifier conforms to a static classifier. The new object resides on the output pin at runtime and is returned as the value of the action. An interface must have at least one class to implement it.
Attribute	**dst : Customer** **dst : AddressType** **dst : CustomerReplicationException** **dst : CustomerInvalidIDException**	An entity that defines a model element's properties with specific values. For example, attributes of a Nation class are represented by property instances owned by the class, such as geography, population, and government.

Operation	**findCustomersbyLastName()**
	updateCustomer:void()
	removeCustomer:void()
		Specifies the behavioral characteristics of a classifier, including the behavior's name, type, parameters, and constraints for invoking the behavior.

Enum Literal	**Home** **Billing** **Office**	A user-defined data type value for an enumeration.
Port	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	Each of the small squares attached to classes that connect the behavior of classes with their internal parts and with the other parts of the system. Ports can specify which service a class provides to its environment and which service a class expects from its environment. Ports are labeled by their different types and are either public (visible to the environment, straddling class boundary), private (only visible to the namespace that owns it, within class boundary), or protected (only visible to elements that have a generalization relationship to the namespace that owns it). If a port's isBehavior property is true, a small state symbol is displayed beside it.
Template Signature	N/A	Bundles the set of formal template parameters for a template element. The

Palette Element	Canvas Element	Description
		signature is owned by the template element and has parameters that define the signature for binding this template.
Element Import	N/A	Identifies an element in another package that can be referenced using its name without a qualifier. The **ElementImport** compartment is collapsed by default.
Association	The blue arrows between classifiers.	The interaction between model elements, represented by a solid line between them. These interaction links can be either unidirectional (indicating that an actor initiates the interaction) or bidirectional (indicating that an actor can participate in the interaction without initiating it).
Shared Aggregation Composite Aggregation Navigable Association	N/A	A nonbinary aggregation that represents a shared relationship between a part and its composites. Shared aggregations have a hollow diamond notation at the end of its aggregation line. A binary aggregation that represents a whole/part relationship between a part and its composites. This form of aggregation requires that a part instance be included in no more than one composite at a time. Composite aggregations have a filled-in diamond notation at the aggregate end of its association line. An association is navigable in the direction of its arrow. The marked association end is owned by the classifier, and the unmarked end is owned by the association. If there are no arrows, the association is navigable in both directions, and each association end is owned by the classifier at the opposite end.
Dependency Abstraction Usage	N/A	Elements whose semantics depend on the definition of a supplier element are in a dependency relationship with the supplier element. Dependency links are shown as dashed arrows between the

Palette Element	Canvas Element	Description
Substitution		two model elements with the arrowhead pointing to the supplier element. An abstraction relationship relates two classifiers by the same concept but at different viewpoints or levels of abstraction. A usage relationship involves one classifier that depends on another classifier for its operation and implementation. A substitution relationship implies that the instances of the substituting classifier can substitute instances of the contract classifier at runtime.
Generalization	The hollow arrows between classifiers.	If one classifier inherits all the behavior of another classifier and furthermore extends it with additional behavior, a generalization link results. The arrow points to the more general of the two.
Provided Interface	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	A provided interface represents services that are offered by instances of a classifier to fulfill contractual obligations.
Required Interface	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	A required interface specifies a usage dependency (which include the services needed to perform a required function) between instances of a classifier and their interfaces.
Constrained Element	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	Those elements of the diagram that are required to evaluate the conditions specified in a constraint. These references are displayed as links on the diagram.
NAry Dependency Target	N/A	A N-ary association is used for representing two or more aggregations to the same aggregate.
NAry Dependency Source	N/A	A N-ary association is used for representing two or more aggregations to the same aggregate.
Association End	N/A	Connects the line depicting an Association Class and the icon depicting the connected classifier. The

Palette Element	Canvas Element	Description
		Association End defines the ends of the Association Class. Names of Association Ends are optional and can be suppressed.
Realization	N/A	An abstraction relationship between a supplier set of elements (specification) and a client set of elements (implementation). The client element realizes the behavior that the supplier element specifies. Realizations model relationships such as transformations, optimizations, and stepwise refinement.
Template Binding	N/A	A relationship between an element and a template that specifies the substitution of actual parameters for the formal parameters of the template.
Instance Specification	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	Specifies the existence of an entity in a modeled system. The entity can be a class, in which case the instance specification describes an object of that class. For example, an instance specification of the class Nation might be Brazil. An entity can also be an association, in which case the instance specification describes the link of the association. Instance specifications can optionally have names, displayed in the format *instancename*:*classifiername*.
Slot	See the "*UML 2.1 Composite Structure Diagram* on page 30" topic.	Specifies the value of a classifier's defining structural feature owned by the slot's instance specification. It represents how an entity modeled by the slot's instance specification has a structural feature with specific values.
Literal String Literal Integer Expression	N/A	A Literal String is a sequence of characters within double quotes that contain a string-valued attribute. A Literal Integer is a sequence of digits that signify an Integer-valued attribute. An Expression, such as *else* or *plus*, represents a node in an expression tree and defines a symbol.

Palette Element	Canvas Element	Description
		Note: These three elements are supported for expanded Instance Specifications only.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

UML 2.1 Composite Structure Diagram on page 30

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Component Diagram Definition

Component diagrams depict the high-level components of a system and their structural relationships and interfaces.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

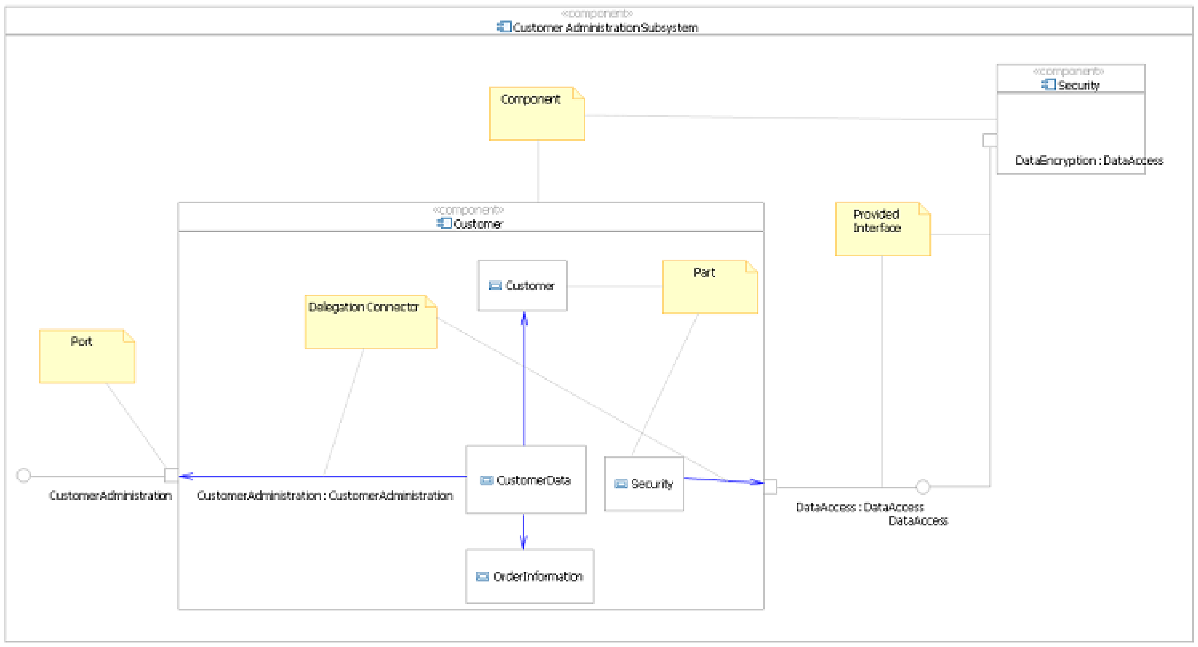
Component diagrams depict the high-level components of a system and their structural relationships and interfaces. A component diagram provides a high-level architectural view of a system.

There is no InstanceSpecification element in the UML 2.1 Component diagram editor. Sometimes the UML specification references Parts of a Component as Instances.

Component diagrams are usually used to model the logical components of a system. Physical architecture issues are better addressed in a Deployment diagram.

Sample Diagram

The following figure is an example of a component diagram. This diagram illustrates a customer administration subsystem.



Component diagrams consist of components, their parts, ports, provided interfaces, required interfaces, and other elements. These elements are described in the table that follows.

Palette Element	Canvas Element	Description
Component	**Customer Administration Subsystem** **Security** **Customer**	A component is a specialized version of a class. A component element uses the same notation rules as a class element.
Artifact		Physical artifacts of a system are usually described in Deployment Diagrams. However, if a component is closely connected with its physical store, you can add and link an artifact in a Component Diagram.
Interface		A type of contract that declares a set of coherent features and conditions. Any classifier that interacts with the interface must fulfill this contract. The type of dependency a classifier has with its interface is distinguished by a circle at the end of a solid line (indicating a provided interface) or a half-circle at the end of a solid line (indicating a required interface).
Class		A classifier whose behavior is described through the interaction of its parts. Within a class you can specify

Palette Element	Canvas Element	Description
		attributes, operations, and other classes.
Part	**CustomerData**, **Security**	A part represents a set of one or more instances which are owned by a containing classifier instance. If an instance owns a set of graphical elements, then the graphical elements could be represented as parts in order to model a relationship between the instance and the graphical elements. You can remove a part from its parent before deleting the parent so that the part is not deleted when the parent is deleted.
Package		A UML construct that allows you to organize model elements into groups. A package element in a diagram displays the package contents. Packages are depicted as file folders and can be used in any UML diagram.
Provided Interface		A provided interface represents services that are offered by instances of a classifier to fulfill contractual obligations. A Provided Interface link can be created between a Classifier and an Interface ('genuine' link) or between a Component or Port to an Interface ('derived' link). Visibility of the link is managed in the **View Filters** preferences page (**Window** ➤ **Preferences** ➤ **Modeling** ➤ **View Management** ➤ **Show/Hide Elements**).
Required Interface		A required interface specifies a usage dependency (which include the services needed to perform a required function) between instances of a classifier and their interfaces. Ports often have Required Interface links. A Required Interface can be created for a Port only if the Port"s **type** reference property is not null.
Delegation Connector		A link that lets two or more instances communicate with each other. Delegation connectors link the external

Palette Element	Canvas Element	Description
		contracts of components (specified by their ports) to the internal realization of that behavior by the component's parts.
Dependency		Elements whose semantics depend on the definition of a supplier element are in a dependency relationship with the supplier element. Dependency links are shown as dashed arrows between the two model elements with the arrowhead pointing to the supplier element.
Assembly Connector		A link that lets two or more instances communicate with each other. Assembly connectors link required interfaces or ports to provided interfaces or ports and specify that one component provides the required services of another component.
Element Import		Identifies an element in another package that can be referenced using its name without a qualifier. A secondary package node located in the upper-left corner of the above diagram contains element imports. The **ElementImport** compartment is collapsed by default.
Attribute		An entity that defines a model element's properties with specific values. For example, attributes of a Nation class are represented by property instances owned by the class, such as geography, population, and government.
Operation		Specifies the behavioral characteristics of a classifier, including the behavior's name, type, parameters, and constraints for invoking the behavior. You can edit operation parameters by selecting the **Manage Parameters** context menu action within the operation.
Port	**DataEncryption**	Each of the small squares attached to classes that connect the behavior of classes with their internal parts and with the other parts of the system. Ports can

Palette Element	Canvas Element	Description
		specify which service a class provides to its environment and which service a class expects from its environment. Ports are labeled by their different types and are either public (visible to the environment, straddling class boundary), private (only visible to the namespace that owns it, within class boundary), or protected (only visible to elements that have a generalization relationship to the namespace that owns it). If a port's isBehavior property is true, a small state symbol is displayed beside it.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Composite Structure Diagram

Composite structure diagrams depict a composition of interconnected elements (including parts and connectors) within a class and any potential collaborations the structure allows. When a composite structure is executed, its interconnected elements collaborate to achieve a purpose.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

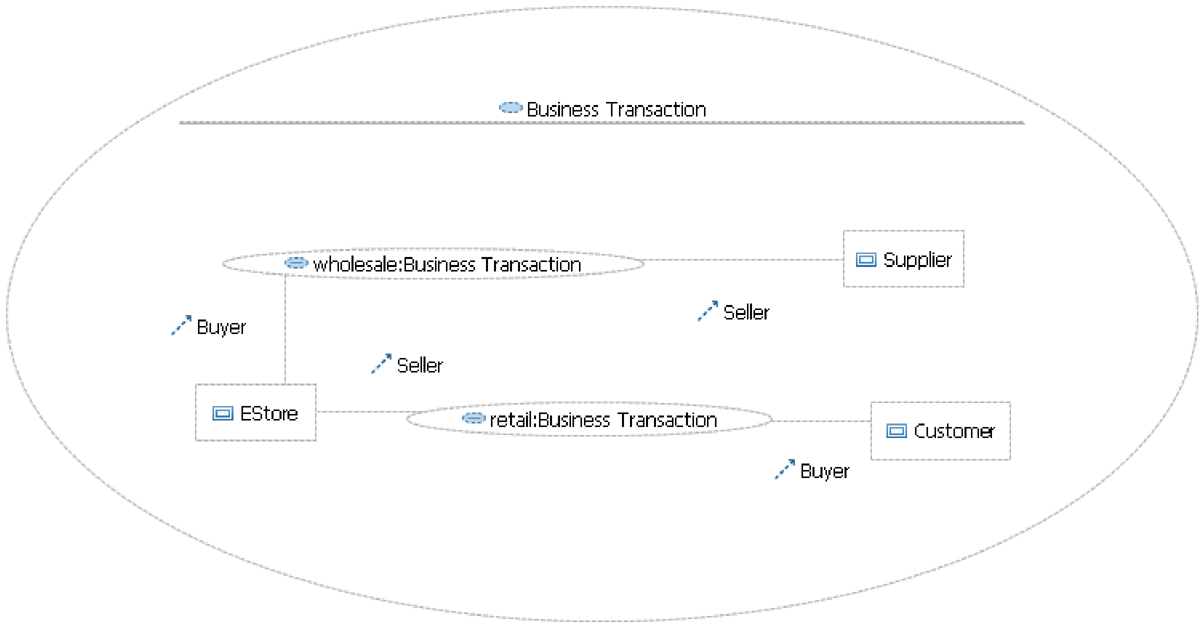
Definition

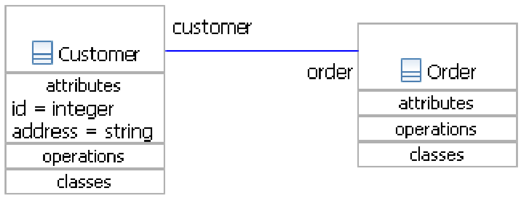
Composite structure diagrams depict the internal structure of a classifier, including its interaction points to the other parts of the system. These diagrams show the configuration of parts that jointly perform the behavior of the containing classifier.

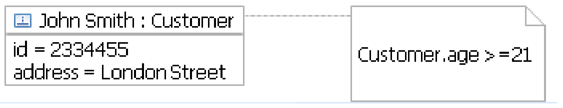
Some of the contained parts can be included by reference. Referenced parts are represented by dotted rectangles. Classifiers can be isolated from and interact with their environments through ports. Different interactions are distinguished by different ports.

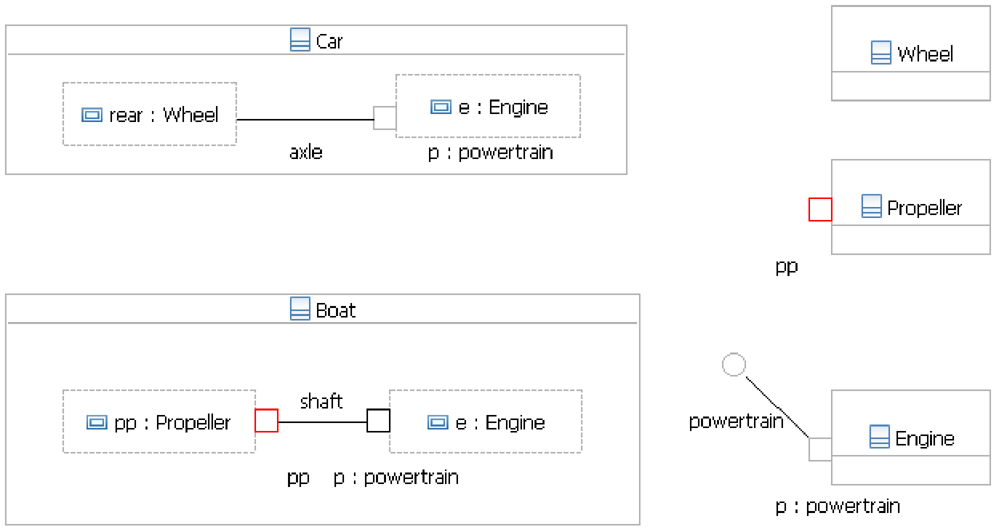
Sample Diagram

The following figures contain examples of composite structure diagrams. The first one illustrates a collaboration that shows how merchandise is moved in an E-Store system. The next diagram shows a John Smith instance specification of the Customer class and various slots that define the instance specifications. The last diagram, courtesy of the *Unified Modeling Language (OMG UML): Superstructure, Version 2.1.2 November 2007*, p. 183, shows how an Engine class is used differently by both a Car class and a Boat class.









Composite structures are made up of several subpackages including ports, collaborations, and classes. These subpackages and other elements included in composite structure diagrams are described in the table that follows.

Palette Element	Canvas Element	Description
Collaboration	**Business Transaction**	Collaborating elements that perform specialized tasks and are structured collectively to accomplish a function. Collaborations show how a collection of cooperating classes achieve something.
Class	**Customer** **Order** **Card** **Wheel** **Propeller** **Car** **Boat** **Engine**	A classifier whose behavior is described through the interaction of its parts. Within a class you can specify attributes, operations, and other classes. Classes can be changed into an expanded notation when its attributes are displayed as rectangles.
Interface	**Powertrain** **Power**	A type of contract that declares a set of coherent features and conditions. Any classifier that interacts with the interface must fulfill this contract. The type of dependency a classifier has with its interface is distinguished by a circle at the end of a solid line

Palette Element	Canvas Element	Description
		(indicating a provided interface) or a half-circle at the end of a solid line (indicating a required interface).
Instance Specification	**John Smith:Customer**	Specifies the existence of an entity in a modeled system. The entity can be a class, in which case the instance specification describes an object of that class. For example, an instance specification of the class Nation might be Brazil. An entity can also be an association, in which case the instance specification describes the link of the association. Instance specifications can optionally have names, displayed in the format *instancename*:*classifiername*.
Constraint	**Customer.age >=21**	A constraint is a Boolean expression that restricts the extension of an element. It restricts by imposing a value that specifies additional semantics beyond what is imposed by other language constructs applied to that element. The element that owns the constraint must have access to Constrained Elements. By default, the value of a constraint is an Opaque Expression element.
Element Import	N/A	Identifies an element in another package that can be referenced using its name without a qualifier. A secondary package node located in the upper-left corner of the above diagram contains element imports. The **ElementImport** compartment is collapsed by default.
Attribute	**id = integer**, **address = string** of **Customer** class	An entity that defines a model element's properties with specific values. For example, attributes of a Nation class are represented by property instances owned by the class, such as geography, population, and government.
Operation	See the "*UML 2.1 Class Diagram* on page 18" topic.	Specifies the behavioral characteristics of a classifier, including the behavior's

Palette Element	Canvas Element	Description
		name, type, parameters, and constraints for invoking the behavior. You can edit operation parameters by selecting the **Manage Parameters** context menu action within the operation.
Port	**p:powertrain** **pp**	Each of the small squares attached to classes that connect the behavior of classes with their internal parts and with the other parts of the system. Ports can specify which service a class provides to its environment and which service a class expects from its environment. Ports are labeled by their different types and are either public (visible to the environment, straddling class boundary), private (only visible to the namespace that owns it, within class boundary), or protected (only visible to elements that have a generalization relationship to the namespace that owns it). If a port's isBehavior property is true, a small state symbol is displayed beside it.
Collaboration Use	**wholesale:Business Transaction** **retail:Business Transaction**	Describes how a collaboration pattern is applied according to its context. It depicts a specific use of a collaboration that explains relationships between classifier properties.
Slot	**id = 2334455** **address = London Street**	Specifies the value of a classifier's defining structural feature owned by the slot's instance specification. It represents how an entity modeled by the slot's instance specification has a structural feature with specific values.
Connector	**axle** **shaft**	A link that lets two or more instances communicate with each other. A connector can either be a delegation connector or an assembly connector. Delegation connectors link the external contracts of components (specified by their ports) to the internal realization of that behavior by the component's parts. Assembly connectors link required interfaces or ports to provided interfaces

Palette Element	Canvas Element	Description
		or ports and specify that one component provides the required services of another component.
Role Binding	The links in the Collaboration diagram	A dependency that maps between features of collaboration types and features of a classifier or operation. The mapping determines which connectable element of the classifier or operation plays which role in the collaboration.
Provided Interface	Dependency to **Powertrain** interface	A provided interface represents services that are offered by instances of a classifier to fulfill contractual obligations.
Required Interface	Dependency to **Power** interface	A required interface specifies a usage dependency (which include the services needed to perform a required function) between instances of a classifier and their interfaces.
Association	The blue link between the **Customer** and **Order** classes.	The interaction between model elements, represented by a solid line between them. These interaction links can be either unidirectional (indicating that an actor initiates the interaction) or bidirectional (indicating that an actor can participate in the interaction without initiating it).
Shared Association	N/A	A nonbinary association that represents a shared relationship between a part and its composites. Shared associations have a hollow diamond notation at the end of its association line.
Composite Association	N/A	A binary association that represents a whole/part relationship between a part and its composites. This form of aggregation requires that a part instance be included in no more than one composite at a time. Composite associations have a filled-in diamond notation at the aggregate end of its association line.
Navigable Association	N/A	An association is navigable in the direction of its arrow. The marked association end is owned by the

Palette Element	Canvas Element	Description
		classifier, and the unmarked end is owned by the association. If there are no arrows, the association is navigable in both directions, and each association end is owned by the classifier at the opposite end.
Constrained Element	Link between **Customer.age >=21** and **John Smith:Customer**	Those elements of the diagram that are required to evaluate the conditions specified in a constraint. These references are displayed as links on the diagram.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Deployment Diagram

Deployment diagrams model hardware functionality in systems, including the components deployed on the hardware and the associations between those components. To demonstrate this type of modeling, this topic uses an E-Store deployment model.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

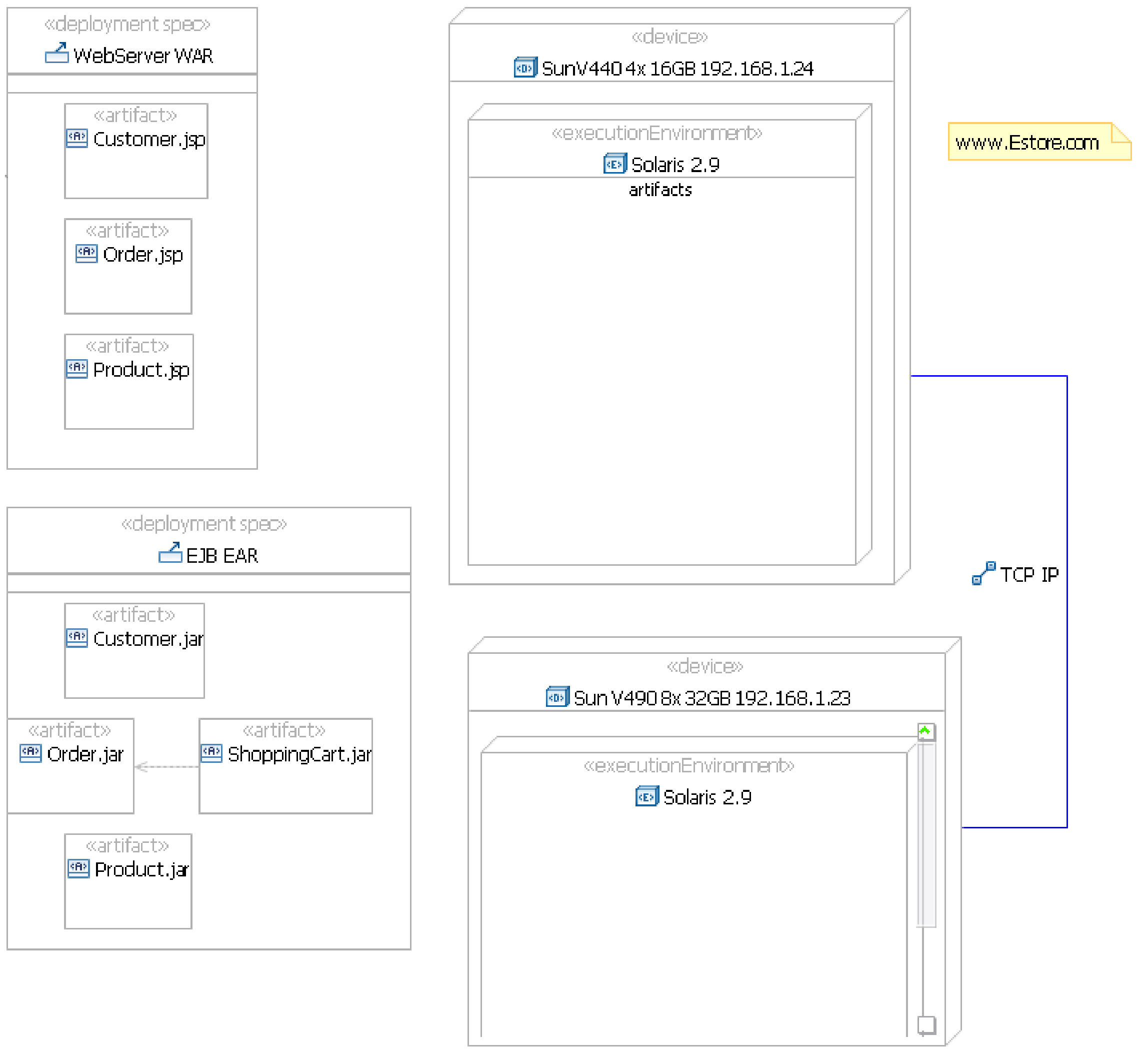
Definition

Through deployment diagrams, a set of constructs specifies the physical architecture of a software system and how it gets executed. These diagrams also define how the software is mapped to physical nodes. Nodes are computational resources upon which software artifacts can be deployed for execution. In such a system, software artifacts are assigned to nodes instead of components, and these artifacts implement collections of components. Artifacts can include source files, executable files, documents, programs, libraries or data bases that are constructed or modified in a project.

Deployment diagrams typically model hardware configurations in conjunction with their software components. They show how artifacts are allocated to nodes by diagraming the deployments defined between them. By modeling object-oriented systems through deployment diagrams, you can view runtime configurations in a static view and visualize the distribution of components in an application.

Sample Diagram

The following figure contains a diagram that shows an example of how the architecture of an E-Store system is executed.



The key elements used in a deployment diagram are nodes, components, and associations. There are two standard stereotypes for nodes: Devices represent physical devices such as a server machine or a workstation, and Environments represent software execution engines such as an Apache Web server, a Tomcat Web container or an Internet Explorer Web browser.

Nodes can be nested within nodes. For example, a node representing an execution environment can be nested in a node representing a device. An association between nodes represents a communication channel over which information is passed. Associations can be stereotyped to indicate a type of a communication protocol or a connection between nodes.

Descriptions for these concepts and the other available items on the diagram palette can be found in the following table.

Palette Element	Canvas Element	Description
Artifact	**Customer.j*** **Order.j*** **Product.j*** **ShoppingCart.jar**	Specifies a physical piece of information (such as a script, file, or mail message) created by a software process or deployment of a system. Artifacts are classifiers that can have a list of identifying properties and operations. They can have composition associations to other artifacts nested within them and can also be extended to become the source of deployment to a node.
Device	**SunV440 4x 16GB 192.168.1.24** **Sun V490 8x 32GB 192.168.1.23**	A physical system resource where artifacts are deployed for execution. Devices can be nested elements where physical machines are decomposed into their elements.
Node	The devices in the diagram are types of nodes.	Represents a computational resource upon which artifacts are deployed for execution. Nodes contain a set of elements that are also deployed on them. Nodes are connected in a network through communication paths, such as servers and workstations.
Environment	**Solaris 2.9** **Apache 4.0** **Tomcat Servlet Engine** **JBoss 5.0**	Creates an execution environment that lets nodes execute the components it owns. Nodes represent the physical hardware environment on which the execution environment resides. Composite associations assign execution environments to their node instances.
Specification	**WebServer WAR** **EJB EAR**	Subtypes of artifacts that define a set of deployment properties to determine how parameters of an artifact deployed on a node are executed.
Manifestation	The association between the **WebServer WAR** deployment specification and the **Tomcat Servlet Engine** execution environment.	Represents the physical expression of one or more model elements by the artifact that owns the elements.

Palette Element	Canvas Element	Description
	The association between the **EJB EAR** deployment specification and the **JBoss 5.0** execution environment.	
Deployment	N/A	Indicates that an artifact or artifact instance, such as an executable or configuration file, has been deployed to a deployment target.
Specification Link	N/A	Links a deployment specification node to a deployment edge. This moves the deployment specification to the selected deployment and updates the deployment property of the deployment specification.
Communication Path	**TCP IP**	An association that lets two deployment targets exchange signals and messages.
Dependency	Link between **ShoppingCart.jar** and **Order.jar**	Elements whose semantics depend on the definition of a supplier element are in a dependency relationship with the supplier element. Dependency links are shown as dashed arrows between the two model elements with the arrowhead pointing to the supplier element.
Element Import	N/A	Identifies an element in another package that can be referenced using its name without a qualifier. A secondary package node located in the upper-left corner of diagram views (not shown above) contains element imports. The **ElementImport** compartment is collapsed by default.
Property	N/A	Extends a deployment target so that the deployment can be modeled to hierarchical nodes that have properties functioning as parts in the internal structure of an encompassing node.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Profile Diagram

A domain metamodel sometimes contains some elements that cannot be expressed directly using UML constructs. To completely express them, you can create a profile definition that lets you define all the domain model's non-UML properties. Profiles provide mechanisms, such as stereotypes and constraints, that allow metaclasses from existing metamodels to be extended so they can be adapted for different purposes. A profile is a collection of such extensions that collectively customize UML for a particular domain.

A UML 2.1 profile diagram is a tool-specific diagram used for creating UML Profiles. This type of diagram is typically used by a toolsmith rather than a practitioner.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

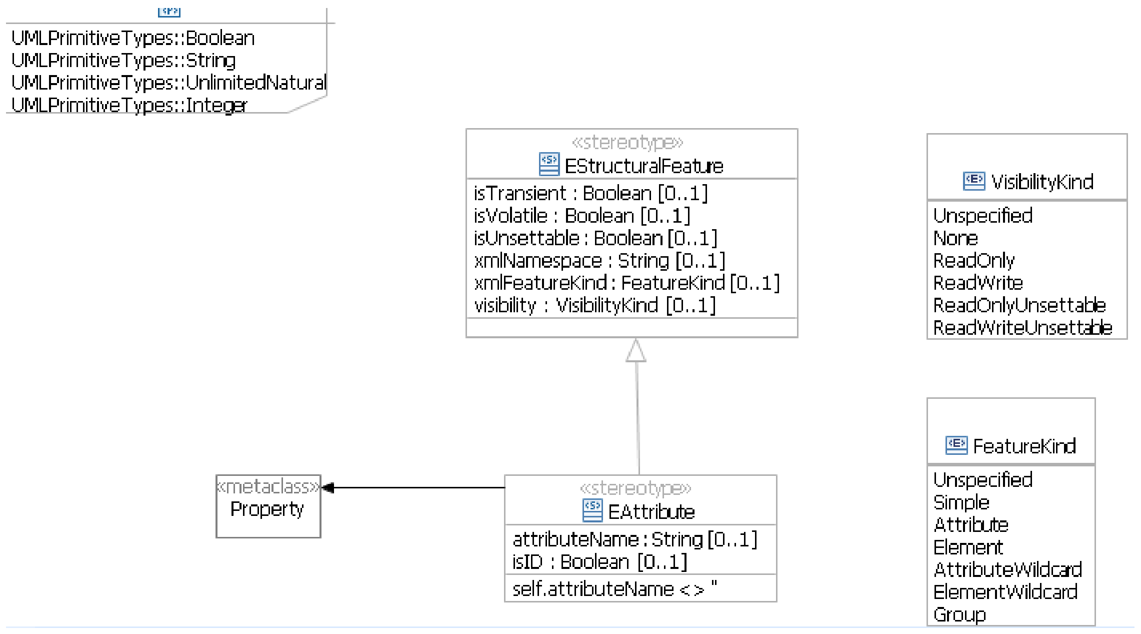
A profile must always be used in conjunction with its reference metamodel.

Stereotypes enable the use of platform- or domain-specific terminology or notation to extend the metaclass beyond its usual avenues of extension. Unlike other metaclasses, stereotypes must always be used in conjunction with another metaclass that it extends. Extensions associate stereotypes to metaclasses, which allows the stereotype to be applied to any selected element. Stereotypes can be used to add Keywords, Constraints, Images, and Properties.

Constraints restrict profiles based on the metaclasses defined in the package.

Sample Diagram

The following figure contains a diagram that shows how a profile can be extended using various diagram palette elements.


UMLPrimitiveTypes::Boolean
UMLPrimitiveTypes::String
UMLPrimitiveTypes::UnlimitedNatural
UMLPrimitiveTypes::Integer
«stereotype»
EStructuralFeature
isTransient : Boolean [0..1]
isVolatile : Boolean [0..1]
isUnsettable : Boolean [0..1]
xmlNamespace : String [0..1]
xmlFeatureKind : FeatureKind [0..1]
visibility : VisibilityKind [0..1]
VisibilityKind
Unspecified
None
ReadOnly
ReadWrite
ReadOnlyUnsettable
ReadWriteUnsettable
FeatureKind
Unspecified
Simple
Attribute
Element
AttributeWildcard
ElementWildcard
Group
«metaclass»
Property
«stereotype»
EAttribute
attributeName : String [0..1]
isID : Boolean [0..1]
self.attributeName < > ''

The key elements used in a profile diagram are profiles, extensions, metaclasses, and stereotypes. Descriptions for these concepts and the other available items on the diagram palette can be found in the following table.

Palette Element	Canvas Element	Description
Profile	**<<profile>>**	A type of Package that defines how a reference metamodel can be extended for the purposes of adapting the metamodel to a specific platform or domain.
Stereotype	**EStructuralFeature** **EAttribute**	A class element that specifies how an existing metaclass gets extended.
Metaclass	**Property**	A class that has instances that are classes. A metaclass defines the behavior of other classes and their instances. A metaclass that has no imported element appears in red highlighting.
Extension	Solid arrow between **EAttribute** enumeration and **Property** metaclass.	Creates an execution environment that lets nodes execute the components it owns. Nodes represent the physical hardware environment on which the execution environment resides.
Enumeration	**VisibilityKind** **FeatureKind**	A data type with values enumerated in the model as user-defined literals.

Palette Element	Canvas Element	Description
Generalization	The hollow arrow between stereotypes.	If one classifier inherits all the behavior of another classifier and furthermore extends it with additional behavior, a generalization link results. The arrow points to the more general of the two.
Property	N/A	Extends a deployment target so that the deployment can be modeled to hierarchical nodes that have properties functioning as parts in the internal structure of an encompassing node.
Constraint	**self.attributeName <> ''** of the **EAttribute** stereotype	A constraint is a Boolean expression that restricts the extension of an element. It restricts by imposing a value that specifies additional semantics beyond what is imposed by other language constructs applied to that element. The element that owns the constraint must have access to Constrained Elements. By default, the value of a constraint is an Opaque Expression element.
Literal	See the "*UML 2.1 Class Diagram* on page 18" topic.	A user-defined data type value for an enumeration.
Element Import	N/A	Identifies an element in another package that can be referenced using its name without a qualifier. A secondary package node located in the upper-left corner of diagram views (not shown above) contains element imports. The **ElementImport** compartment is collapsed by default.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

A Typical Scenario of Creating a Profile on page 58

UML 2.1 State Machine Diagram

State machine diagrams describe the dynamic aspects of a system or of part of a system. By graphically depicting the lifecycle of a model element, state machine diagrams show the different states that the model element can be in and how it responds to events.

To demonstrate this type of modeling, this topic diagrams the order processing in an E-Store system.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

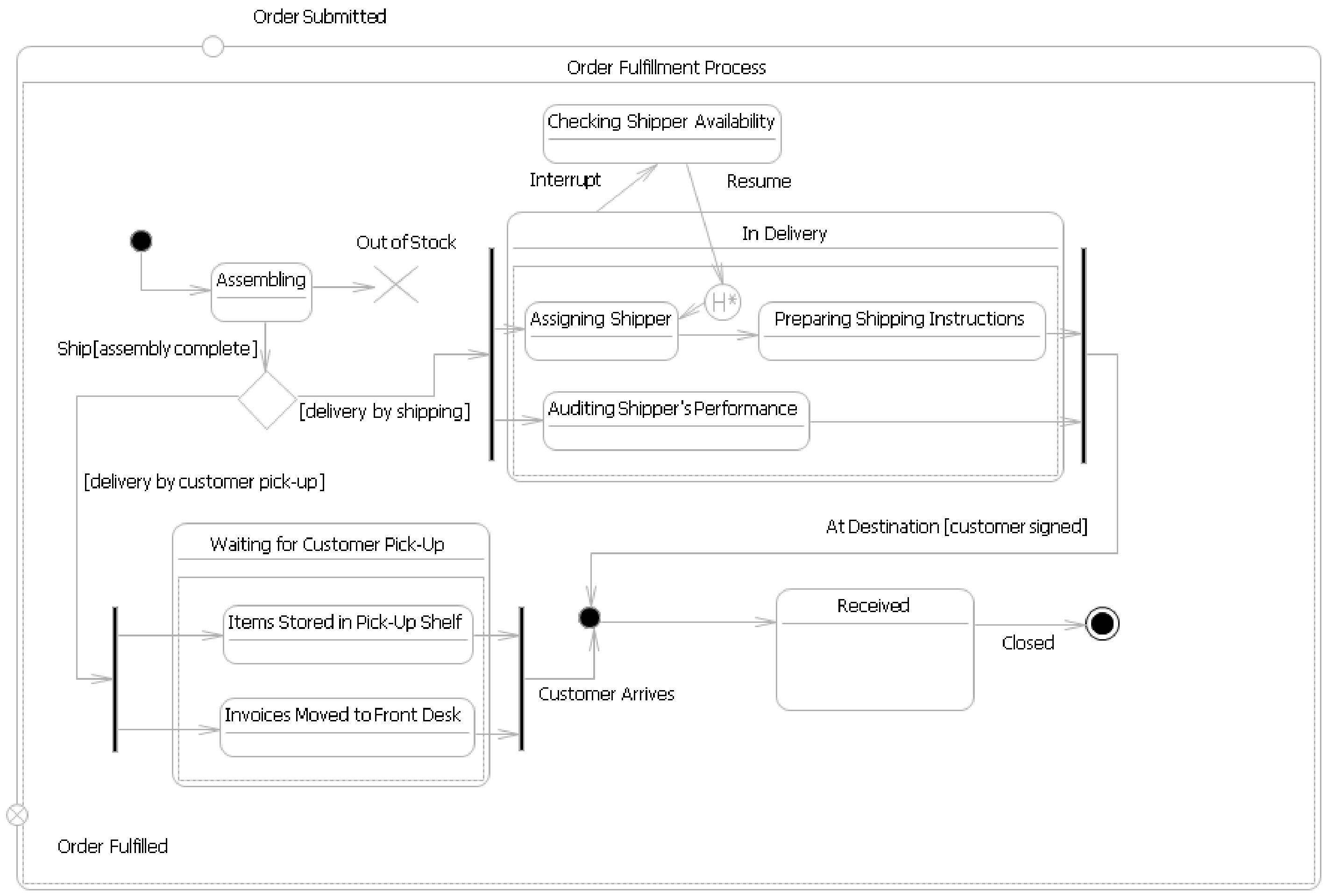
An important way to model the dynamic behavior of model entities is to use a state machine diagram. There are two kinds of state machine diagrams: one that describes the logical behavior of the system (or a part of the system) and one that describes the usage protocol of it.

A behavioral state machine diagram specifies the behavior of a model's elements, such as a class.

A protocol state machine diagram expresses a model's usage protocols, such as legal transitions, sequences, and lifecycles.

Sample Diagram

The following state machine diagram shows the order fulfillment process of an E-Store system. Execution of the diagram begins with the Initial node and finishes with Final state node.



Order Submitted
Order Fulfillment Process
Checking Shipper Availability
Interrupt
Resume
In Delivery
Out of Stock
Assembling
Ship[assembly complete]
Assigning Shipper
H*
Preparing Shipping Instructions
[delivery by shipping]
Auditing Shipper's Performance
[delivery by customer pick-up]
At Destination [customer signed]
Waiting for Customer Pick-Up
Items Stored in Pick-Up Shelf
Received
Closed
Customer Arrives
Invoices Moved to Front Desk
Order Fulfilled

The key graphical elements of a state machine diagram include states, events, and transitions. States represent a condition or situation during the life of an object in which it satisfies a certain condition, performs an activity, or waits for an event. Events specify a noteworthy time-space occurrence that alters a state. Transitions track the movement from one state to another in response to an event.

Many of a state machine diagram's palette elements represent pseudostates. Pseudostates are abstract groupings of the 10 different transient vertices in the state machine: initials, entry points, exit points, deep histories, shallow histories, forks, joins, junctions, terminates, and choices. Descriptions for pseudostates and other elements used for state machine diagrams can be found in the following table.

Palette Element	Canvas Element	Description
State Machine	**Order Fulfillment Process**	A state machine describes the behavior of a part of a system. State machine nodes are interconnected by transitions that get triggered by events. Once triggered, state machines execute a series of activities associated with elements of the state machine. A state machine owns one or more regions.

Palette Element	Canvas Element	Description
Simple State Composite State Submachine State	**Assembling**, **Checking Shipper Availability**, **Assigning Shipper**, **Preparing Shipping Instructions**, **Auditing Shipper's Performance**, **Items Stored in Pick-Up Shelf**, **Invoices Moved to Front Desk**, **Received** **In Delivery**, **Waiting for Customer Pick-Up** N/A	A Simple State has no regions or submachine state machines. A Composite State has either one region or several regions with mutually exclusive disjoint subvertices and a set of transitions. Substates within the regions are either contained by another state (indirect substates) or not (direct substates). A Submachine State specifies the insertion of the specification of a submachine state machine. You can use the Submachine to open the Submachine State by double-clicking the state. The state machine that contains the submachine state is called the containing state machine. The same state machine can be a submachine more than once in the context of a single containing state machine.
Region	Area containing substates in the **In Delivery** and **Waiting for Customer Pick-Up** composite states	Within a state, regions are used to group substates. When a state is created, it automatically comes with its own region, which can be deleted.
Final State	**Closed**	Signifies that the enclosing region is complete. When the region of a final state is directly contained in a state machine that has all its other regions complete, the entire state machine is complete.
Initial	Point before the **Assembling** state	The default vertex that provides the source for a transition to the default state of a composite state. A region cannot have more than one initial vertex.
Shallow History	N/A	A pseudostate that restores the most recent active substate of the containing state (that is, the configuration state that was active when the enclosing composite state last exited). A composite state cannot have more than one shallow history vertex.

Palette Element	Canvas Element	Description
Deep History	History point within the **In Delivery** composite state	A pseudostate that restores the most recent active configuration state that was active when the enclosing composite state last exited. A composite state cannot have more than one deep history vertex.
Fork	The vertical line following the **[delivery by shipping]** and **[delivery by customer pick-up]** transitions	A pseudostate that splits an incoming transition into two or more transitions that terminate on orthogonal target states (vertices in different regions of a composite state). Transition segments in a fork vertex have no guards or triggers.
Join	The vertical line preceding the **At Destination [customer signed]** and **Customer Arrives** transitions	A pseudostate that merges several transitions from source states in different orthogonal regions. Transitions in a join vertex have no guards or triggers.
Junction	The central point between the **At Destination [customer signed]** and **Customer Arrives** transitions	A pseudostate that connects transition segments into a single transition.
Choice	The central point between the **Ship [assembly complete]**, **[delivery by shipping]** and **[delivery by customer pick-up]** transitions	A pseudostate that performs a dynamic branch within a single transition.
✕ Terminate	**Out of Stock**	A pseudostate that, when activated, terminates the execution of the object that owns the state machine.
● Entry Point	**Order Submitted**	The point at which execution of the state machine or composite state starts. This pseudostate receives an external signal that identifies an internal state as a target.
⊗ Exit Point	**Order Fulfilled**	The point at which execution of the state machine or composite state completes. This pseudostate identifies an internal state as a source.
Transition	All arrows	Represents a direct relationship between the exit point of a source state and the entry point of a target state. In

Palette Element	Canvas Element	Description
		response to an event, a transition moves a state from one state configuration to another.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

UML 2.1 Use Case Diagram

Use case modeling is used to visually depict the functional requirements of a system based on how it behaves. To demonstrate this type of modeling, this topic uses the common Automated Teller Machine (ATM) system.

Note: Only those elements found on the Diagram Elements table below can be placed onto this type of diagram. For a list of elements available to all diagrams, refer to "*UML 2.1 Common Diagram Elements and Preferences* on page 9."

Definition

A use case diagram describes the required usages of a system, or what a system is supposed to do. By diagraming functional requirements through use cases, you can avoid stating a functional requirement that is not directly tied to specific user tasks needed to accomplish a business goal.

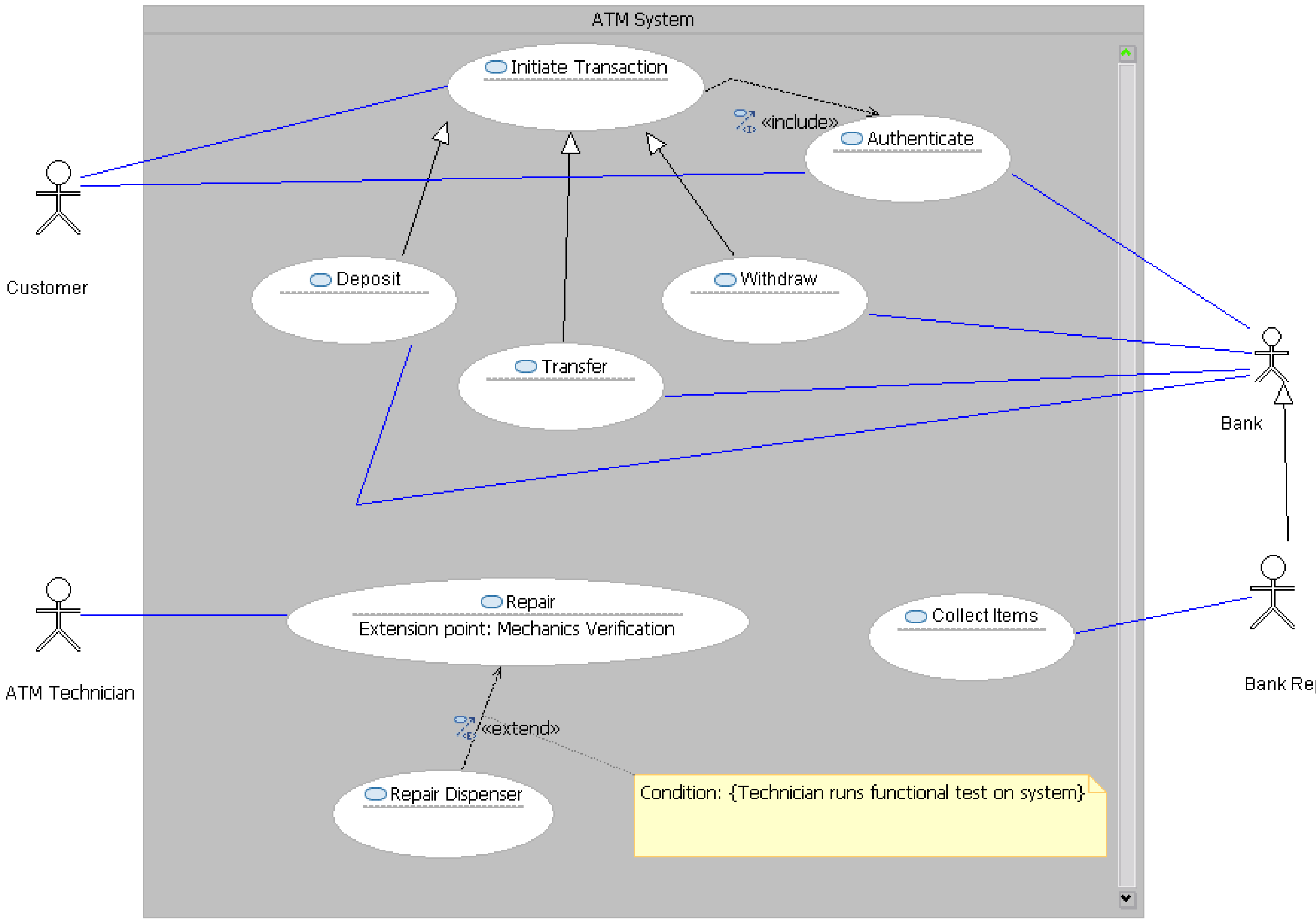
The diagrams can either depict a comprehensive system-wide view or a design-level view that further refines more general use cases. Use case diagrams can also be divided into flows and scenarios. A scenario depicts the use case instance from start to finish. A flow is a portion of a use case instance. It is important to remember that use case diagrams do not show the sequential flow of data between actors and the system (data can go in both directions). Use activity diagrams to depict sequential flow.

Sample Diagram

The following figure contains a diagram that shows an example of actors and use cases for an ATM system.

Refactoring the ATM system could yield many more use cases (for example, specifying accounts to withdraw from, performing card reader repairs, and so on). Furthermore, depicting a complete ATM use case model would probably be most useful using more than one diagram. For this topic, only a few basic behaviors of an ATM system are covered to highlight different roles and relationships, and which nodes on the palette are used for these.

The basic behaviors for the ATM system shown in the following diagram include a customer attempting a financial transaction with a bank, an ATM technician performing a repair, and a bank representative collecting materials.



The key concepts that take part in a use case diagram are actors, use cases, and subjects. Different types of relationships can exist between use cases, including *include*, *extend*, and *generalization* relationships. Descriptions for these concepts and the other available items on the diagram palette can be found in the table that follows.

Palette Element	Canvas Element	Description
Use Case	All oval-shaped elements on the diagram. The label above the horizontal line specifies the name of the use case. Any label below the horizontal line indicates an extension point.	The use case node is the action or sequence of actions that actors engage in to yield an observable goal. It can be any element that displays behavior, including a component, subsystem, or class. A use case is defined according to the needs of the actor. Relationships between use cases can be either extend, include, or generalization. Besides the use case's name and brief description, elements that describe use cases include flow or scenarios, special requirements, pre-

Palette Element	Canvas Element	Description
		and post-conditions, and extension points. Use the **Show as ➤ Classifier/UseCase** context menu to optionally display a use case as a Classifier rectangle.
Actor	Customer, ATM Technician, Bank, Bank Rep	An actor node is a role (usually a person or thing, depicted by a stick figure) outside of the system that interacts with the system through a use case to achieve an observable goal. Use the **Show as** context menu to optionally change the actor notation to display as a rectangle instead of a stick figure. Between actors, only a generalization relationship can exist.
Subject	ATM System	A subject node represents a system under consideration with which the actors and other subjects interact. The required behavior of the subject is described by the use cases. When a subject is created on a Use Case Diagram, a component is created in the namespace for the diagram canvas. Then after a use case is created on the subject, a new use case element is added to the subject's namespace and a relationship is formed between the use case and the subject.
Package	N/A	A package groups elements of the diagram and provides a namespace for those grouped elements. The Package node can be used to represent a *Classifier*, which is something (usually representing the Subject) that owns a use case.
Constraint	N/A	A constraint is a Boolean expression that restricts the extension of an element. It restricts by imposing a value that specifies additional semantics beyond what is imposed by other language constructs applied to that element. The element that owns the

Palette Element	Canvas Element	Description
		constraint must have access to Constrained Elements. By default, the value of a constraint is an Opaque Expression element.
Extension Point	Mechanics Verification (within the Repair use case)	The point in a use case at which behaviors of other use cases can be added. At an extension point, one use case's behavior replaces the behavior of or adds behavior to another use case.
Element Import	N/A	Identifies an element in another package that can be referenced using its name without a qualifier. A secondary package node located in the upper-left corner of diagram views (not shown above) contains element imports. The **ElementImport** compartment is collapsed by default.
Association	The blue lines between Actors and Use Cases	The interaction between an actor and a use case, represented by a solid line between them. These interaction links can be either unidirectional (indicating that an actor initiates the interaction) or bidirectional (indicating that an actor can participate in the interaction without initiating it). Association links can further be refined into multiplicities (how often the use case and actor interact), labels (roles specified at each end of the association), and direction (who initiates communication, although not necessarily a sequential flow of events).
Extend	Repair Dispenser	If a certain condition is met at a specific extension point, a use case can be extended to another use case. This results in an extend relationship between the use cases. For example, whenever the Repair use case in the diagram above reaches the value specified by the Mechanics Verification extension point, it is extended by the Repair Dispenser use case. An extended use case does not have a dependency on the use case it extends to.

Palette Element	Canvas Element	Description
		Extend links are indicated by a dashed arrow pointing from the use case providing the extension to the base use case.
Include	Authenticate Use Case	If one use case includes a basic behavior that other use cases show, you can separate the common behavior out into another use case and establish an include relationship. Include use cases are required in order for the original use case to execute successfully. For example, in order for the Initiate Transaction use case in the diagram above to complete, the actor must be verified through the Authenticate use case. Include links are indicated by a dashed arrow pointing from the base use case to the included use case.
Generalization	Withdraw, Transfer, and Deposit Use Cases; Bank Rep Actor	If one use case or actor inherits all the behavior of another use case or actor and furthermore extends it with additional behavior, a generalization link results. The arrow points to the more general of the two.
Constrained Element	N/A	Those elements of the diagram that are required to evaluate the conditions specified in a constraint. These references are displayed as links on the diagram.
Dependency	N/A	Elements whose semantics depend on the definition of a supplier element are in a dependency relationship with the supplier element. Dependency links are shown as dashed arrows between the two model elements with the arrowhead pointing to the supplier element.

Note: Depending on where the cursor is placed on the canvas, a hover context menu is displayed briefly with different Palette elements available for quick access.

Related Topics

UML 2.1 Diagrams on page 8

UML 2.1 Common Diagram Elements and Preferences on page 9

OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

Procedures

This section provides how-to information on using Together UML 2.1 diagrams. For a thorough understanding of these procedures, users should be familiar with the latest UML specification, available from the *Catalog of OMG Modeling and Metadata Specifications*.

Related Topics

Exporting a Model to UML 2.1

1. From the main menu, choose **File** ➤ **Export**. The **Export** wizard opens.
2. In the **Select** page of the wizard, choose **Modeling** ➤ **UML 2 Tools**, and click **Next**.
3. In the **Source Model Selection** page, specify the model from your workspace that you want to convert to UML 2.1. If you want the export to ignore synchronized package diagrams, check the **Ignore synchronized package diagrams** check box.

   Click **Next**.
4. In the **Target Project Selection** page of the wizard, specify the target project from your workspace that you want to convert to UML 2.1.

   Specify the conversion destination by adjusting the **Domain Model**, **Diagram Models Folder**, and **Trace Models Folder** fields as necessary. Click **Next**.
5. Select any available audits to expose conversion problems with the source model.
   The results are displayed in the **Audits Result List** pane.
6. Click **Fix All** to fix all problems found. Click **Next**.
7. In the **Hyperlinks and Requirement Traces** page, add or select any previously exported hyperlinks and requirement traces that you would like to retain.
8. Click **Finish**.

Related Topics

Procedures on page 53

Converting a UML 2.0 Model to a UML 2.1 Model

1. From the main menu, choose **File** ➤ **Export** ➤ **Modeling** ➤ **UML 2 Tools**.
2. On the **Source Model Selection** page, select the source model, or select any imported (referenced) models listed in the **Referenced Models List** pane.
3. Click **Next**.
4. On the **Target Project Selection** page, select a target project among those listed in the project tree in the top panel. In the bottom pane, specify a file for your domain model and a directory for your diagram model.
5. Click **Next**.
6. On the **Run Audits on Source Model** page, optionally select a set of audits to run to verify whether your source model contains any conversion mismatches. Results are displayed in the **Audits Result List** pane. Click **Fix All** to fix all problems found.

   Click **Next**.
7. In the **Hyperlinks and Requirement Traces** page, add or select any previously exported hyperlinks and requirement traces that you would like to retain.
8. Click **Finish**.

Related Topics

Procedures on page 53

UML 2.1 Diagrams on page 8

Creating a Required Interface

Required interfaces are what a component requires from other components in its environment in order to offer a full set of provided functionality.

1. On your diagram canvass, create a component and an interface node using the appropriate icons on the palette.
2. Create a Port on the boundary of the component using the **Port** icon on the palette.
3. In order for the derived link to be calculated, set the property type of the Port by first creating a TypingClass class and then setting the property type of Port to `TypingClass` in the **Properties** View.
4. Click the **Required Interface** icon on the diagram palette.
5. Click the client port and drag the mouse to the interface node.

UML 2.1 lets you reroute the source and target of the required interface links. If you reorient the target of the link, the target of the genuine link is changed. If you reorient the source of the link, the original usage link is changed.

Related Topics

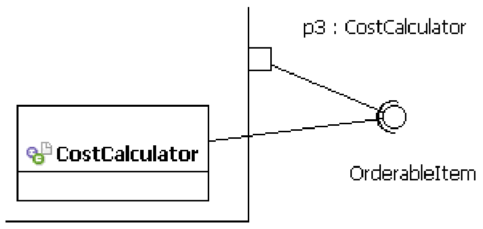
Procedures on page 53

Provided and Required Interface Links of a Port on page 6

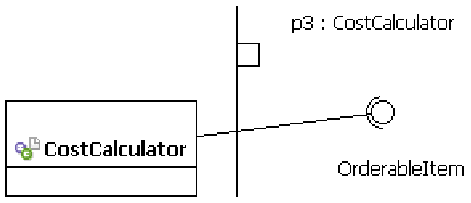
Required Interface on page 7

Managing the Visibility of Provided and Required Interface Links

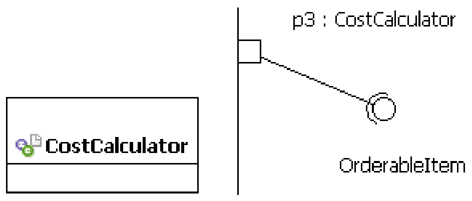
Because they are derived links, provided and required interface links usually represent another occurrence of an already existing link. For example, in the following image, a single Required Interface link is a link from both a typing classifier of a port, and from a port to an interface.



1. To manage these links on the diagram, from the main menu choose **Window ➤ Preferences**.
2. Expand the **UML2 Diagrams** feature, and select **View Filters** under the **UML Component Diagram** node. The **Hide Derived Links** option lets you hide links that either do not exist or that are calculated, such as links from ports. With this option turned on, the preceding component diagram would have its derived link hidden, as seen in the following image.



The **Hide Genuine Links** option lets you hide original links, such as Usage or **InterfaceRealization** links. With this option turned on, the component diagram would have its original link hidden, as seen in the following image.



Related Topics

Procedures on page 53

Provided and Required Interface Links of a Port on page 6

Working with Associations

An association defines a relationship between two or more TypedElements, such as properties of a class.

Related Topics

Procedures on page 53

Creating an Association on page 56

Creating an Association from Existing Properties on page 56

Change the Aggregation Type of an Association on page 57

Editing the Properties of AssociationEnds on page 57

UML 2.1 Diagrams on page 8

Creating an Association

An association is a relationship between TypedElements, such as properties of a class.

1. On your diagram canvass, create two classes using the **Class** icons on the palette.
2. On the diagram palette, click the **Association** icon under the Links group.
3. On your diagram, click the client class and drag the mouse to the target class node.

Related Topics

Working with Associations on page 56

Creating an Association from Existing Properties

1. On your diagram canvass, create classes A and B using the **Class** icons on the palette.
2. Create a Property in Class A. In the **Properties** View, set the type of this property as B.
3. Select the Class A Property. Using the context menu of the element, choose **Create Association to ➤ B**.

   A new Association from a previously existing property is created.

Related Topics

Working with Associations on page 56

Change the Aggregation Type of an Association

An aggregation is a type of association in which the child class might or might not continue to exist if the parent class is deleted. If the child class is not owned by the parent and continues to exist when the parent is removed, the aggregation is shared. Such a relationship is distinguished from a composite association, in which the child class is owned by the parent class and ceases to exist when the parent class is removed.

1. Select the appropriate Association Link on the diagram.
2. Using the context menu, choose **Association Type** and choose either **None**, **Shared**, or **Composite**.

Related Topics

Working with Associations on page 56

Editing the Properties of AssociationEnds

An aggregation is a type of association in which the child class might or might not continue to exist if the parent class is deleted. If the child class is not owned by the parent and continues to exist when the parent is removed, the aggregation is shared. Such a relationship is distinguished from a composite association, in which the child class is owned by the parent class and ceases to exist when the parent class is removed.

1. Select the appropriate Association Link on the diagram.
2. Using the context menu, choose **Association Type** and choose either **None**, **Shared**, or **Composite**.

Related Topics

Working with Associations on page 56

Running a UML2Tools Report on a UML 2.1 Diagram File

1. Right click on the UML 2.1 diagram file in the **Navigator** view and select **Run UML 2 Tool report**.
2. Select the report format (**html** or **pdf**) from the **Format** drop down list.
3. Specify the output path in the **Output Path** field.
4. Click **OK**. The report displays when complete.

Related Topics

Procedures on page 53

UML 2.1 Diagrams on page 8

Working with UML 2.1 Profiles

Related Topics

Procedures on page 53

A Typical Scenario of Creating a Profile on page 58

Creating a Project on page 59

Creating a Profile on page 60

Populating Profile Diagrams on page 61

Defining Profiles on page 62

Registering Profiles on page 63

Applying Profiles on page 63

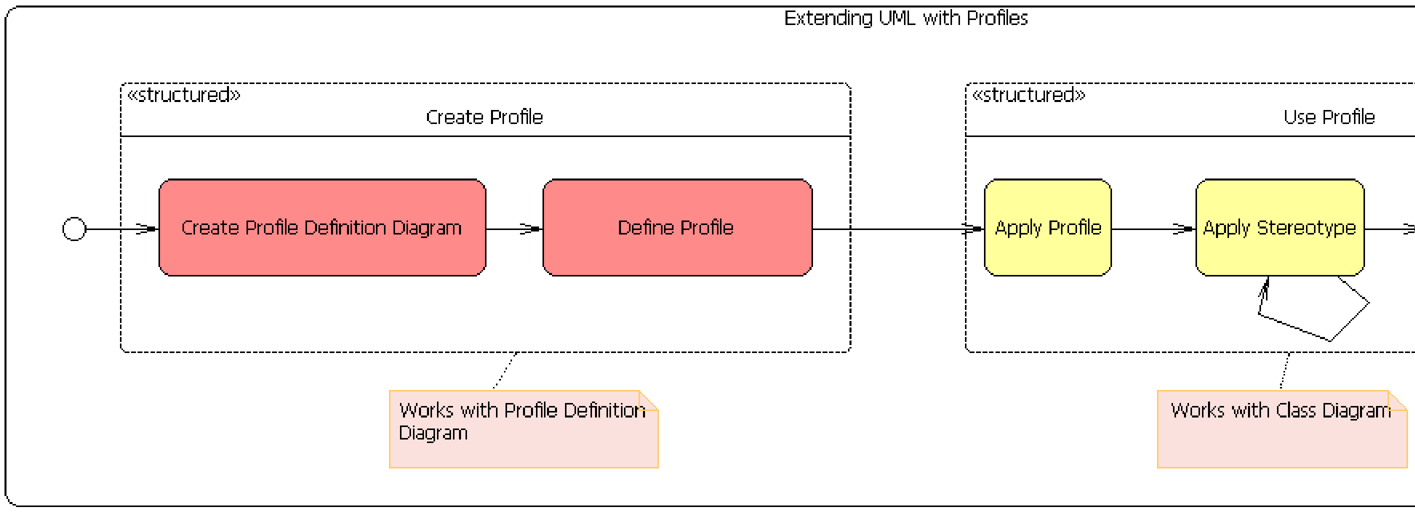
Applying Stereotypes on page 64

A Typical Scenario of Creating a Profile

UML is a standard modeling language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other nonsoftware systems. Although the general modeling concepts of UML are sufficient for the majority of modelers, some elements cannot be expressed directly using UML constructs. In these situations, users can use an extension mechanism, known as a UML Profile, to collectively customize UML for a particular domain using domain-specific concepts and techniques. Profiles provide refinement mechanisms, such as stereotypes and constraints, that allow metaclasses from existing UML domain metamodels to be extended so they can be adapted for different purposes without changing the metamodel. Consequently, you can interpret the semantics of a profile in the context of the UML specification.

There are several advantages to customizing UML with a Profile. Profiles provide a syntax to constructs without notations, and they contribute additional constraints that restrict how a metamodel and its constructs are used. They also provide alternate notations for UML symbols and add semantics that are otherwise nonexistent or unspecified in the domain metamodel. In addition, you can use profiles to configure transformation rules between models.

A graphical depiction of a basic profile definition workflow follows. First you create a Profile using the Profile Definition Diagram Editor. Then to apply the profile, you create a class diagram with an element of the extended metaclass. You then apply the defined stereotypes to this element. Ultimately, there will be two different diagrams and models, each dependent on the other.



Use the following steps to create a profile:

1. *Creating a Project* on page 59.
2. *Creating a Profile* on page 60.
3. *Populating Profile Diagrams* on page 61.
4. *Defining Profiles* on page 62.
5. *Registering Profiles* on page 63.
6. *Applying Profiles* on page 63.
7. *Applying Stereotypes* on page 64.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Creating a Project

1. On the main menu, choose **Window ➤ Open Perspective ➤ Other**.
2. Choose the **Resource** perspective and click **OK**.
3. On the main menu, choose **File ➤ New ➤ Project**.
4. Under the **General** node, choose **Project** from the list of wizards and click **Next**.
5. Supply a project name, such as `Introduction to UML2.1 Profiles`, and click **Finish**.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Creating a Profile

The profile element is at the root of a UML profile and adapts a metamodel to a particular domain or platform by extending the metamodel.

1. In the Project Explorer view, select your project. On the main menu, choose **File ➤ New ➤ Other**.
2. Under the **UML 2.1 Diagrams** node, choose **Profile Definition Diagram** from the list of wizards and click **Next**.
3. Select `Introduction to UML2.1 Profiles` as the parent folder for the diagram model and supply a name for your profile diagram.

   Tip: Profile diagrams always end with a `.umlprofile` extension.

4. If you would like your domain and diagram models to have the same name, click **Finish**. To supply a different name for you domain model, click **Next**.
5. Select `Introduction to UML2.1 Profiles` as the parent folder for the domain model and supply a name for your profile diagram.

   Tip: Resources that contain a profile always end with a `.profile.uml` extension.

   Click **Finish**.
6. If the Properties view is not yet displayed, from the main menu choose **Window ➤ Show View ➤ Properties**. The new diagram also created a new model. This default model has a single specialized package with a default name of profile. The package has the profile profile applied to it. In this way, the profile mechanism is used self-referentially for the creation of new profiles. To change the name of the profile, select the element in the upper-left corner of the new diagram and supply a value for its Name property, such as `EJB` for Enterprise JavaBeans. Because this name serves as an ID of the profile, the Name property should never be empty.

   The `EJB` profile element appears on the created diagram and is the root of your UML 2.1 profile.

   Note: In UML 2.1, a profile is created with some standard primitive types already predefined, including Boolean, String, and Integer. You can have your profile reference other primitive types from libraries by importing those primitive types (**UML Editor ➤ Package ➤ Import Type...**). To import additional elements, click the **ElementImport** node from the palette and then click the <<`profile`>> polygon at the upper left part of the diagram. Because the **ElementImport** compartment is collapsed by default, you must expand it.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Populating Profile Diagrams

Note: Using the Profile Definition Diagram Editor provided by Together, you can add all diagram elements graphically in the Modeling Perspective. To add elements on the diagram canvas, click the appropriate icon on the palette and then click your diagram canvas to add the element.

1. From the profile diagram palette, click the **Metaclass** element and click within your diagram to add the element. Name the metaclass `Component`.

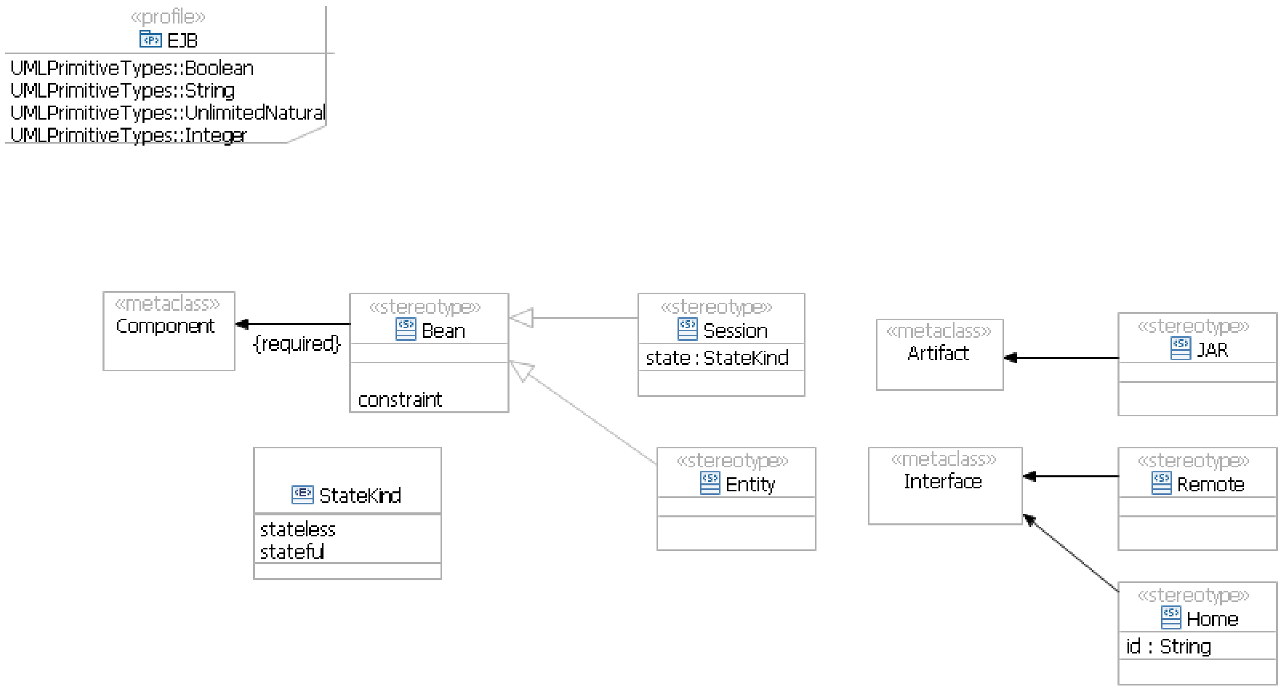
   Note: Because it is the name of a referenced metaclass, you can use the code assist feature (Ctrl+space) to bring up the list of available metaclass names.

2. From the profile diagram palette, click the **Stereotype** element and click within your diagram to the right of the `Component` metaclass so that the element is added. Name the stereotype `Bean`.
3. Connect the stereotype with your metaclass by clicking the **Extension** element from the palette and dragging the cursor from the stereotype to the metaclass.
4. From the profile diagram palette, click the **Stereotype** element again and click within your diagram to the right of the first stereotype you created so that the element is added. Name the stereotype `Session`.
5. Connect the `Session` stereotype with your `Bean` stereotype by clicking the **Generalization** element from the palette and dragging the cursor from the `Session` stereotype to the `Bean` stereotype.
6. From the profile diagram palette, click the **Enumeration** element and click within your diagram to add the element. Name the enumeration `StateKind`.

   An enumeration contains constant values for a new data type.
7. From the profile diagram palette, click the **Literal** element and click within the Enumeration you added in the last step to add the Literal. Name the enumeration literal `stateless`. Add another enumeration literal named `stateful`. Enumeration literals specify the user-defined data value for the enumeration.
8. Click the OCL Constraint element from the palette and click within `Bean` stereotype to add the OCL Constraint. Supply an opaque expression that specifies how the stereotype is constrained. For example, `A Bean must realize exactly one Home Interface` restricts not only the extension of the Bean stereotype but also gets extended to the `Component` metaclass.

   The language of the constraint is indicated above the constraint text, such as `{OCL}`. You can edit the OCL Constraint directly in the diagram or in the **Properties** view. The Constraint editor in the **Properties** view highlights errors in the OCL code and provides code-assist.

   To disable constraints, click **Window** ➤ **Preferences** ➤ **Model Validation** ➤ **Constraints**, select the UML2 Tools Constraints category, and specify which constraints to disable by unchecking its check box.
9. Continue to populate the profile diagram and extend its elements with metaclasses and stereotypes as follows:



The `Session` stereotype extends the `Bean` stereotype, which extends the `Component` metaclass. Through these extensions, the state property of the `Session` stereotype is appended to the `Component` metaclass. In addition, the `Artifact` and `Interface` metaclasses define other elements to be extended.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Defining Profiles

After your profile diagram has been created, you must define your profile so that it will be saved in the UML model and allow subsequent use of profile content. By defining profiles at the metamodel level, all created extensions can appear as part of that metamodel. This is possible when the contents of the profile are converted to an equivalent representation of the metamodel that then get stored as an annotation on the profile.

1. Right-click within your profile diagram.
2. From the context menu of the profile diagram, choose **Profile** ➤ **Define**.

Note: A profile gets defined once by the person who first checks the profile in to the repository. Subsequent users of the profile do not have to define it again. However, this procedure must be repeated whenever the profile changes.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Registering Profiles

Adding your profile to the registry makes later applications of the profile more simplified. In order for your profile to display in the **Apply Profile** context menu, it is important to register the profile.

1. Open the profile's plug-in descriptor `plugin.xml` file in your workspace.
2. Register the profile using the `org.eclipse.uml2.uml.dynamic_package` extension point.

   The profile will be deployed in the platform later. For example, to register the UML Standard profile, edit the `plugin.xml` file as follows:

```
<extension      point="org.eclipse.uml2.uml.dynamic_package">
<profile uri="http://www.eclipse.org/uml2/schemas/Ecore/5" location=
"platform:/plugin/org.eclipse.uml2.uml.resources/profiles/Ecore.profile.uml#_0"/>
</extension>
```

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Applying Profiles

All profiles are restricted forms of a metamodel. Because the two are related, profiles must always be used in conjunction with their referenced metamodels. An import relationship between a profile and the metamodel's metaclass is what allows stereotypes to be used as extensions. In order for these stereotypes to extend the elements in a package, the profile must be applied.

1. If the profile has not yet been added to the registry in the `plugin.xml` file, the corresponding `*.profile.uml` resource must be loaded so that the profile appears in the **Apply Profile** context menu list. To load the profile, create a class diagram in your project and add a class element. Right-click in the diagram and choose **Load resource...** from the menu.
2. When the wizard appears, click **Browse Workspace...** and select the required resource in your project (the `*.profile.uml` file) that contains the profile. Click **OK**.
3. Right-click in the diagram and choose **Apply Profile ➤ <profile name>**.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

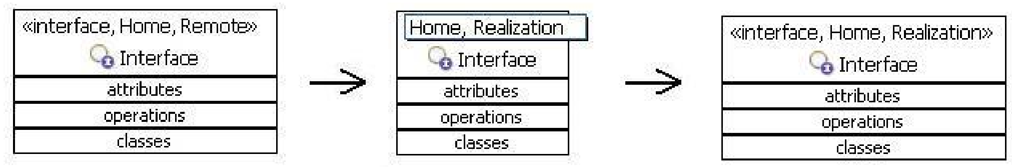
Applying Stereotypes

After a profile has been applied to a package, you can apply the profile's defined stereotypes to instances of the metaclass. Stereotypes applied to elements extend those elements as defined in the stereotype properties. Stereotypes can be applied to both nodes and links.

1. In the class diagram that you created, select one of the nodes or links with which you want to extend the metaclass.
2. In the context menu of the selected diagram element, right-click and choose **Apply Stereotype ➤ <stereotype name>**.
3. Alternatively, click within the stereotype label of your class diagram element to select it and type a comma-separated list of stereotypes in the label.

   Note: Currently, UML 2.1 permits stereotypes to be applied only to classifiers.

   Stereotypes added to the list will be applied, and stereotypes removed will be unapplied. In the following example, the **Realization** stereotype gets applied as it replaces the **Remote** stereotype, which gets unapplied.

   The name of the final applied stereotype appears between guillermets.
4. Edit the properties of the applied stereotype as necessary using the **Tagged Value** tab of the Properties View.

Related Topics

Working with UML 2.1 Profiles on page 58

UML 2.1 Profile Diagram on page 40

Reference

Related Topics

UML 2 Tools Wizard on page 65

UML 2.1 Export Audits on page 66

UML 2 Tools Wizard

File ➤ Export ➤ Modeling ➤ UML 2 Tools

The UML 2 Tools Wizard is used to convert Together UML 2.0 models to Eclipse UML 2.1 (UML2Tools) models.

Wizard Pages

The UML 2 Tools wizard contains the following pages:

Source Model Selection	Displays Together projects that are available to convert. Select the source model, or select any imported (referenced) models listed in the **Referenced Models List** pane. Checking the **Ignore synchronized package diagram** check box prevents the diagram with the package contents from being converted.
Target Project Selection	Select a target project among those listed in the project tree in the top panel. In the bottom pane, specify a file for your domain model and a directory for your diagram model.
Run Audits on Source Model	If you want to verify whether your source model contains any conversion mismatches before running the conversion, select a set of audits to run. The results are displayed in the **Audits Result List** pane.
Hyperlinks and Requirement Traces	Add or select any previously exported hyperlinks and requirement traces that you would like to retain.

Related Topics

Reference on page 65

UML 2.1 Diagrams on page 8

UML 2.1 Export Audits on page 66

UML 2.1 Export Audits

To the largest extent possible, the automated model conversion from Together UML 2.0 to UML 2.1 preserves both diagram and model semantics. This includes element layout, formatting, and properties. However, some modeling concepts might change or might not be supported by Together in the same way. The UML 2.1 Export Audits provide advance warning about which model elements might not be converted as expected.

The following audits are available for exporting to UML 2.1.

Audit	Description
Part-Port Audit	Identifies incorrectly converted port elements from previous versions of Together and provides automatic repairing of the elements so that the ports are correctly converted.
OCL Constraint Audit	Searches for OCL constraints that have been added to the model. Because of differences between the OCL and metamodel implementations, OCL Constraints in the model might be converted with syntax changes or incorrect OCL expressions.
Applied Profile Audit	Identifies elements on which a UML Profile has been applied and alerts the user that custom viewmaps for the elements are lost when the profile gets extracted to a separate model during conversion.
Custom Properties Audit	Identifies elements on which custom properties have been applied and alerts the user that these elements will not be converted.
Full-Screen Diagram Audit	Identifies diagrams that represent the StateMachine using a full canvas and alerts the user that UML 2.1 converts these diagrams to show the StateMachine as an explicit element added to the canvas.
Imported Model Audit	Identifies models that contain imported elements and alerts the user that additional steps are required, including converting the referred model first and supplying models in the workspace that will convert the model with imported elements.
Element with Hyperlink Trace Audit	Identifies models that contain elements with hyperlink traces and alerts the user that these elements will be preserved by a generated trace model.

Audit	Description
Element with Requirement Trace Audit	Identifies models that contain elements with requirement traces and alerts the user that these elements will be preserved by a generated trace model.

Related Topics

Reference on page 65

UML 2 Tools Wizard on page 65