Standardized Technical Architecture Modeling

Conceptual and Design Level

Version 1.0

March 2007

Copyright 2007 SAP AG. All Rights Reserved

No part of this document may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Excel, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, System i, System i5, System p, System p5, System x, System z, System z9, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, Informix, i5/OS, POWER, POWER5, POWER5+, OpenPower and PowerPC are trademarks or registered trademarks of IBM Corporation.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

The information in this document is proprietary to SAP. No part of this document may be reproduced, copied, or transmitted in any form or for any purpose without the express prior written permission of SAP AG.

This document is a preliminary version and not subject to your license agreement or any other agreement with SAP. This document contains only intended strategies, developments, and functionalities of the SAP® product and is not intended to be binding upon SAP to any particular course of business, product strategy, and/or development. Please note that this document is subject to change and may be changed by SAP at any time without notice.

SAP assumes no responsibility for errors or omissions in this document. SAP does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material. This document is provided without a warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

SAP shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials. This limitation shall not apply in cases of intent or gross negligence.

The statutory liability for personal injury and defective products is not affected. SAP has no control over the information that you may access through the use of hot links contained in these materials and does not endorse your use of third-party Web pages nor provide any warranty whatsoever relating to third-party Web pages.

Table of Contents

1 Goal and Scope ... 3
1.1 Scope of Standardized Technical Architecture Modeling ... 3
1.2 Currently Out of Scope ... 3
2 Specification of Diagram Types ... 4
2.1 Diagram Types for Structural Descriptions ... 5
2.1.1 Component/Block Diagram ... 5
2.1.2 Class Diagram ... 9
2.1.3 Package Diagram ... 14
2.2 Diagram Types for Behavioral Descriptions ... 15
2.2.1 Use Case Diagram ... 15
2.2.2 Activity Diagram ... 17
2.2.3 Sequence Diagram ... 21
2.2.4 State Machine Diagram ... 25
2.3 UML Profiles ... 27
3 Appendix A: Component/Block Diagram ... 28
3.1 Overview ... 28
3.2 Abstract Syntax ... 29
3.3 Class Descriptions ... 32
3.3.1 Access (from BasicBlockElements) ... 32
3.3.2 AccessEnd (from BasicBlockElements) ... 33
3.3.3 AccessPort (from BasicBlockElements) ... 34
3.3.4 Agent (from BasicBlockElements) ... 34
3.3.5 BlockElement (from BasicBlockElements) ... 35
3.3.6 Channel (from BasicBlockElements) ... 36
3.3.7 ChannelEnd (from BasicBlockElements) ... 37
3.3.8 ChannelPort (from BasicBlockElements) ... 38
3.3.9 Generation (from AdditionalBlockConcepts) ... 38
3.3.10 ModifyAccess (from BasicBlockElements) ... 39
3.3.11 MultiplicityDots (from AdditionalBlockConcepts) ... 39
3.3.12 Pointer (from AdditionalBlockConcepts) ... 40
3.3.13 ProtocolBoundary (from AdditionalBlockConcepts) ... 41
3.3.14 ReadAccess (from BasicBlockElements) ... 42
3.3.15 Storage (from BasicBlockElements) ... 42
3.3.16 WriteAccess (from BasicBlockElements) ... 43
4 Links & Literature ... 45

Table of Figures

Figure 2–1 Conceptual Level Component/Block diagram 8
Figure 2–2 Design Level Component/Block diagram 8
Figure 2–3 Class diagram Conceptual Level 12
Figure 2–4 Class diagram Conceptual Level 12
Figure 2–5 Class diagram Design Level 13
Figure 2–6 Package Diagram 14
Figure 2–7 Use Case diagram 17
Figure 2–8 Activity Diagram Conceptual Level 20
Figure 2–9 Sequence diagram Conceptual Level 23
Figure 2–10 Sequence diagram Design Level 24
Figure 2–11 State Machine Diagram 26
Figure 3–1 Dependencies between packages described in this chapter 29
Figure 3–2 The metaclasses that define the basic Block diagram elements 30
Figure 3–3 Additional Block diagram concepts used for didactical purposes 31
Figure 3–4 Additional Block diagram concepts describing special dependencies 31
Figure 3–5 Notation of agents accessing storages 33
Figure 3–6 Graphical simplification of assembly and delegation accesses 33
Figure 3–7 Example showing agents 35
Figure 3–8 Grouped agents accessing a storage 35
Figure 3–9 Variations of channels 37
Figure 3–10 Usage of multiplicity dots 40
Figure 3–11 Use of a protocol boundary 42
Figure 3–12 Nesting of storages 43

© Copyright SAP AG 2007

1 Goal and Scope

1.1 Scope of Standardized Technical Architecture Modeling

SAP's standardized technical architecture modeling defines and describes

- which diagram types are allowed to model technical architecture at SAP (see Chapter 2)
- what elements in a certain diagram type are allowed, optional or prohibited(see Chapter 2)
- which extensions of the UML meta model have been made for specific diagram types (see Chapter 2 and Appendix A)
- the semantics of newly added elements in diagram types and how those elements can be used (see Chapter 2)

SAP's standardized technical architecture modeling provides as few models as possible but as many as necessary, whereas the primary requirement of simplicity will be fulfilled.

This standardized modeling does not currently focus on how business processes or business entities within SAP solutions should be modeled. Therefore, the standard is not interfering with any business process-oriented or business driven modeling methods. However, necessary alignment with existing methods is in investigation and will be covered in one of the next versions.

1.2 Currently Out of Scope

This document will not discuss tools for modeling. Even though this document defines the diagram types and elements, which can be used to model technical architecture, the document does not prescribe the abstraction levels for those diagrams. The correct abstraction level for a given subject depends very much on the subject itself and its surrounding context.

This document is not supposed to be used as (self-)training material for any modeling technique. In fact, it is rather to be seen as a formal specification of standardized modeling on top of the existing specification of UML. Therefore, this document is intended to be read by people with at least basic understanding of modeling in UML. All example diagrams included in the document are to be seen as pure notational examples without being embedded in a special context.

© Copyright SAP AG 2007

2 Specification of Diagram Types

All diagram types, which are included in this version of the standardized technical architecture modeling, are based on standard diagram types of the *Unified Modeling Language Version 2.0*. The following specification documents published by the OMG are the basis for this version:

- Meta-Object Facility (MOF™) Core Specification v2.0 (January 2006)
- OCL Specification v2.0 (June 2005)
- Unified Modeling Language™ (UML®) Infrastructure v2.0 (March 2006)
- Unified Modeling Language™ (UML®) Superstructure v2.0 (August 2005)

Due to the fact that UML is widely accepted as standard notation, it is assumed that it is also more or less well known by the majority of software developers and architects inside and outside SAP.

When modeling high-level architectural structures, Block diagrams (originated from the Fundamental Modeling Concepts) have successfully served their purpose in the past. They effectively help avoiding free-style pictures of conceptual architecture and improve communication due to their simplicity. Block diagrams only consist of a few modeling elements, are easy to learn and intuitively understandable. In order to improve integration of Block diagrams, UML Component diagrams have been formally extended as described in detail (see Appendix A: Component/Block Diagram).

The unification of modeling methods is not only motivated by the improvement of modeling methods but also by the reduction of diagram types.

The following diagram types are sufficient to fulfill the modeling needs for technical architecture. They are specified in detail in the following sections.

Diagram Types for Structural Descriptions[1]

SAP's standardized technical architecture modeling covers the following diagram types for structural descriptions.

- Component/Block Diagrams
- Class Diagrams
- Package Diagrams

Diagram Types for Behavioral Descriptions

SAP's standardized technical architecture modeling covers the following diagram types for behavioral descriptions.

- Use Case Diagrams
- Activity Diagrams
- Sequence Diagrams
- State Machine Diagrams

Most of the diagram types are allowed to be used on two different abstraction levels: on *Conceptual Level* and *Design Level*[2]. Diagrams on conceptual level characteristically contain more abstract information, which describe architectural concepts in a broader context. Diagrams on design level primarily focus on the reflection of code structures. It is assumed that on design level the model exceeds the granularity on conceptual level. Therefore, the number of allowed and optional elements on design level usually increases.

---

[1] UML diagram types can be differentiated between diagram types for describing structural aspects of a software solution and diagram types describing the behavioral aspects of a software solution.

[2] It is necessary to *explicitly* clarify which particular abstraction level is used for a diagram. One approach would be to explain the abstraction level in the diagram's caption or the surrounding text. Alternatively, it is also possible to use the stereotype «conceptual» or «design» within the diagram itself.

© Copyright SAP AG 2007

More specific characteristics of the abstraction levels for a particular diagram type are explained in the section describing the diagram type.

The following sections contain lists denoting permissions for diagram elements related to the corresponding diagram types covered by the standard. A more detailed description is provided for each specific diagram type.

The columns *Conceptual Level* and *Design Level* in the element tables indicate whether an element is applicable for that level or not. Please refer to the following symbols:

- **X** indicates that those elements are to be used in regular case (SHOULD)[3]
- **O** (OPTIONAL) indicates that those elements may be used in some cases (MAY)
- **-** indicates that those elements are not to be used on the respective level in any case (MUST NOT)[4]

Please note that all elements that are not explicitly listed here are *disallowed*. The element lists do contain some of the most common disallowed elements already. However, it is important that these lists are not exhaustive.
This also implies that elements, which are newly introduced in versions later than the UML version covered (see beginning of section 2) in TAM are *disallowed* by default. TAM would have to be modified in order for these new elements to be allowed.

2.1 Diagram Types for Structural Descriptions

2.1.1 Component/Block Diagram

2.1.1.1 Purpose

The new Component/Block diagram as described in this document is based on UML component diagrams and integrates aspects of the FMC[5] Block diagram. It intends to describe a static structure of a (software) system and, thus, provides a conceptual view on the architecture of this particular system. An extension to the UML meta model is defined within the scope of this standardized modeling in order to formally extend the basic UML component diagrams. For further details on semantics and graphical notation please refer to Appendix A: Component/Block Diagram.

Basically, the Component/Block diagram describes a system using active (*agents*) and passive (*storages*, *channels*) visual elements. The connection is established by arcs identifying how agents can access passive elements (read, write, and modify). On conceptual level only the block diagram elements agents, storages, channels, and accesses are used to describe a system. On design level it is possible to, additionally, integrate UML component elements (component, interfaces, and connectors) describing specific regions of the system in more detail. Characteristically, these regions are of central interest for implementation.

---

[3] The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. If a certain diagram element marked as SHOULD is not necessarily needed in a particular diagram, this is a *valid reason* (in the sense of RFC 2119) not to use it. This is because not each and every element is required to complete a diagram in any circumstances.

[4] In order to achieve a common understanding of standardized diagrams, it is important that the number of allowed elements is limited. An effective restrain of diagram elements is hereby achieved by marking the no-use elements as "MUST NOT" instead of "SHOULD NOT".

[5] FMC stands for "Fundamental Modeling Concepts", a modeling technique that has been used at SAP since 1990. Refer to http://www.fmc-modeling.org for more information.

2.1.1.2 Abstraction Levels

Component/Block diagrams are available on conceptual level and design level.

2.1.1.3 Diagram Elements[6]

Element	Brief Description	Conceptual Level	Design Level
Agents Agent	Active elements, which are capable of doing a certain action. Agents can contain agents, storages, subsystems, components and classes (nesting).	X	X
Channels R ►	Passive elements that are used for communication between agents. All transferred information is volatile. Optionally with request and/or data flow direction.	X	X
Storages Storage	Passive elements on which agents can act upon. Usually, storages contain data of any kind. Storages can be nested within agents, storages and components/subsystems.	X	X
Accesses (Read, Write, Modify)	Access arcs define the way, how agents *can* access a passive element.	X	X
Components, Classes, Subsystems Component	Implementation-close (active) part of the described software system, which is embedded in its environment using well-defined interfaces (provided and required). Can implicitly include internal state. Nesting components, classes and subsystems within agents is allowed, whereas components are only allowed to contain components, classes and storages. Subsystems can contain subsystems, components, classes and storages.	-	X

[6] Please note that, in these tables, the according specializations of meta classes, defined in Appendix A: Component/Block Diagram, are treated separately from their generalizations (e.g. *Agent* as specialization of *Component*).

© Copyright SAP AG 2007

Element	Brief Description	Conceptual Level	Design Level
Interfaces (provided, required)	Interfaces define connection points between components and their environment. Components provide a number of services (provided Interfaces) and rely on services that are provided by others (required Interfaces)	-	X
Connectors (assembly, delegation)	*Assembly* connectors define connections between matching pairs of provided and required interfaces. *Delegation* connnectors describe a forwarding of information. They are drawn between a port and an according destination (e.g. interface of the same type).	-	X
Ports Component	Interaction points between components and their environment. Only if intended to be shown explicitly (can be invisible).	-	O
Multiple Agents/Storages Multiple Agent Storage 1 ⋮ Storage n	Explicit visualization of multiple agents or storages by either staggering the elements or showing three dots between elements of the same kind. Only allowed for Agents and Storages, not for Components, Subsystems, etc.	O	O
Protocol Boundaries Protocol	Protocol boundaries usually partition a diagram in order to accentuate certain boundaries in communication.	O	O
Dependencies «create»	Visualiziation of certain dependencies between components, agents and/or storages. Additionally to UML standard dependencies, the following types are defined: *Create* (agent/component ➔ agent/component): dynamic generation of active system parts (component or agent) e.g. out of a meta-repository *Pointer* (storage ➔ storage): reference-like relationship between contents of storages	O	O
Storages containing Agents	Storages are not allowed to contain agents or components. (passive elements are not allowed to contain active ones)	-	-

© Copyright SAP AG 2007

2.1.1.4 Example Diagrams

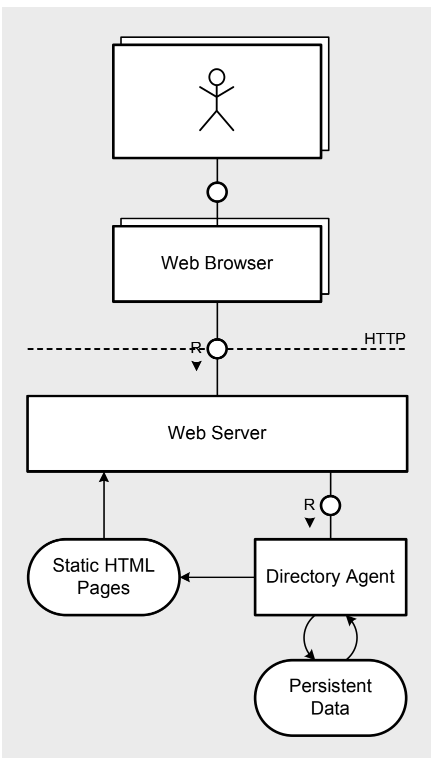

Figure 2–1 Conceptual Level Component/Block diagram

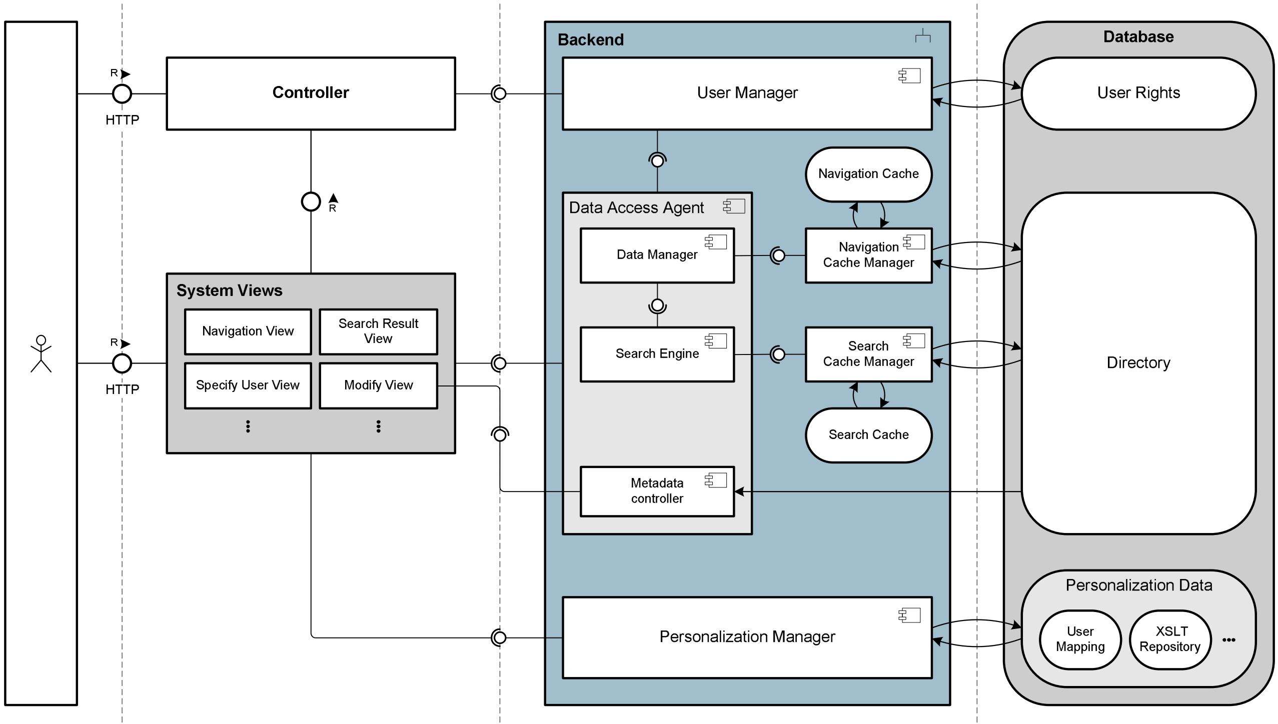

Figure 2–2 Design Level Component/Block diagram

© Copyright SAP AG 2007

2.1.2 Class Diagram

2.1.2.1 Purpose

Class diagrams describe the structure of (object-oriented) software systems.

2.1.2.2 Abstraction Levels

Class diagrams are available on conceptual level and design level.

On conceptual level, class diagrams show on a rather coarse level the system entities and their relationships. A class diagram can be a glossary by being more formally than a list explaining the relations between the terms used. For example, database table modeling in a way comparable to Entity/Relationship diagrams (E/R diagrams).

On design level concrete classes of the implementation are represented as classes in the model. Primary goal is the visualization of issues directly related to object-oriented implementation with all relevant details.

2.1.2.3 Diagram Elements

Element	Brief Description	Conceptual Level	Design Level
Classes Class	Instantiable abstractions, for further details see MOF (Meta-Object Facility). On conceptual level close to entity types of E/R diagrams.	X	X
Attributes Class Attribute	Classes can have attributes of certain types.	O	X
Operations (Methods) Class Operation(param)	Operations that can be performed by a class.	O[7]	X
Association	Representation of relationships. Optionally, roles can be annotated at the association's ends.	X	X
Aggregation	"has-a" relationship with weak linkage	X	X
Composition	"has-a" relationship with strong linkage	X	X

[7] In case that a class diagram on conceptual level is used for E/R-like purposes, it is most likely not useful to use operations.

© Copyright SAP AG 2007

Element	Brief Description	Conceptual Level	Design Level
Specialization	Visualization of “is-a” relationships between classes.	X	X
Navigability	Association ends can be navigable in either or both directions.	O	X
Packages Package	Packages indicate the membership of the shown classes.	O	X
Data types Class Attribute: String «dataType» Datatype	Preferably used for typing of attributes and operation parameters and return values. Alternatively, it can be visualized as a special classifier defining the data type.	O	X
Visibility Class + Attribute	An element’s visibility determines in which ways it can be accessed.	-	X
Interfaces «interface» Interface	Interfaces contain a number of declarations. These can be implemented by classifiers (e.g. class or component).	-	X
Interface Realization	Realization (Implementation) of an interface.	-	X
Templates Parameter Class	Parameterization of classifiers (usually classes).	-	X

© Copyright SAP AG 2007

Element	Brief Description	Conceptual Level	Design Level
Dependencies «use»	Use, Import, Instantiate	-	O
Association classes Association Class	Association that also has properties of a class.	O	O
N-ary Associations	Association with N ends (N > 2).	O	-
Multiple Specialization	Specialization from multiple base classes.	O	O[8]
Instances (Objects) Object: Class	Instances of classes showing structures at run-time.	-	O
Qualification	Qualified associations.	-	-

[8] Please note that it has to be possible to reflect modeled multiple inheritance in the according implementation by using the particular technology (e.g. the programming language or framework).

2.1.2.4 Example Diagrams

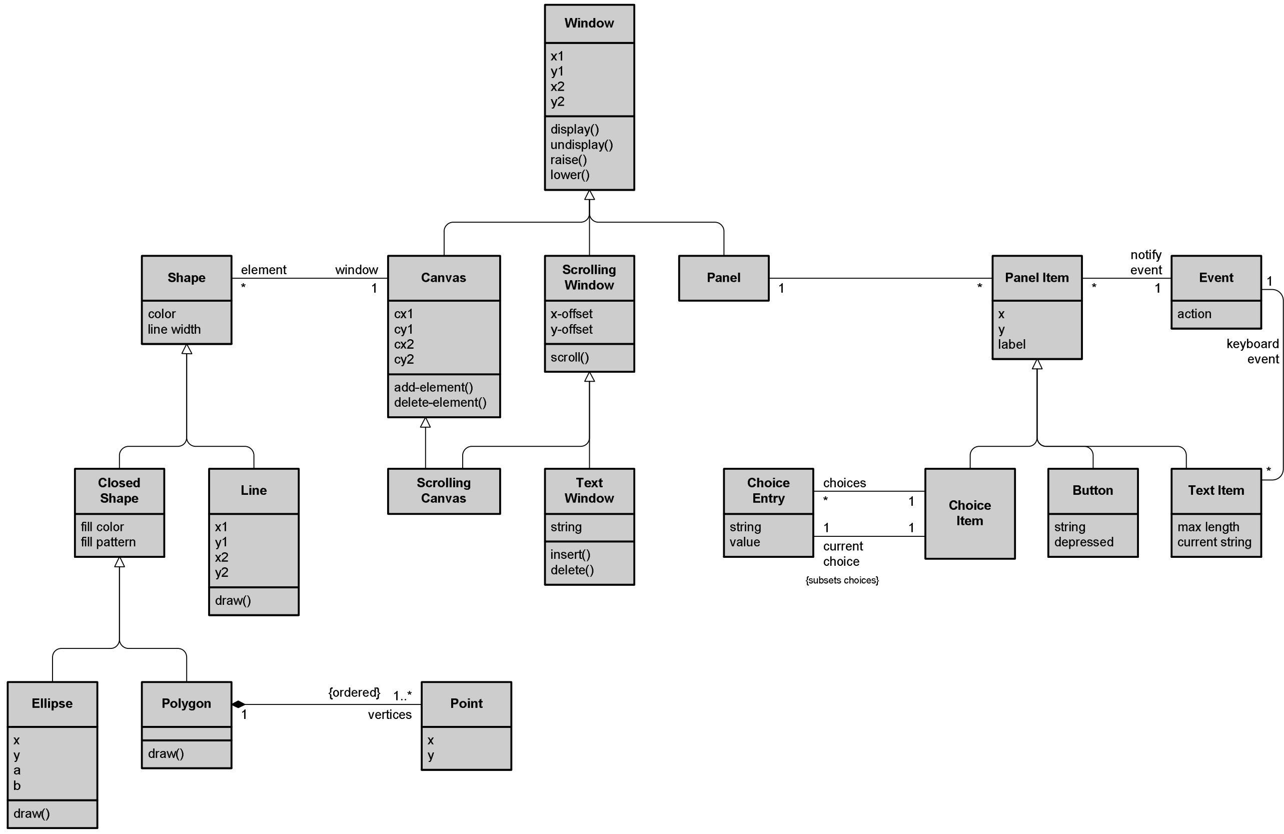

Figure 2–3 Class diagram Conceptual Level

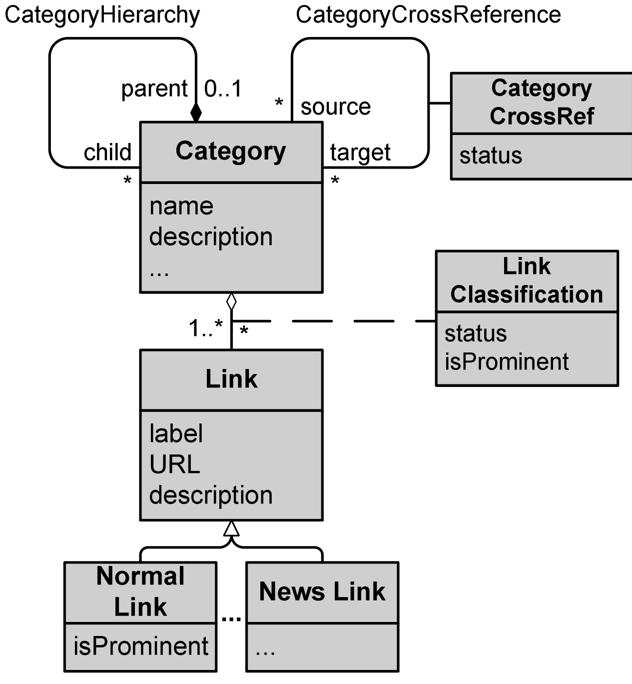

Figure 2–4 Class diagram Conceptual Level

© Copyright SAP AG 2007

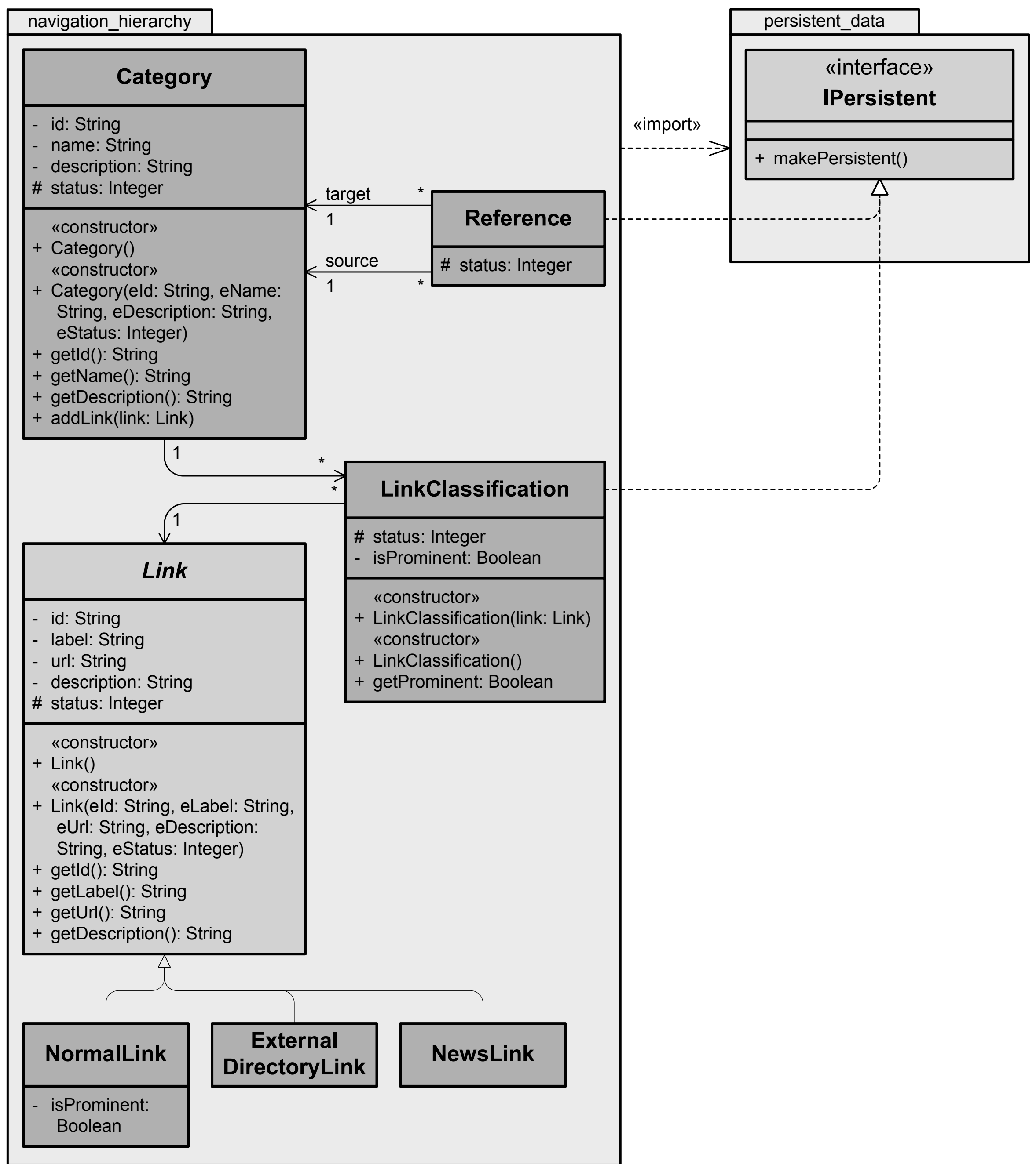

Figure 2–5 Class diagram Design Level

2.1.3 Package Diagram

2.1.3.1 Purpose

Package diagrams describe dependencies that exist between packages. Consequently, a general overview of visibility and containment of classes is provided.

Even though packages are already included in class diagrams, the context of package diagrams is slightly different. Whereas in class diagrams, packages are usually used in order to show assignments of classes to packages, package diagrams are used to visualize inter-package dependencies and modular aspects of the package structure.

2.1.3.2 Abstraction Levels

Package diagrams are available on design level only.

2.1.3.3 Diagram Elements

Element	Brief Description	Conceptual Level[9]	Design Level
Packages Package	Packages can contain classes, components, etc.	**n/a**	**X**
Dependencies «import»	For example merge, import	**n/a**	**X**

2.1.3.4 Example Diagrams

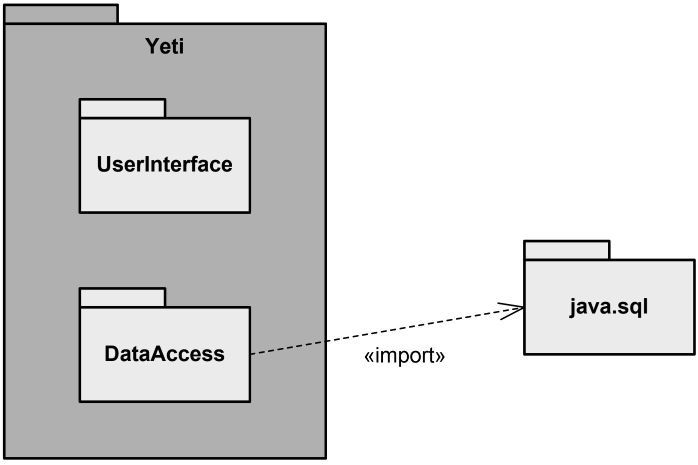

Figure 2–6 Package Diagram

[9] A column which is displayed with a grey background indicates that the respective level is not applicable for that diagram type. In addition all elements are marked as **n/a** indicating that these elements are not applicable for that level.

© Copyright SAP AG 2007

2.2 Diagram Types for Behavioral Descriptions

2.2.1 Use Case Diagram

2.2.1.1 Purpose

Use case diagrams help to specify the desired functionality of a system. Therefore, *use cases* (diagram elements) describe a certain behavior and *associations* describe which *actors* take part in those. They are not supposed to show detailed technical aspects.

Please notice, that it is important to distinguish between use cases and use case *diagrams*. The latter are only intended for structuring requirements or functionality graphically. They may also be used to identify, structure and visualize user interfaces of the system. For this, use case diagrams need a further detailed description. This can be accomplished according to a template which is not part of the standard. Typically, such a template covers the following aspects:

> **Business goal** - A short description of the interaction described in the use case and eventually its business context
>
> **Preconditions** - Must be fulfilled in order an interaction to be performed
>
> **Interaction scenario(s)** - A use case may result in different sequences of execution of communication steps by the involved actors. At least one scenario is to be defined: the happy day scenario. The most important alternatives should be described in separate scenarios.
>
> **Post-conditions** - State of the system after a successfully accomplished interaction is to be described in the post-conditions.

Additional sections might contain e.g. error handling or special requirements. The interaction scenarios may be a pure textual description, for more complex interactions, the use of sequence diagrams or activity diagrams on conceptual level is recommended.

2.2.1.2 Abstraction Levels

Use case diagrams are available on conceptual level only.

2.2.1.3 Diagram Elements

Element	Brief Description	Conceptual Level	Design Level
Use Cases Use Case	Representation of a desired functionality.	X	n/a
Actors Actor «actor» Actor	Actors participate in a particular Use Case (e.g. human actor, machine actor).	X	n/a

© Copyright SAP AG 2007

Element	Brief Description	Conceptual Level	Design Level
Include Dependency «include»	Containment of Use Cases.	**X**	**n/a**
Specialization	Inheritance of Use Cases and/or Actors	**O**	**n/a**
Extends Dependency «extends»	Extension of a Use Case under certain circumstances. Might be annotated with a condition.	**O**	**n/a**
Extension Points Use Case Extension Point	Coupling point for extensions in form of other Use Cases.	**O**	**n/a**

2.2.1.4 Example Diagrams

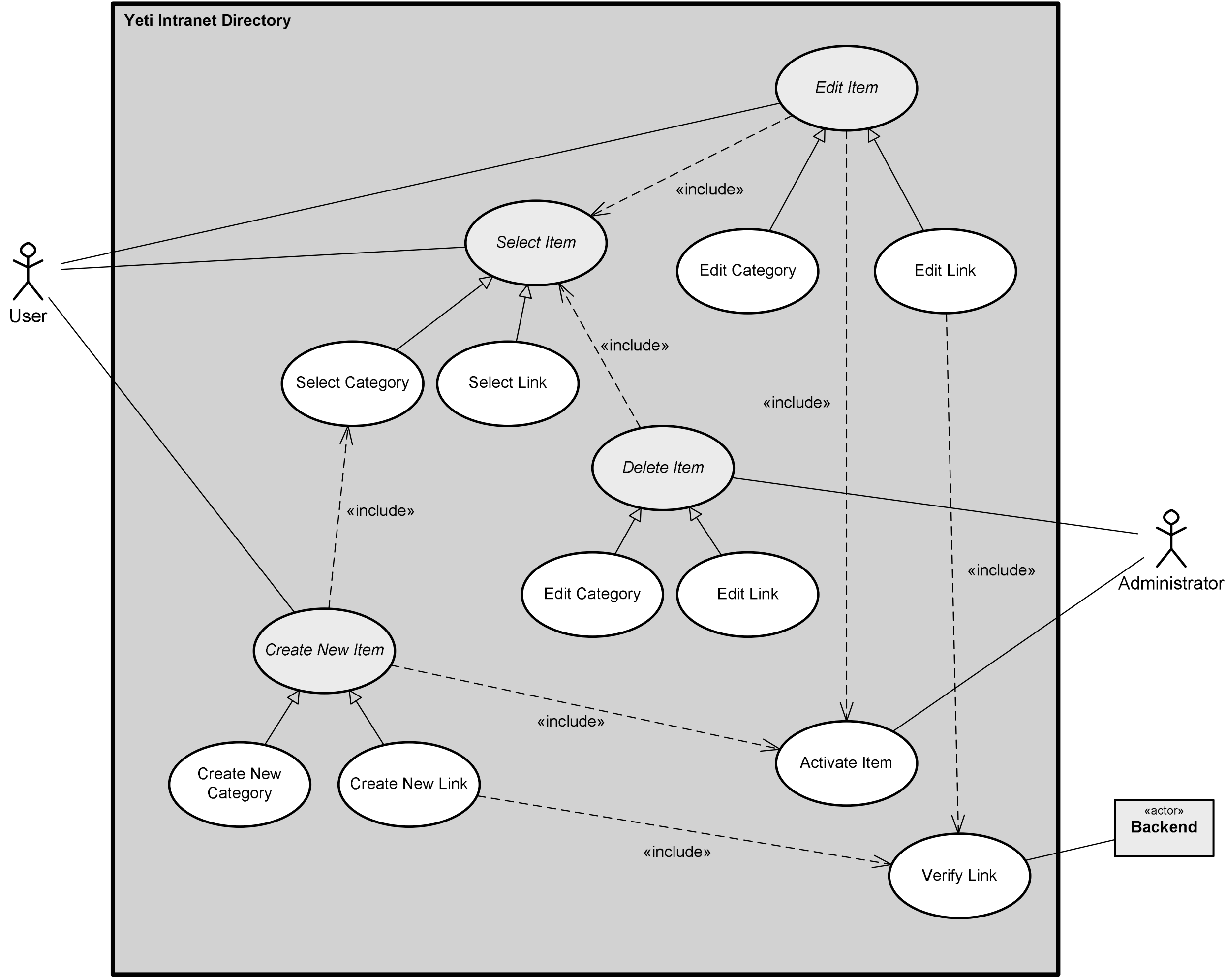

Figure 2–7 Use Case diagram

2.2.2 Activity Diagram

2.2.2.1 Purpose

Activity diagrams provide behavioral descriptions of a system. *Actions* are utilized to show certain units of operation. *Control Flow* (and optionally *Object Flow*) is visualized as connecting arcs between the *Actions*. Optionally, Activity diagrams can describe interactions of different entities by visualizing boundaries as swimlanes. Please notice that the whole sequence of actions (including *Initial* and *Activity-Final Node*) is called *Activity*.

2.2.2.2 Abstraction Levels

Activity diagrams are available on conceptual level and design level.

On conceptual level, activity diagrams describe more abstract views than those on design level. This means, that the whole diagram explains a process on quite a general level. Characteristically, *actions* are operations that describe a certain task without providing concrete implementation information.

On design level, more detailed information regarding the implementation is provided. Generally, each *action* should be mappable onto a certain piece of implementation (for example, procedure, function, loop, etc.).

2.2.2.3 Diagram Elements

Element	Brief Description	Conceptual Level	Design Level
Actions Action	Steps in execution	X	X
Control Flow	Connections between actions showing the order of execution and time dependencies	X	X
Initial / Activity-Final Node	Initialization and termination of an activity.	X	X
Decision / Merge	Show basic mechanisms of the control flow (if, loop, …)	X	X
Fork/Join	Parallel execution and synchronization points.	X	X
Object Flow ObjectNode	Exchange of objects between activities.	O	O
Flow-Final Node	Termination of a flow (object or control flow).	O	O
Swimlanes Partition	Partitioning the activity according to different contextual entities (e.g. objects, namespaces, and subsystems). Activities are unambiguously assigned to the particular entity they belong to.	O	O

Element	Brief Description	Conceptual Level	Design Level
Exception Handlers Exception Handler	On occurrence of an exception, the linked handler is supposed to maintain a proper functioning.	-	O
Interruptible Regions	Explicitly marked region of an activity that contains a number of actions. During any of the contained actions a certain interruption is possible. All tokens inside the region are removed in case of exception. Note: "InterruptibleRegion" was renamed to "InterruptibleActivityRegion" in UML v2.1.	-	O
Expansion Region	Separate region of an activity with explicit inputs and outputs.	-	-
Signals (Events)	Signals can be exchanged between objects.	-	-
Send/Receive Signal (Event)	Actions that are related to the transmission of signals.	-	-
States	obsolete (UML v1.x)	-	-

© Copyright SAP AG 2007

2.2.2.4 Example Diagrams

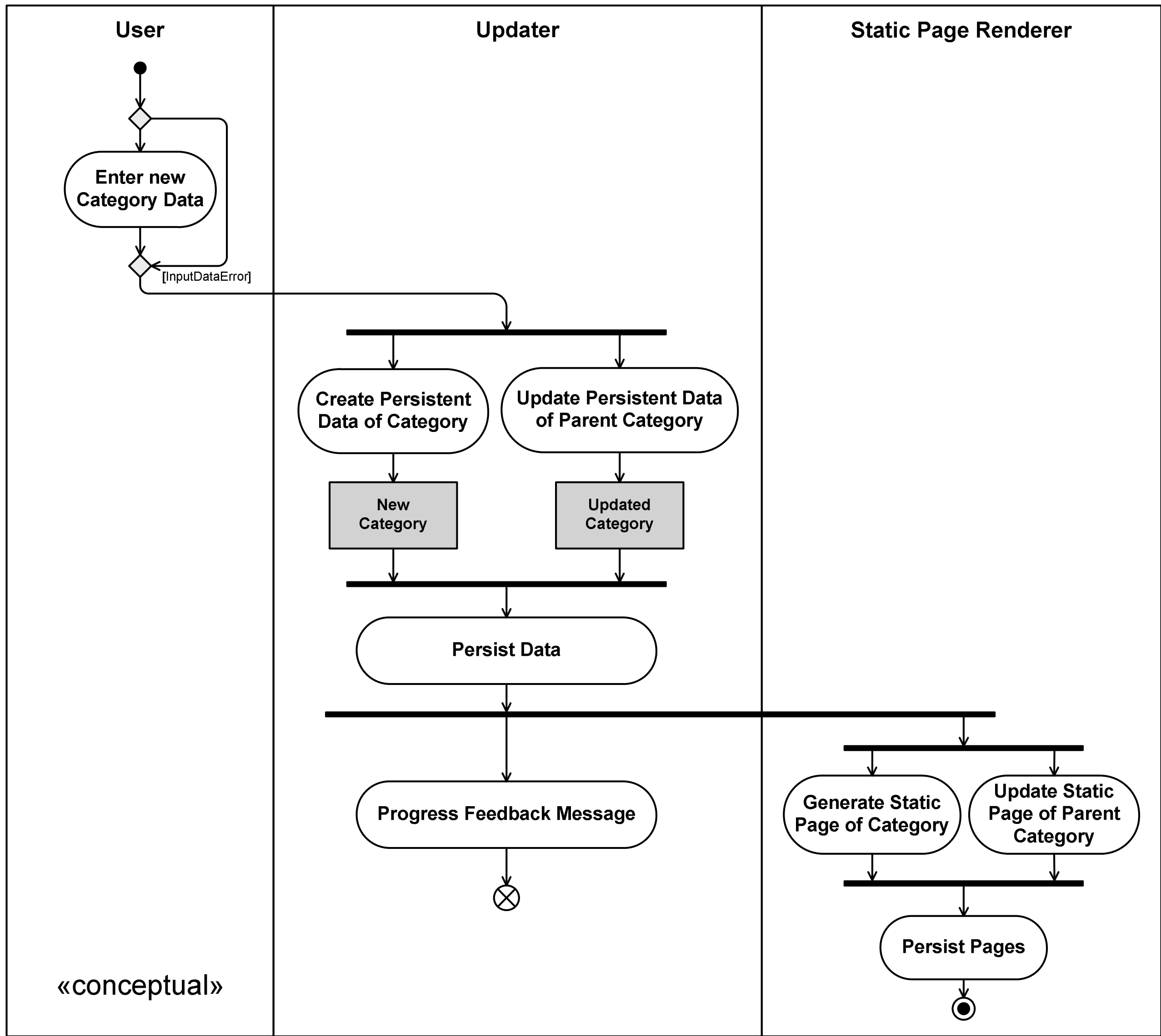

Figure 2–8 Activity Diagram Conceptual Level

2.2.3 Sequence Diagram

2.2.3.1 Purpose

Sequence diagrams are intended to show interactions between objects in chronological order (for example, system parts, classes, components). The central aspect covered is message exchange between the objects. Although UML defines all elements necessary for complete behavioral descriptions, it is recommended to concentrate on sequential flows without many conditions or parallel execution. Characteristically, the resulting diagrams are more exemplary than generic.

UML also introduces an alternative representation called *Interaction Overview Diagram*[10] as a hybrid of Activity diagrams and Sequence diagrams. In some cases this kind of diagrams can be useful in order to visualize the control flow combined with interactions of several entities.

Since not all modeling at SAP is based on object orientation, it is allowed that a lifeline does not necessarily correspond to an object in the sense of object orientation but also e.g. to an ABAP function module, a C function or even a web service. This implies that messages do not have to be method calls in any case, but may also be any other kind of message or call as long as this message is applicable for the kind of "object" used. It is, thus, necessary to describe the kind of objects in the text provided with the diagram, if it differs from objects as OO-objects.

2.2.3.2 Abstraction Levels

Sequence diagrams are available on conceptual level and design level.

On conceptual level, sequence diagrams describe interactions in rather general terms. Typically, the objects used on conceptual level correspond to agents modeled in a conceptual level component/block diagram.

On design level, messages are usually method or function calls between objects that do exist in real implementation. All modeled information should be mappable onto interactions in the real system or the system-to-be.

2.2.3.3 Diagram Elements

Element	Brief Description	Conceptual Level	Design Level
Lifelines :Lifeline	All entities participating in a particular interaction are objects. Each object owns exactly one lifeline. On conceptual level, also human interaction is allowed to be shown (based on human agents, which are specializations of the meta class *Component*) Part decomposition is also allowed.	X	X
Messages	Objects exchange messages (asynchronous, synchronous, return message). This also includes the special case of messages originating and arriving at the same lifeline.	X	X
Object Creation create	Objects are capable of creating new objects.	-	X

[10] These are treated as a subform of Sequence diagrams by the technical architecture modeling standard although they are formally defined as a specialization of Activity diagrams by the UML Superstructure.

Element	Brief Description	Conceptual Level	Design Level
Lifeline Termination	Destruction of an object.	-	X
Return Values Return Value	Synchronous calls usually require return values. On conceptual level, the return messages might be left out for reasons of simplicity.	O	X
Method Activation	Visualization of activation on a lifeline. Additionally, differently shaded activation without focus of control is allowed to be shown if the object is (actively) waiting for an answer of a synchronous call (see examples).[11]	O	O
Interaction Use ref Interaction	Referencing another diagram.	O	O
Combined Fragments opt	Combined Fragments allow certain regions with special properties (e.g. loop, alternative, option, parallel execution)	O	O
Timing/Duration constraints {duration constraint}	Define timing and duration constraints at certain moments. (Also includes deferred messages)	O	O
State Invariants {constraint}	Constraint that is evaluated at run time. It is placed on a lifeline and has the semantics of an invariant.	-	O
Continuations	Continuations of different branches.	-	-

[11] The notation variant differentiating between focus of control and without control was part of the first versions of UML, starting with 0.8. It is helpful in situations, where detailed communication with all returning messages is shown, because it then emphasizes the main thread of control in the overall diagram. However, it cannot be applied in all situations. Therefore it is optional.

Element	Brief Description	Conceptual Level	Design Level
Coregions	Coregions define a certain time interval, in which the temporal ordering of the arrival of messages is not of relevance (only the arrival at all).	-	-
1.x-style conditions & parallel processing	"Duplication" of lifelines. (UML 1.x) Obsolete, since UML version 2.0 introduced combined fragments as means for modeling conditions and parallel execution.	-	-

2.2.3.4 Example Diagrams

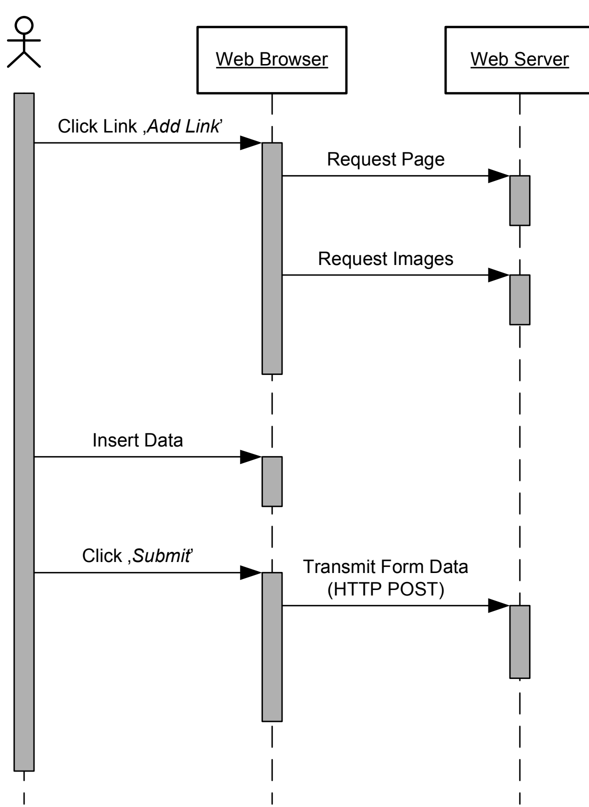

Figure 2–9 Sequence diagram Conceptual Level

© Copyright SAP AG 2007

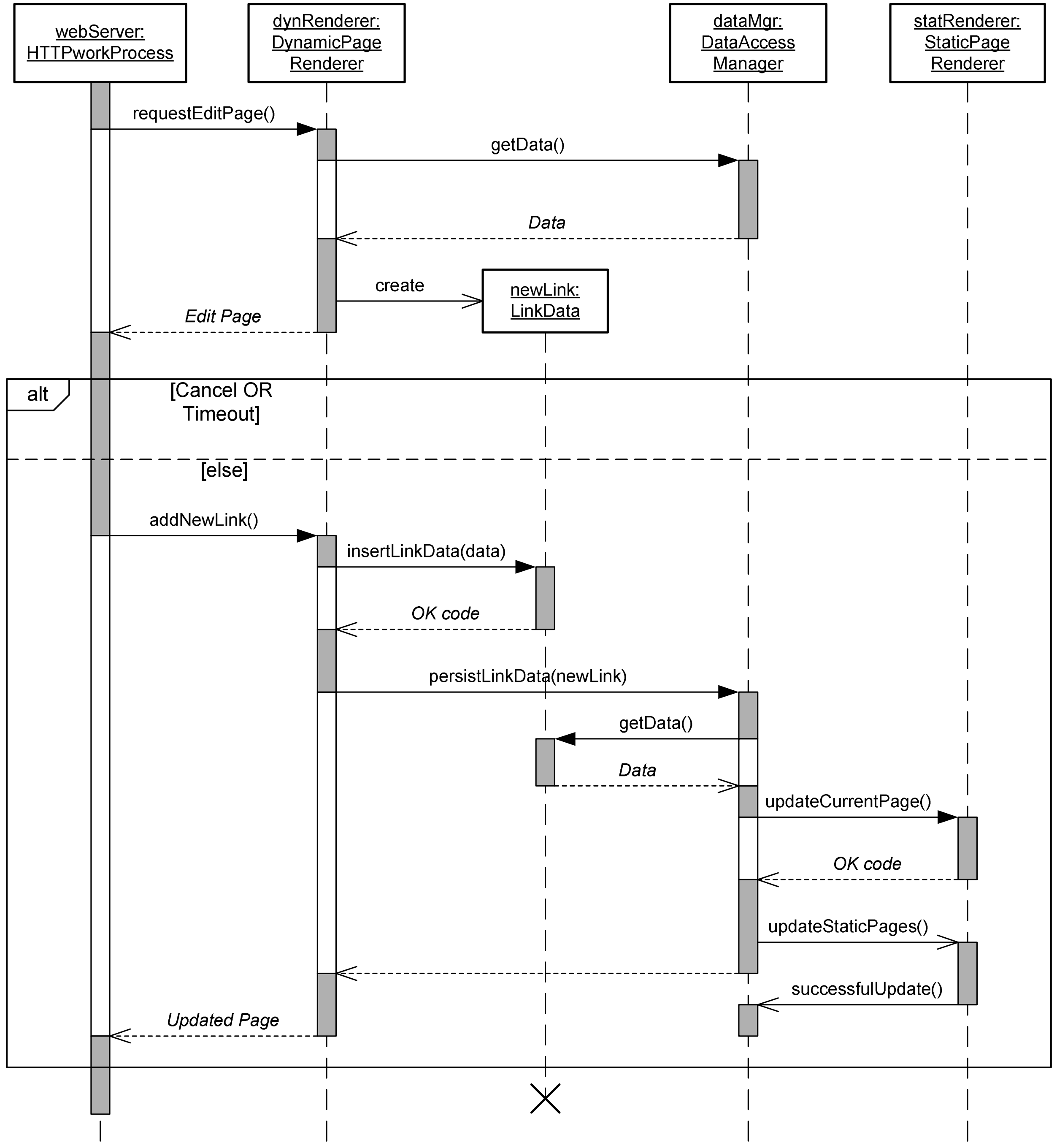

Figure 2–10 Sequence diagram Design Level

2.2.4 State Machine Diagram

2.2.4.1 Purpose

These diagrams semantically define state machines of objects, components, subsystems, etc. Therefore, states (possibly nested) and valid transitions between them are the main elements.

2.2.4.2 Abstraction Levels

State machine diagrams are available on conceptual level only.

2.2.4.3 Diagram Elements

Element	Brief Description	Conceptual Level	Design Level
States State	States the respective thing is in (invariant). (including composite states for structural reasons)	**X**	**n/a**
Transitions event[guard]/action	Transitions between states are triggered by an event. (*event[guard]/action*)	**X**	**n/a**
Initial/Final States	Initialization or termination of a State Machine diagram.	**X**	**n/a**
Submachine States ab : sm cd	Specification of a state by the help of another state machine.	**O**	**n/a**
Concurrent Regions State	Concurrency within a composite state.	**O**	**n/a**
Entry/Exit point State	Pseudo states used for entering/leaving a composite state.	**O**	**n/a**
Choice	Pseudo state indicating a decision.	**O**	**n/a**

© Copyright SAP AG 2007

Element	Brief Description	Conceptual Level	Design Level
Junction	Sequencing of transitions in order to e.g. define complex guard-conditions.	**-**	**n/a**
History States	States storing the configurations of composite states. (applying to deep and shallow history state)	**-**	**n/a**
Fork/Join	Splitting transitions into vertices to states in a number of concurrent regions.	**-**	**n/a**
Actions	Explicitly modeled actions in a State Machine diagram.	**-**	**n/a**
Receive/Send Signal (Event)	Actions that are related to the transmission of signals.	**-**	**n/a**

2.2.4.4 Example Diagrams

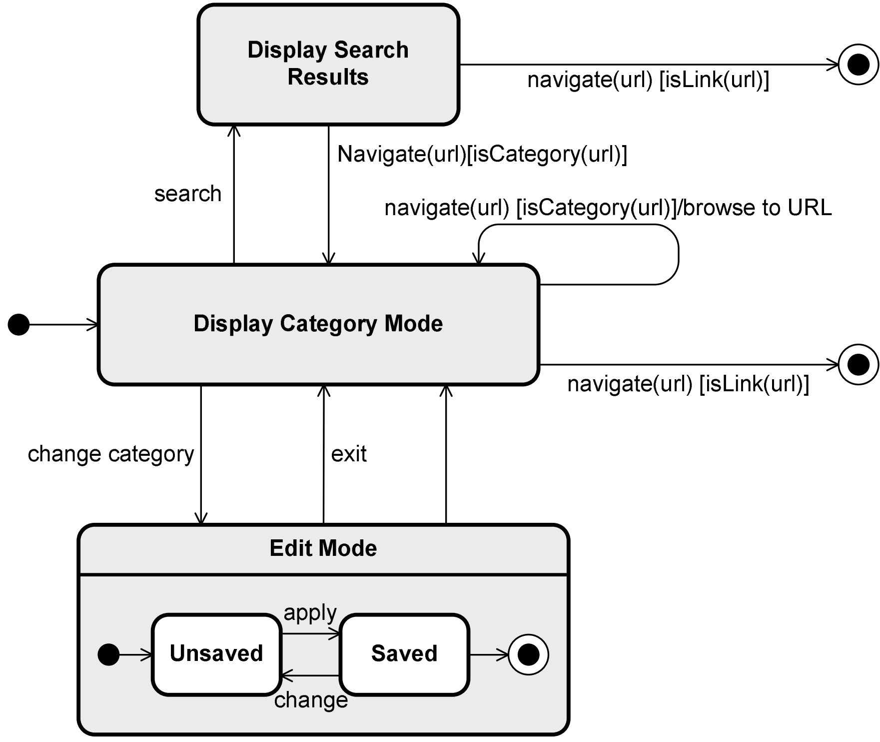

Figure 2–11 State Machine Diagram

2.3 UML Profiles

Profiles are a lightweight extension mechanism to the UML meta model. The SAP standardized technical architecture modeling defines, which profiles are allowed to be used. No profiles except for the ones explicitly listed below are intended to be used.

Presently, TAM does not contain nor refer to any profiles. If necessary, existing profiles might be included into this document as references later. Additional profiles might also be created and included into the standard during an update process.

© Copyright SAP AG 2007

3 Appendix A: Component/Block Diagram

3.1 Overview

Describing the architecture of a software system on conceptual level requires the modeling method to enable the user to express different facets of the system. However, it is of essential necessity that the resulting diagrams do not become too complex in order to be understood quickly. The diagrams should serve their purpose of covering all aspects that are particularly interesting for the addressees of the diagram with a maximum of understandability and readability. Usually, this kind of rather abstract diagrams is utilized for showing the general structure of a system including the embedding in its environment. Detailed implementation decisions are too concrete for this abstraction level. This correlates with the circumstance that these diagrams are mostly intended to be used in the early project phases and of course for Knowledge Transfer purposes. But still, it is of course desired to be able to link these diagrams to others that are more focused on implementation aspects in course of the development phase.

Along with two other diagram types manifested in FMC (Fundamental Modeling Concepts)[12] they were especially developed for serving exactly the purpose of modeling systems on a high level and, therefore, improving (human) communication of those systems. Block diagrams help standardizing and formalizing diagrams and, thus, help to avoid ad-hoc or freestyle notation.

Besides, the Unified Modeling Language (UML) continuously became more and more popular all over the world. Today, it can be seen as the de-facto standard. Nevertheless, it lacks the facility to effectively model the aforementioned aspects of abstract system structures, their environment and so forth. This certainly results from its primary targeting at aspects rather close to implementation. This again can be seen as a consequence of the evolutionary history of UML.

UML diagram types form the basis for the standardized technical architecture modeling although Block diagrams are the most suitable means for modeling structural aspects of a system's architecture. Therefore, it is desired to integrate Block diagrams into UML enabling the best possible linkage to other diagram types and, thus, provide a possibility to formally refine Block diagrams (on design level) towards implementation.

This is achieved by defining an extension to UML on level of the meta model. Component diagrams pose a diagram type, which is appropriate to describe composite structures in detail. The means provided can be used in order to show structures that are close to implementation. By specializing a number of elements taken from the Component diagrams and applying syntax and semantics of Block diagram elements it is possible to completely integrate Block diagrams into UML. It is even possible to explicitly define areas of special interest for implementation by using UML subsystems, components, ports etc. only in particular regions of one diagram. In doing so, a number of nesting rules have to be considered.

This chapter formally describes all changes to the meta model in detail (similar to the UML Superstructure). The introduced package *BasicBlockElements* contains all these changes and is merged into the according native packages.

---

[12] The sound basis of the development of FMC was established in the 1970s where Siegfriend Wendt firstly initiated research activities related to what is known as FMC today. The methodology, the notation as well as the name evolved over the years. So, FMC was formerly labeled as *SPIKES* (**S**tructured **P**lans for **I**mproving **K**nowledge Transfer in **E**ngineering of **S**ystems) for a while.

© Copyright SAP AG 2007

3.2 Abstract Syntax

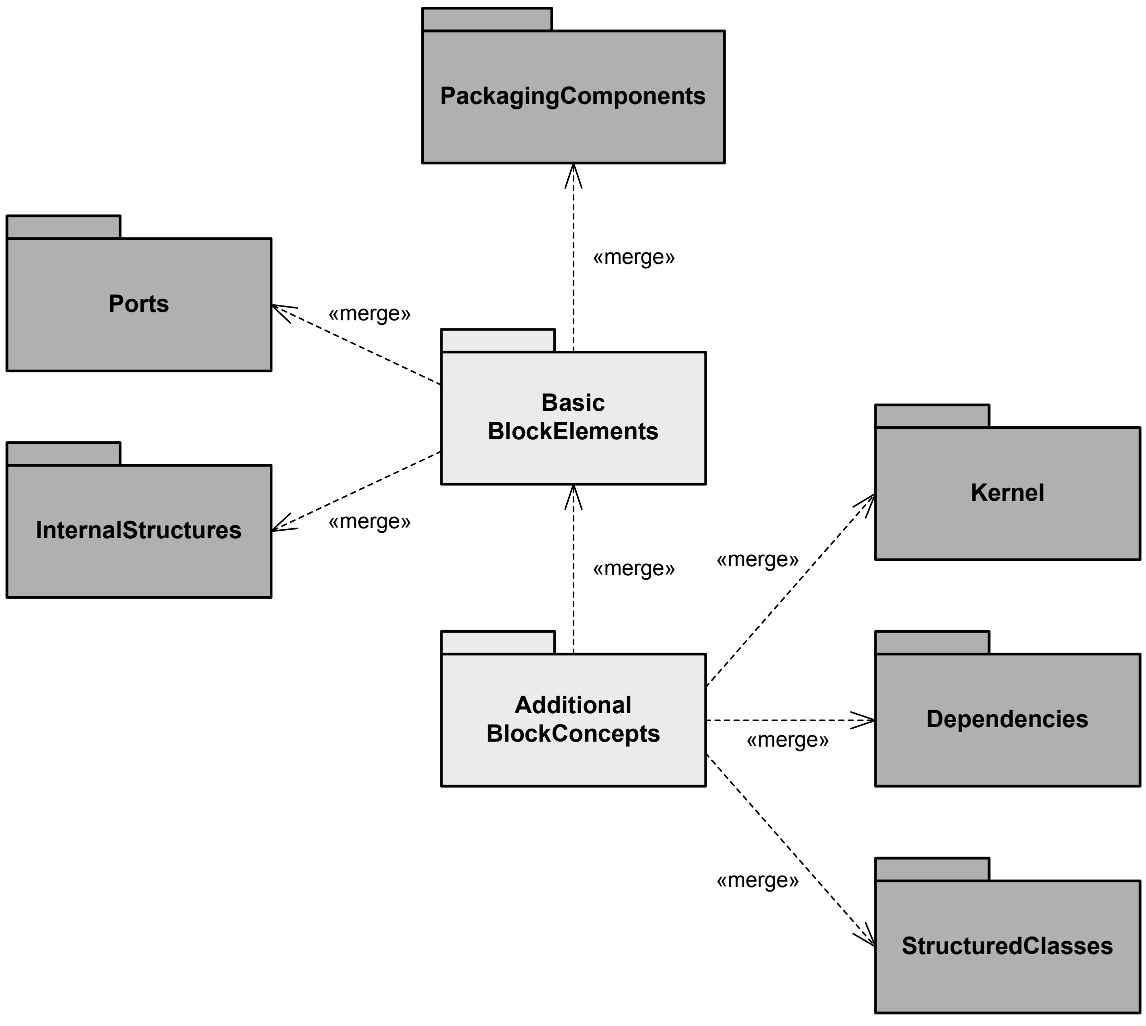

Figure 3–1 Dependencies between packages described in this chapter[13]

[13] Please note that transitive dependencies are not shown.

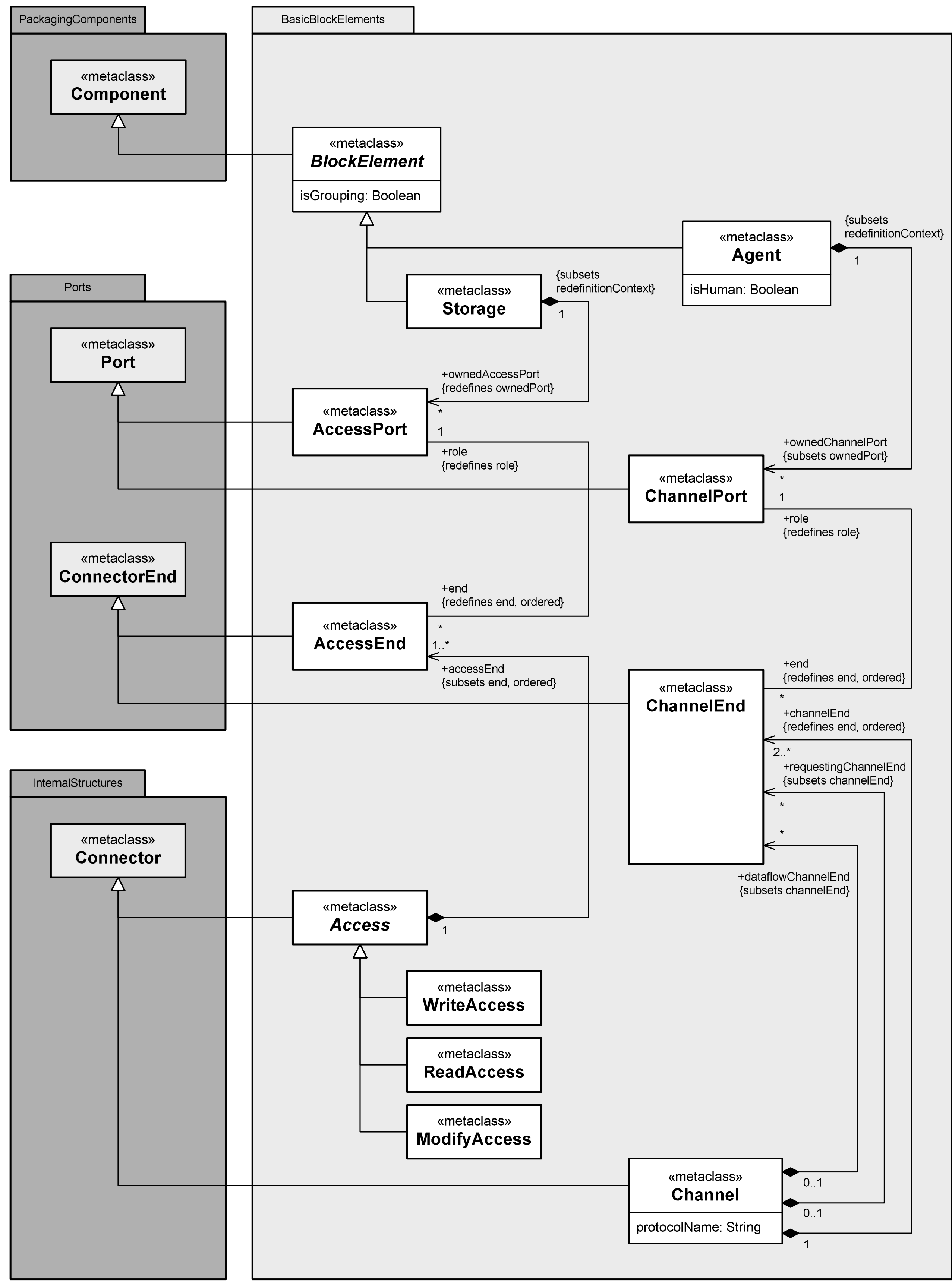

Figure 3–2 The metaclasses that define the basic Block diagram elements

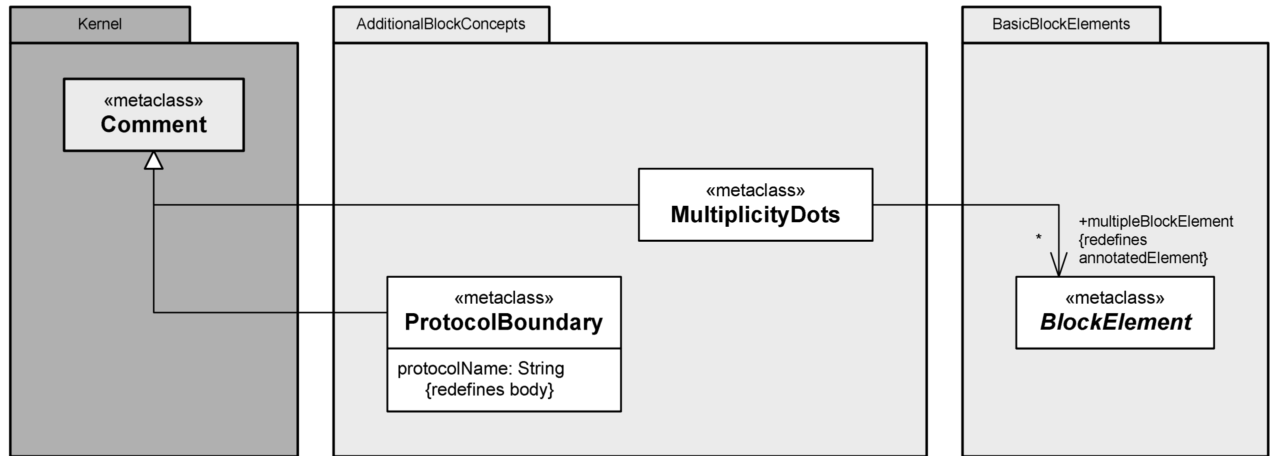

Figure 3–3 Additional Block diagram concepts used for didactical purposes

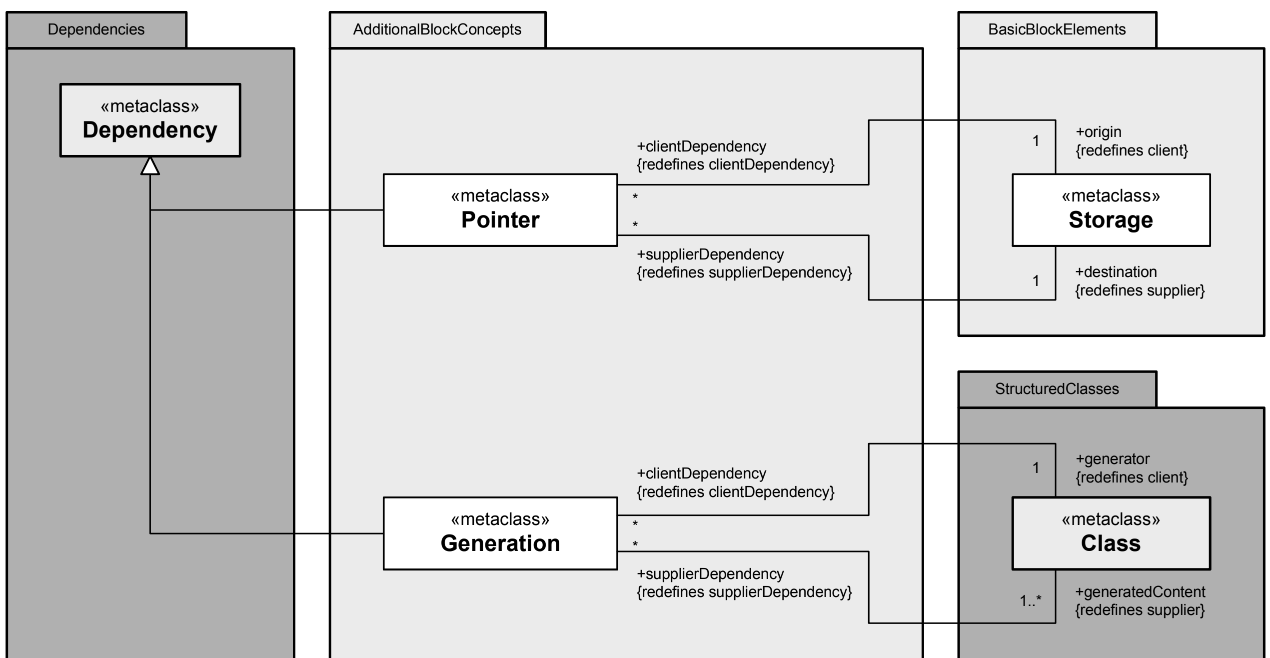

Figure 3–4 Additional Block diagram concepts describing special dependencies

© Copyright SAP AG 2007

3.3 Class Descriptions

3.3.1 Access (from BasicBlockElements)

Generalizations

- “Connector (from InternalStructures)” in UML Superstructure

Description

Access is an abstract meta class derived from InternalStructures::Connector. Concrete accesses are modeled by subclasses of access.

Formally, an Access describes a contract between storages and an active component covering the possible access mode of data contained in the storage.

Attributes

No additional attributes

Associations

- accessEnd: AccessEnd [1..*]
  Each access has to have at least one access end. These have to end at different access ports. The set of access ends is ordered. (Subsets *Connector.end*)

Constraints

[1] If the inherited attribute *kind* (of type ConnectorKind) is set to the enum-value *assembly*, an access must only be defined between exactly one active component (agent, component, subsystem, class) and one (or more) storages (respectively to one of their access ports).

[2] If the inherited attribute *kind* (of type ConnectorKind) is set to the enum-value *delegation*, an access is the direct continuation of the access arriving at the outside of the particular access port. Thus, it has to delegate all arriving signals to a nested storage (respectively to one of its access ports).

[3] Each Access has at least one AccessEnd on side of the storage(s) and a common ConnectorEnd on side of the active component..

Semantics

An access is used to show a *possible* access of an active component (e.g. agent, component) on a storage.

Write accesses define that an active component is able to write the contents of a storage but not to read it.

Read accesses define that an active component is able to read the contents of a storage but not to write it.

Modifying accesses define that an active component is able to read and write the contents of a storage (in one step).

Notation

Accesses are drawn as directed edges. Their direction determines what kind of access is being used.

An edge pointing from an active component (e.g. agent, component, subsystem, class) to a storage is a *write access*.

An edge pointing from a storage to an active component is a *read access*.

Two edges pointing in one direction each (usually in shape of a semicircle) is a *modifying access*.

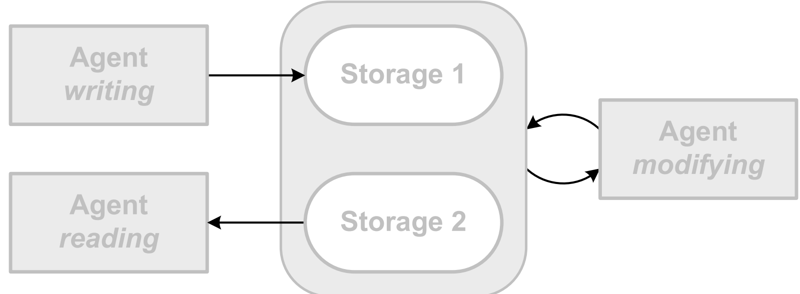

Figure 3–5 Notation of agents accessing storages

The figure shows a graphical simplification of assembly and delegation accesses to the nested storages 1 & 2. Formally, there are two accesses between the agent writing and storage 1. They are graphically simplified into one arc.

The following figure shows two equivalent ways of accessing nested storages. Please note that the dotted access port shown is usually invisible. Of course, this simplification rule can also be applied cascaded.

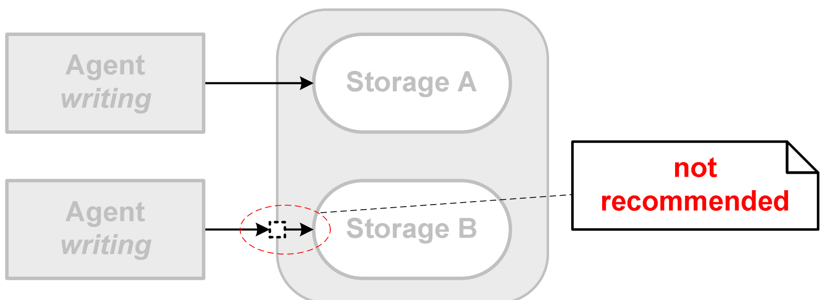

Figure 3–6 Graphical simplification of assembly and delegation accesses

3.3.2 AccessEnd (from BasicBlockElements)

Generalizations

- “ConnectorEnd (from Ports)” in UML Superstructure

Description

In the meta model, an AccessEnd is a subtype of Ports::ConnectorEnd. It is the special end of an access on side of the storage.

Attributes

No additional attributes

Associations

- role: AccessPort [1]
  Each access end has to be attached to an access port. (Redefines *ConnectorEnd.role*)

Constraints

[1] An access end always belongs to an access.

Semantics

An access end realizes the attachment of an access to a storage. Each access end actually ends at an access port (not at a usual port) owned by the storage.

Access ends are not visualized in a diagram.

Notation

Access ends are invisible.

3.3.3 AccessPort (from BasicBlockElements)

Generalizations

- “Port (from Ports)” in UML Superstructure

Description

As specialization of Ports::Port, each AccessPort belongs to a Storage and each Storage can own multiple AccessPorts (but no usual Ports). All kinds of information exchange has to be accomplished by utilizing AccessPorts.

It is helpful to imagine that trivial access routines are provided by the storage. Interfaces (not explicitly shown) are exposed in order to access the storage using these basic operations.

Attributes

No additional attributes

Associations

- end: AccessEnd [*]
  The access ends attached to the access port. (Redefines *ConnectableElement.end*)

Constraints

[1] An access port can only be owned by storages, but *not* by agents, components, subsystems, classes.

[2] The inherited attribute *isBehavior* (of type Boolean) is set to *false*.

Semantics

An access port is a special interaction point allowing a storage to be accessed by active components. All altering or reading of a storage is accomplished using access ports. Thus, only storages are allowed to own access ports.

Access ports are not visualized in a diagram.

Notation

Access ports are invisible.

3.3.4 Agent (from BasicBlockElements)

Generalizations

- “BlockElement (from BasicBlockElements)” on page 35

Description

Agents are BlockElements. They can either be connected to agents via channels, to components via interfaces or to storages via accesses. They are allowed to contain agents, subsystems, components, classes and storages.

Attributes

- isHuman: Boolean
  If *true*, indicates that the agent represents a human rather than a technical component. If *false*, the agent can be anything different from a human as long as it is active (e.g. hardware or software).

Associations

© Copyright SAP AG 2007

- ownedChannelPort: ChannelPort [*]
  References a set of channel ports that an agent owns. (Subsets *EncapsulatedClassifier.ownedPort*)

Constraints

[1] If an agent is human, it is not allowed to contain any elements (structural elements).

Semantics

Agents are active components of a system and usually communicating with each other using channels and/or accessing storages. Agents represent parts of a system on a conceptual level. It is possible to nest agents within other agents or that storages are nested within agents. If an outer agent accesses a storage or communicates via a communication channel, this means that possibly all contained agents can use this channel and/or access.

An agent can be concretized regarding implementation by refining an agent using subsystems, components and/or classes.

It is also possible to show anonymous agents which are merely a grouping of agents. In combination with accesses and/or channels, groupings are used as a graphical simplification. The figure below shows Agent A & B both accessing the storage but only Agent A communicating with Agent 1 (and Agent B with Agent 2).

Even though classes (with only the name compartment) and agents have the same notation, the context clarifies whether it is a class or an agent.

Notation

Agents are shown as rectangular elements. Their name is placed inside the rectangle. The fill color of the rectangle may vary in order to serve didactical or specific semantical purposes.

If a human agent is shown, a stickman is shown inside the rectangle.

A grouping agent (anonymous) usually has a thinner line than the others.

Agents can also have L-/U-/T-shape.

Figure 3–7 Example showing agents

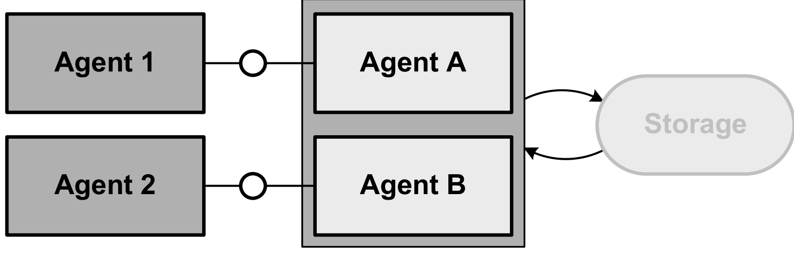

Figure 3–8 Grouped agents accessing a storage

3.3.5 BlockElement (from BasicBlockElements)

Generalizations

© Copyright SAP AG 2007

- “Component (from PackagingComponents)” in UML Superstructure

Description

BlockElement is an abstract meta class generalizing the standard block diagram elements.

Concrete nesting rules are defined in the according specializations of BlockElement. Regardless of the specialized BlockElement, it is disallowed to place them overlapping in a diagram. All elements have to be placed non-overlapping.

Attributes

- isGrouping: Boolean
  If *true*, indicates that the block element is a anonymous grouping of other block elements. See the notation section for slightly different notation in this case. If *false*, the block element, usually, carries a name and a semantical meaning within the diagram. Please note that it can also contain block elements in this case.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Grouped block elements are usually introduced by the modeler for didactical purposes. They help reducing the number of accesses and channels and can help to clarify a common context of block elements.

Block elements can be shown as multiple elements of the same kind. This can either be accomplished by staggering two agents or storages and placing two dots in the corner or, alternatively, by using three dots in between the elements of the same kind.

Further definitions on semantics are given for the concrete subclasses.

3.3.6 Channel (from BasicBlockElements)

Generalizations

- “Connector (from InternalStructures)” in UML Superstructure

Description

Similarly to an access, the Channel is derived from InternalStructures::Connector. It formally describes a contract between two agents.

Attributes

- protocolName: String
  Information about which protocol is being used or what kind of information passes the channel.

Associations

- channelEnd: ChannelEnd [2..*]
  A channel consists of at least two channel ends, each representing the participation of active components in communication of any kind. The set of channel ends is ordered. (Redefines *Connector.end*)

- requestingChannelEnd: ChannelEnd [*]
  A special channel end indicating that the active component to which it is attached has a requesting position during communication. (Subsets *Channel.channelEnd*)

- dataflowChannelEnd: ChannelEnd [*]
  A special channel end indicating that the active component to which it is attached has a position

receiving a noteworthy amount of data during communication (data flow). (Subsets *Channel.channelEnd*)

Constraints

[1] A channel must only be defined between agents.

[2] The number of dataflow channel ends indicating the data flow has to be smaller than the total number of channel ends and has to be non-ambiguous.

[3] The inherited attribute *kind* (of type ConnectorKind) is set to the enum-value *assembly*.

Semantics

Communication channels are used for any kind of communication between agents. Formally, they also define (a rather general) pair of required/provided interface. All information that is exchanged, however, can be of any kind. The channel can be annotated with the particular protocol used or other information regarding the communication.

A channel can have requests being sent into one or both directions. Furthermore, the dataflow direction can also be defined.

Channels can only be used between agents (not components), because the communication is rather abstract and not concrete enough for implementation details. When modeling parts of a diagram close to implementation using components, these have to be fully specified using provided/required interfaces.

Please note that communication channels as defined in the FMC meta model are formally *locations* and, thus, have a certain similarity to storages. In this extension of the UML meta model, channels are seen as *connectors*. However, this difference in definition does not imply a difference in interpretation of this diagram element.

Notation

Channels are circles with a white fill (by default). They are connected to agents using arcs.

Optionally, the request direction can be denoted by adding a "*R*" with a black triangle pointing in the same direction as the requests are transmitted. It is also possible that requests are sent in both directions. In this case, one "*R*" and two triangles are drawn.

For channels, a graphical simplification similar as for accesses is allowed.

Additionally, the dataflow direction can be shown by adding arrow heads pointing into the same direction.

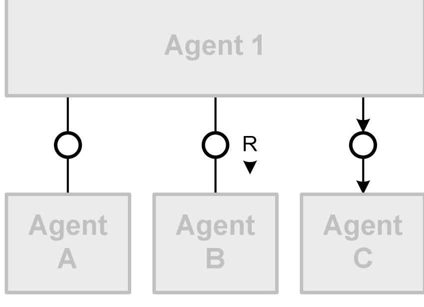

Figure 3–9 Variations of channels

3.3.7 ChannelEnd (from BasicBlockElements)

Generalizations

- "ConnectorEnd (from Ports)" in UML Superstructure

Description

In the meta model, a ChannelEnd is a subtype of Ports::ConnectorEnd. It is the special connector end on both ends of the channel.

Attributes

No additional attributes

Associations

- role: AccessPort [1]
  Each channel end has to be attached to a channel port. (Redefines *ConnectorEnd.role*)

Constraints

[1] A channel end always belongs to a channel.

Semantics

A channel end realizes the attachment of channels to an agent. Each channel end actually ends at a channel port (not at a usual port) owned by the agent.

Channel ends are not visualized in a diagram.

Notation

Channel ends are invisible.

3.3.8 ChannelPort (from BasicBlockElements)

Generalizations

- "Port (from Ports)" in UML Superstructure

Description

A ChannelPort is a specialization of Ports::Port. Each ChannelPort belongs to an Agent and each Agent can own multiple ChannelPorts (additionally to normal Ports). A ChannelPort can only be owned by Agents, because channels are allowed to be used between Agents only.

Attributes

No additional attributes

Associations

- end: ChannelEnd [*]
  The channel ends attached to the channel port. (Redefines *ConnectableElement.end*)

Constraints

[1] A channel port can only be owned by agents, but *not* by, storages components, subsystems, classes.

[2] The inherited attribute *isBehavior* (of type Boolean) is set to *false.*

Semantics

A channel port is a special interaction point allowing an agent to communicate with another agent.

Channel ports are not visualized in a diagram.

Notation

Channel ports are invisible.

3.3.9 Generation (from AdditionalBlockConcepts)

Generalizations

- "Dependency (from Dependencies)" in UML Superstructure

© Copyright SAP AG 2007

Description

A Generation is a special kind of Dependency.

Attributes

No additional attributes

Associations

- generator: Storage [1]
  The active component generating the new active component. (Redefines *Dependency.client*)
- generatedContent: Storage [1..*]
  The active component being generated. (Redefines *Dependency.supplier*)

Constraints

[1] The generator as well as the generatedContent is not allowed to be a Storage.

Semantics

A component might be able to generate a new active component. This can e.g. be accomplished by interpreting information from a meta data repository.[14]

Notation

A generation dependency is shown like a normal dependency (dashed line with open arrow head) with the keyword *«generate»*.

3.3.10 ModifyAccess (from BasicBlockElements)

Generalizations

- “Access (from BasicBlockElements)” on page 32

Description

Concrete specialization of Access.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See Access (from BasicBlockElements)

Notation

See Access (from BasicBlockElements)

3.3.11 MultiplicityDots (from AdditionalBlockConcepts)

Generalizations

[14] This generation of active content is in a way comparable to structure variances in FMC. The generation dependency was introduced as a lightweight substitute to the quite quite complex structure variances.

- “Comment (from Kernel)” in UML Superstructure

Description

MultiplicityDots are an additional information which is, therefore, manifested as a special comment.

Please note that these dots are not formally defined in FMC but, however, they are in frequent use.

Attributes

No additional attributes

Associations

- multipleBlockElement: BlockElement [*]
  The concrete block diagram elements, the multiplicity dots refer to. (Redefines *Comment.annotatedElement*)

Constraints

[1] Multiplicity dots must refer to block diagram elements of the same kind (either only agents or storages).

[2] The inherited property *body* has to be empty.

Semantics

Multiplicity dots can visualize multiplicity in the existence of block diagram elements of the same kind.[15] If there are e.g. channels attached to a multiple agent, this channel usually is also multiplied (see example).

The dots are only allowed to show multiplicity of block diagram elements, but not for subsystems, components, classes, etc.

Notation

Three black dots in between the block diagram elements, which should be shown as multiply existent, are shown.

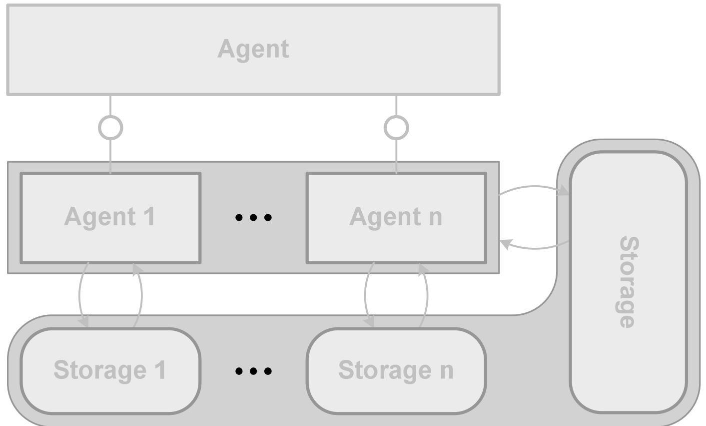

Figure 3–10 Usage of multiplicity dots

3.3.12 Pointer (from AdditionalBlockConcepts)

Generalizations

[15] The dots mainly intend to show that multiplicity itself occurs but do not primarily aim at describing this multiplicity in concrete numbers. In the sense of UML, these dots usually represent a concrete multiplicity of 1..*. However, it is possible to add digits to the names of the block diagram elements (see example) and express a more precise multiplicity.

- “Dependency (from Dependencies)” in UML Superstructure

Description

A Pointer is a special kind of Dependency.

Attributes

No additional attributes

Associations

- origin: Storage [1]
  The storage referencing another. (Redefines *Dependency.client*)
- destination: Storage [1]
  The storage being referenced. (Redefines *Dependency.supplier*)

Constraints

No additional constraints

Semantics

A pointer allows the visualization of a storage referencing the contents of another.

Notation

A pointer dependency is shown like a normal dependency (dashed line with open arrow head) with the keyword *«pointer»*.

3.3.13 ProtocolBoundary (from AdditionalBlockConcepts)

Generalizations

- “Comment (from Kernel)” in UML Superstructure

Description

A ProtocolBoundary adds additional information about the operation of communication channels to the diagram. Because of this, it is manifested as a special comment.

Attributes

- protocolName: String
  Information about which protocol is being used or what kind of information passes the channels on the boundary. (Redefines *Comment.body*)

Associations

No additional associations

Constraints

No additional constraints

Semantics

Protocol boundaries impose a means for visualizing the common protocol that is used by a number of communication channels.

Often, protocol boundaries are also used to separate elements into different logical regions. In this case, no protocol name is annotated. However, it is important to notice that the protocol boundary does not add any semantics to a diagram (e.g. containment or grouping information). Therefore, it is not a substitute for e.g. a surrounding agent shown around a number of elements.

Notation

A protocol boundary is shown as a dashed line lying under a set of communication channels. Additionally, the protocol's name is annotated.

Figure 3–11 Use of a protocol boundary

3.3.14 ReadAccess (from BasicBlockElements)

Generalizations

- "Access (from BasicBlockElements)" on page 32

Description

Concrete specialization of Access.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See Access (from BasicBlockElements)

Notation

See Access (from BasicBlockElements)

3.3.15 Storage (from BasicBlockElements)

Generalizations

- "BlockElement (from BasicBlockElements)" on page 35

Description

Storages are BlockElements. Each Storage can own multiple AccessPorts (but no usual Ports) through which it can be accessed by active components.

Attributes

No additional attributes

Associations

- ownedAccessPort: AccessPort [*]
  References a set of access ports that a storage owns. (Redefines *EncapsulatedClassifier.ownedPort*)

Constraints

[1] A storage is only allowed to own access ports.

[2] All communication has to be accomplished via accesses and access ports.

[3] Storages are not allowed to contain behavior of any kind. Thus, definition of methods, etc. is prohibited.

[4] Storages are not allowed to contain active components of any kind (e.g. agents, components, subsystems, classes). Only other storages can be nested.

Semantics

Storages are passive system components and are used to store any kind of information (volatile or persistently). They are not able to contain behavior except some trivial access routines used for accessing the storage's contents. They can only be accessed via access ports and attached access ends.

Storages can be accessed by all kinds of active system components (e.g. subsystems, components, classes).

Similarly to agents, a grouping storage (anonymous) can be used in order to reduce the number of accesses.

Notation

A storage is a round node. Its name is located inside. The fill color of the storage may vary in order to serve didactical or specific semantical purposes.

A grouping storage (anonymous) usually has a thinner line than the contained ones.

Storages can also have L-/U-/T-shape.

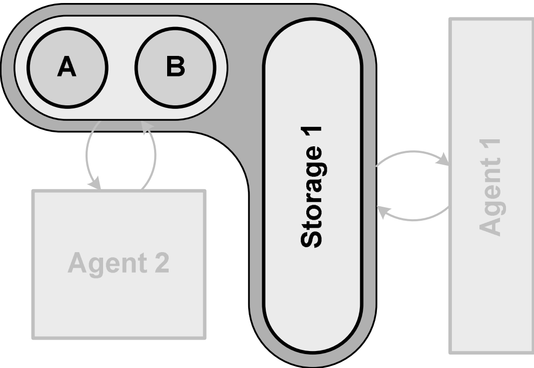

Figure 3–12 Nesting of storages

3.3.16 WriteAccess (from BasicBlockElements)

Generalizations

- "Access (from BasicBlockElements)" on page 32

Description

Concrete specialization of Access.

Attributes

No additional attributes

© Copyright SAP AG 2007

Associations

No additional associations

Constraints

No additional constraints

Semantics

See Access (from BasicBlockElements)

Notation

See Access (from BasicBlockElements)

4 Links & Literature

UML 2.0 Information by Object Management Group: http://www.uml.org/#UML2.0

UML 2.0 Specification (Superstructure) http://www.omg.org/cgi-bin/doc?formal/05-07-04

Andreas Knoepfel, Bernhard Groene, Peter Tabeling: Fundamental Modeling Concepts – Effective Communication of IT Systems. Wiley 2006

Fundamental Modeling Concepts (FMC) Homepage: http://www.fmc-modeling.org