An OMG® Structured Patterns Metamodel Standard™ Publication

Structured Patterns Metamodel Standard™

V1.2 with change bars

OMG Document Number: formal/2017-11-02

Release Date: December 2017

Standard document URL: http://www.omg.org/spec/SPMS/1.2

Normative Machine Consumable File(s):

http://www.omg.org/spec/SPMS/20170601/SPMS.xmi

http://www.omg.org/spec/SPMS/20160301/PHORML.xml

http://www.omg.org/spec/SPMS/20160301/APML.xmi

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: http://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

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

Table of Contents

1 Scope....1
2 Conformance....2
3 References....2
4 Terms and Definitions....2
5 Symbols....3
6 Additional Information....3
6.1 Acknowledgments....3
7 SPMS Overview (Informative)....5
8 Definitions Classes....9
8.1 Introduction....9
8.2 PatternElement (Abstract)....11
8.3 PatternDefinition....11
8.4 Role....12
8.5 PatternSection....12
9 Observations Classes....13
9.1 Introduction....13
9.2 Binding....14
9.3 PatternInstance....14
9.4 PatternObservation....15
10 Formalisms Classes....17
10.1 Introduction....17
10.2 FormalizedDefinition (Abstract)....19
10.3 Assertion....19
10.4 BooleanExpression (Abstract)....19
10.5 AndExpression....20
10.6 OrExpression....20
10.7 NotExpression....20
10.8 DefinitionTerminal....21
10.9 FreeVariable....21
10.10 FormalBinding (Abstract)....21
10.11 VariableToRole....21
10.12 PropertyToRole....22
10.13 PropertyToVar....22
11 Relationships Classes....23
11.1 Introduction....23
11.2 InterpatternRelationship (Abstract)....23
11.3 RelatedPattern....24
11.4 MemberOf....24
11.5 Perspective....24
11.6 Nature....25
11.7 Category....25

11.8 KnownUse....26
12 PIN Classes....27
12.1 Introduction....27
12.2 Overview....27
12.3 PINbox Class....28
12.3.1 Collapsed....29
12.3.2 Standard....29
12.3.3 Expanded....30
12.4 Equality Class....31
12.5 BindingGlyph Class....34
12.6 Multiplicities....37
12.6.1 Stacked PINbox....37
12.6.2 MultiBranched Annotation....38
12.7 Peeling and Coalescing....41
13 PHORML Overview (Informative)....43
14 PHORML::Core Classes (Informative)....47
14.1 Entity (Abstract)....47
14.2 Model....48
14.3 NamedEntity (Abstract)....48
15 PHORML::RequiredEntitySet Classes (Informative)....49
15.1 Introduction....49
15.2 TypedEntity (Abstract)....50
15.3 MethodAndFieldContainer (Abstract)....50
15.4 Object....50
15.5 Method....51
15.6 Field....51
15.7 Type....51
16 PHORML::Reliances Classes (Informative)....53
16.1 Introduction....53
16.2 RelianceBase....54
16.3 Method Invocation....55
16.4 Field Use....55
16.5 State Change....55
16.6 Cohesion....56
Annex A: Entity Extension Examples....57
Annex B: Procedural Language Modeling....59
Annex C: AST-Based Pattern Metamodel Language (APML)....61
Annex D: Bibliography....67

Preface

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks.
Adopted specifications are available from this URL:

http://www.omg.org/spec

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

OMG Headquarters
109 Highland Ave
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this document by completing the Issue Reporting Form listed here: http://issues.omg.org/issues/create-new-issue.

General

Patterns are ubiquitous in software design, production, analysis, and maintenance. Numerous communities have arisen that support the authoring and curating of patterns of various kinds, including anti-patterns, security patterns, design patterns, architectural patterns, build patterns, and so on. There is a great need for a standard for the sharing of this information, both within and between these patterns communities. This document describes and defines a metamodel for use by these communities, to support several use cases of patterns in software, without specifying how communities should use the metamodel for their own purposes. This standard creates a foundation for information sharing, and leaves the details of which precise information to be stored and shared up to the communities that will be using that information.

For example, a design pattern community will be concerned with patterns of software design, while an architecture pattern community will be concerned with patterns of system design. Both communities have common needs surrounding how to organized the definitions of patterns, how to relate and categorize definitions, how to report on observed instances of patterns within their context, a need to display those instances to a user, and how to describe those patterns in an appropriate formal manner for their community. The context of those communities is independent of these needs, which are common to working with patterns regardless of the domain. This specification defines a container for sharing the above information.

1 Scope

The Structured Patterns Metamodel Standard (SPMS) specification defines a common standard for the definition and description of patterns as used in architecting, designing, and implementing software systems, working with software faults or security issues, and any situation where a pattern is appropriately applied.

SPMS has three main goals:

1. Sharing of pattern definitions in repositories or catalogs, including human-oriented specifications and machine-oriented formalisms for automated tool use.
2. Sharing of pattern instances – indicators of the existence of a pattern within a model – regardless of how that pattern was determined, with traceability back to the methodology, and traceability to the model artifacts that prove its existence, if applicable. These instances may come from manual assertion, or from the results of an automated too.
3. A visual representation for pattern instances that augments existing modeling representations and supports both automated production of graphical diagrams, and informal “line and box” style human-generated sketching.

The first goal is supported by the **Definitions** package, which defines a metamodel for defining and storing pattern specifications, suitable for use in tooling and repositories.

The second goal is supported by the **Observations** package, which defines a metamodel for pattern instances. The classes defined here offer support for both human-oriented use cases (consulting, investigation, education) and machine-oriented use cases (automated analysis tools, automated results analysis, etc.).

Both goals are further supported by the **Relationships** package, which augments the Definitions package with metadata appropriate for a repository or catalog of patterns. This metadata offers a set of semantic relationships between pattern definitions and instances, enhancing searchability and other use cases appropriate to the domain. Again, both human-oriented and machine-oriented use cases are supported in this package.

The **Formalisms** package supports the first goal more thoroughly for automated tool use cases and research purposes. It provides a mechanism for linking to a variety of formal metamodels such as Object Constraint Language (OCL), Knowledge Domain Metamodel (KDM), Abstract Syntax Tree Metamodel (ASTM), or Pattern Hierarchical Object Relation Metamodel Language (PHORML), depending on the needs of the modeler and community.

The third goal is supported by the **Pattern Instance Notation (PIN)** metamodel, which defines a common metamodel for the graphical depiction of pattern instances. It relies on the abstractions defined in SPMS. PIN and the corresponding elements in SPMS are equivalent in their expressive power, and have a one-to-one coherence of features.

PIN was developed hand in hand with the Patterns package of SPMS and provides a simple and human-oriented approach for quickly depicting instances of patterns, how they work in concert, and how they are expressed in an implementation or further design document. Most notably, PIN can be used entirely by itself to illustrate pattern interactions independent of an implementation, or used as an annotation with the variety of other graphical notations, such as UML diagrams.

2 Conformance

The principle goal of SPMS is the exchange of definitions, descriptions, and depictions of software patterns and related abstractions in software. To be SPMS compliant, a tool must completely support the normative SPMS model elements listed in this document as Required, which currently are contained within the Definitions package. The Observations package (Clause 9) is normative, but optional, intended to support reporting instance of patterns. The Relationships package (Clause 10) is normative, but optional, intended for use in repositories or catalogs. The Formalisms package (Clause 11) is normative, but optional, intended to support automated analysis tools. The PIN metamodel (Clause 12) is normative, a tool shall support a graphical notation. PHORML (Clauses 13-16) is informative only.

An implementation shall further provide:

- The capability to generate XMI documents based on the SPMS XMI schema capturing a tool's representation of the instance model of existing patterns within a software system.

The capability to import pattern models via representations based on the SPMS XMI schema and to map the pattern object model into the existing model of the tool.

3 Normative References

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of any of these publications do not apply.

- OMG Specification formal/2015-03-01, Unified Modeling Language (UML), v2.5
- OMG Specification formal/2016-11-01, Meta Object Facility (MOF), v2.5.1
- OMG Specification formal/2015-06-01, Diagram Definition (DD), v1.1

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Pattern

A 'software pattern,' as commonly accepted in the existing literature, such as in Desing Patterns, Gamma et. al. A common definition is “a solution to a problem within a particular context of forces and constraints.” We do not distinguish between ‘design patterns,’ ‘architecture patterns,’ or other types of patterns for the purposes of this document. This document focuses on the needs and requirements common to all patterns communities.+

Pattern Specification

As per the Pattern Based Engineering (PBE) literature, the human-oriented prose canonical specification of a pattern.

Pattern Implementation

As per the PBE literature, the embodiment of a Pattern Specification, as it appears in an implemented system.

Pattern Instance

As per the PBE literature, a single instance of a pattern within a Pattern Implementation. Note that a Pattern Implementation may give rise to many instances of the same Pattern Specification.

Pattern Description

As per the PBE literature, an informal description of a pattern, consisting of the name, and the necessary roles.

Repository

A collection of patterns definitions for pattern specifications, intended for community sharing,

Catalog

A collection of patterns definitions for pattern specifications, not intended for community sharing, perhaps internal to an automated tool.

5 Symbols

There are no symbols defined in this document.

6 Additional Information

6.1 Acknowledgments

The following companies submitted this specification:

- The Software Revolution, Inc.
- CAST
- KDM Analytics

The following persons were members of the core team that designed and wrote this specification: Jason McC. Smith (TSRI); Razak Ellafi, Camal Tazine, Bill Curtis (CAST); Nikolai Mansourov, Djenana Campara (KDM Analytics); Alain Picard, Stéphane Vaucher (Benchmark Consulting); Bob Martin, Sean Barnum (MITRE); Yann-Gaël Guéhéneuc (École Polytechnique de Montréal) and Maged Elaasar (Crossplatform Software, Inc).

The following companies supported this specification:

- TSG Consulting, Inc.
- Benchmark Consulting
- MITRE
- eCube Systems
- University of North Carolina at Chapel Hill
- École Polytechnique de Montréal

7 SPMS Overview (Informative)

The Structured Patterns Metamodel Standard (SPMS) is a metamodel for defining and describing patterns of software and other like abstractions. It is independent of software implementation language, and is highly independent of implementation details. It provides a common platform by which an architect, designer, researcher or author may express patterns as intended to be implemented, as found within an existing implementation, or proposed for refactoring purposes.

SPMS is composed of five primary packages as shown in Figure 7.1 with pre-existing OMG standards, which are outside the scope of this document, in grey.

- The ***Definitions*** package defines classes for defining patterns of various types through the PatternDefinition cluster, and for representing instances of those definitions via the PatternInstance class. The Definitions package defines the 'wrappers' for patterns. All SPMS compliant tooling, repository, or effort must support the Definitions package.

- The ***Observations*** package supports the reporting of observed instances of patterns through the PatternInstances class. A PatternInstance points to a PatternDefinition from the Definitions package, and then defines an appropriate number of Binding instances to bind the Roles from a PatternDefinition to MOF::Elements. This lets a Role be bound to elements of any number of MOF based models. PatternInstances have their observation metadata recorded by a PatternObservation, which states when, by whom, and how a PatternInstance was found.

- The ***Formalisms*** package enhances the PatternDefinition's capabilities by offering a hook for formal definitions for automated tool use. Multiple formalisms may be associated with a single PatternDefinition, to support multiple use cases or views. Any modeling formalism based off of MOF may be used to define a pattern, including UML, ASTM, KDM, or OCL. Additionally, the Formalisms package defines a simple logical expression format for combining elements from disparate formalisms, without requiring full OCL compliance. This provides researchers and students with a quicker path to working with SPMS Formalisms. The Formalisms package is Normative, but Optional. Only automated tooling is expected to include this package.

This three-prong approach lets us define the patterns, instances of those patterns, and the specifics of how a pattern is expressed in a system clearly. It also allows us to use the same metamodel to support human-oriented education or developer support through repositories, to support automated toolings through formal definitions, and to support the sharing of both pattern definitions and the results of patterns analysis, whether by automated or manual means.

- Repository support is significantly extended with the ***Relationships*** package. This defines a small set of classes for providing semantic linking between PatternDefinitions and PatternInstances. It is expected that this package will be most useful to those providing and managing a shared repository of patterns, but it may be useful to tool vendors as well. The Relationships package is Normative, but Optional.

- Finally, SPMS provides support for visualization of pattern instances within a model via the ***Pattern Instance Notation***, or ***PIN*** metamodel. PIN has a one-to-one correspondence with the relevant portions of the Patterns package, and therefore is suitable for inclusion in a graphical tool. The PIN metamodel is Normative, while the specific graphical representation is allowed to vary. An example notation is provided, suitable for both automated support and human sketching of a design as either standalone or supplementary annotation of a diagram in a notation such as UML.

Different stakeholders shall implement support for some combination of the above packages. A batch-processing automated analysis tool may implement only Definitions, Observations, and Formalisms, while a website repository with front-end support tooling for multiple interested groups will likely support all five.

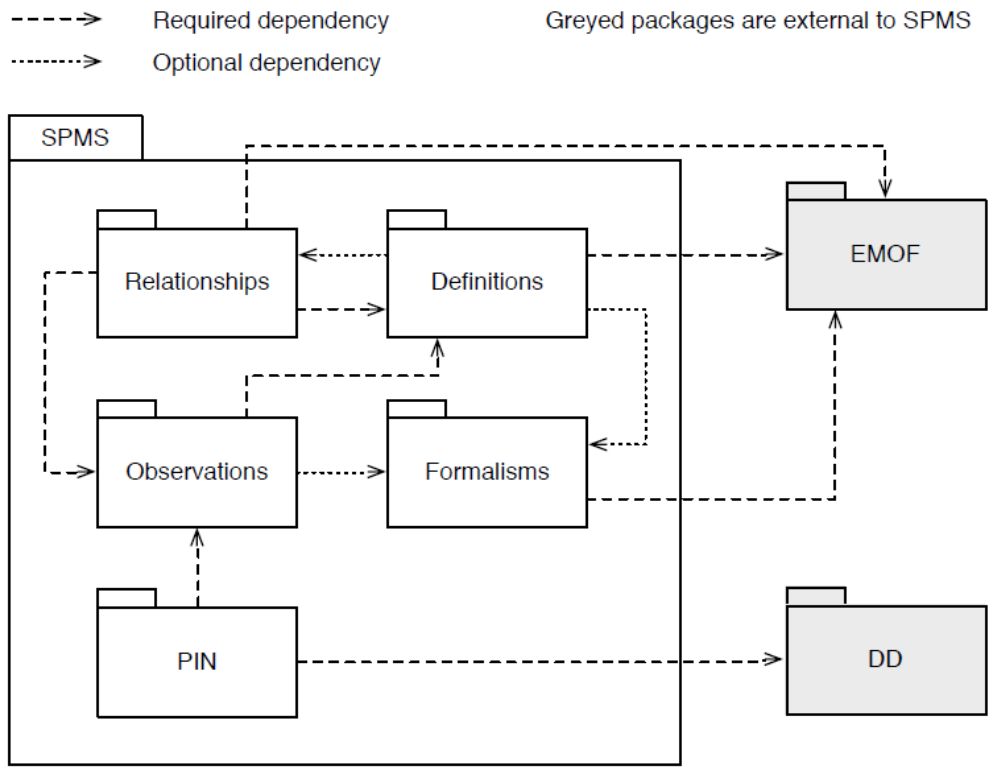

Figure 7.1 - SPMS Metamodels Overview – Normative Packages

In addition, this document defines an exemplar pattern modeling system, PHORML, which is included in Clauses 13 through 16 as a minimalist example for illustrative, non-normative purpose of modeling software patterns. Further, PHORML has an optional dependency on the APML package, described in Annex C, an exemplar approach for integrating ASTM and OCL source materials. Their package dependencies are illustrated in Figure 7.2 with pre-existing OMG standards, which are outside the scope of this document, in grey.

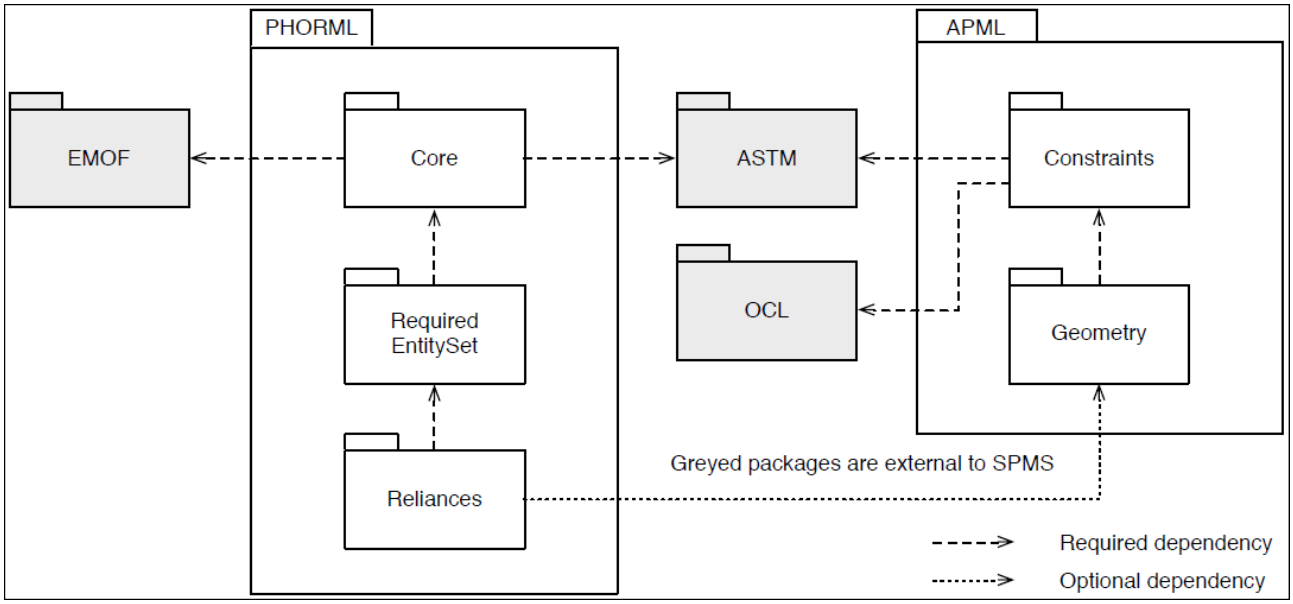

Figure 7.2 - SPMS Metamodels Overview – Non-normative Packages

This page intentionally left blank.

8 Definitions Classes

8.1 Introduction

The heart of SPMS from a modeling point of view is the Definitions package, shown in Figure 8.1. This provides the necessary small amount of formal structure needed to define pattern definitions, and do so incrementally and hierarchically. Generally speaking, a pattern can be quickly denoted by its accepted Name, and outlined by defining the participants, or Roles, that need to be fulfilled for a pattern to be expressed in a model or implementation.

A quick example using some mild formalisms may be illustrative.

Assume that a pattern may be most generally represented in the following form:

PatternName(Role1 : *a*, Role2 : *b*, Role3: *c*)

PatternName is simply the name the pattern is known by. The set of Role1, Role2, Role3 represent the conceptual elements required to form the pattern. They most closely align with the Participants listing in the canonical pattern literature format. The elements *a, b,* and *c* are variables that will be bound to concrete entities in a larger design or implementation to form a pattern instance. The above form is known as a *Pattern Descriptor*.

For example, the Pattern Descriptor ExtendMethod(OriginalBehavior : *a*, ExtendedBehavior : *b*, Operation : *c*) states that the pattern ExtendMethod has three Roles associated with it: an OriginalBehavior, an ExtendedBehavior, and an Operation.

Each PatternDefinition contains a list of these Roles, and a list of PatternSections. These PatternSections are prose entries, or pointers to external resources, that describe for a human reader the definition of the pattern as defined according to appropriate patterns communities that adopt SPMS.

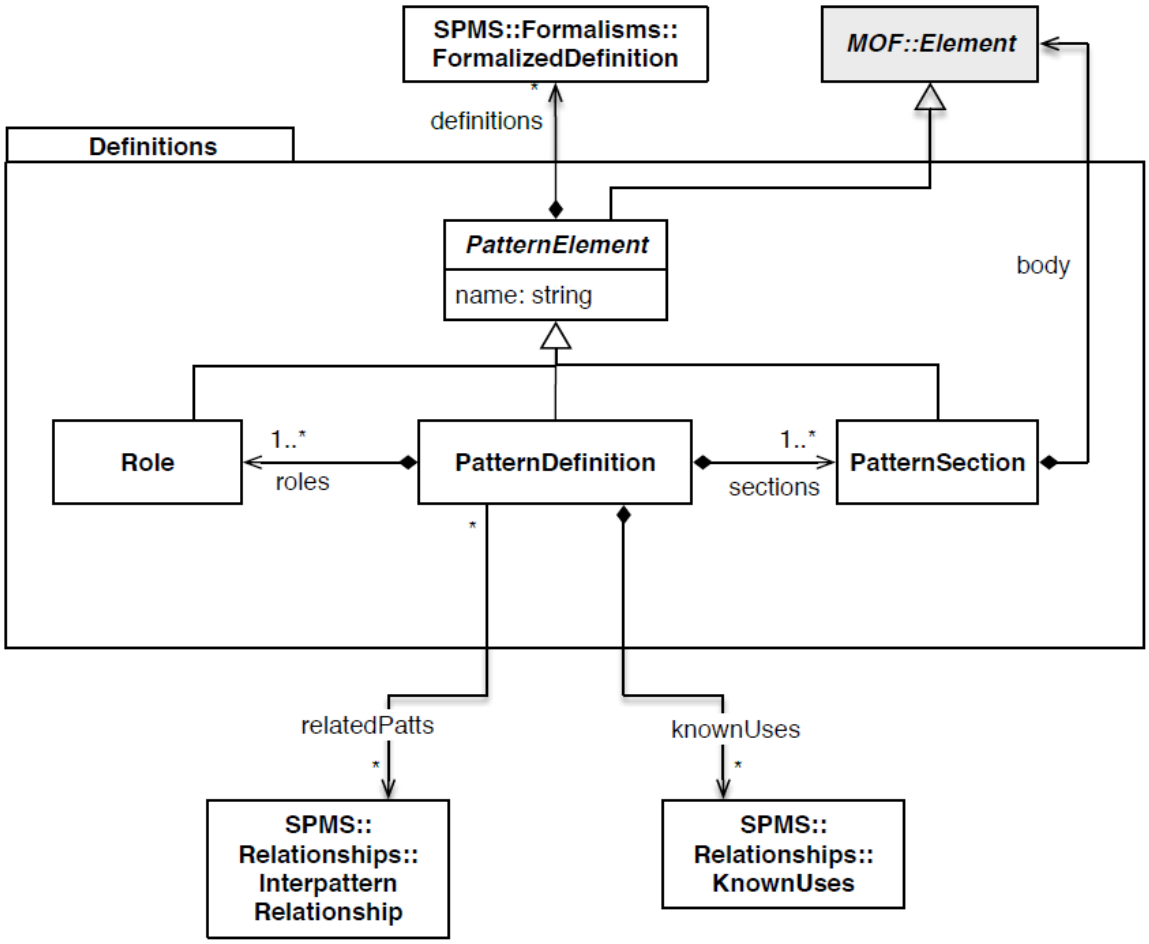

Figure 8.1 - Definitions package

8.2 PatternElement (Abstract)

Common base class for providing a name for elements within the Definitions package. Specialized by PatternDefinition, Role, and PatternSection.

Attributes (Required)

name : String	The name by which the element is referred to in the model.

8.3 PatternDefinition

Within the Pattern-Based Engineering discipline, a fully specified pattern written in the usual form as delineated by the patterns community and found in literature such as the "Gang of Four" text, is termed a *Pattern Specification.* This specification is intended for human consumption, and is the form of pattern definition that most practitioners are familiar with.

The SPMS analogue to this is the PatternDefinition, which is composed of both the traditional informal prose portions of a specification, and one or more optional formal definitions of a pattern. Instances of this class are suitable for inclusion in a repository of pattern definitions for community sharing and reference. A PatternDefinition does not represent the existence of a pattern in a particular implementation, system, or context, instead it represents the definition on how to express a pattern. The PatternDefinition can be thought of as analogous to a class in most object-oriented languages, while a PatternInstance is an instantiated object of that class.

A PatternDefinition has associations to PatternSections, one for each of the sections found in a pattern specification, and associations to one or more Roles, which define the necessary pieces of the PatternDefinition. Optional associations include links to KnownUses of the PatternDefinition as examples in existing software systems, one or more FormalizedDefinitions for analysis purposes, and links to InterpatternRelationships, which provide guidance on what other PatternDefinitions may be of interest to the consumer of this PatternDefinition. Both human- and machine-oriented tasks are therefore supported.

Generalizations

PatternElement

Associations (Required)

sections : PatternSection [1..*]	The sections of the pattern specification.
roles : Role [1..*]	The roles that are required to be fulfilled for a pattern instance to exist.

Associations (Optional)

knownUses : KnownUses [*]	A set of known uses of this pattern in the community.
definitions : FormalizedDefinition [*]	A set of formal definitions of the pattern. These may be of various forms.
relatedPatts : Relationships:: InterpatternRelationship [*]	A set of related patterns, organized for repository searching.

8.4 Role

A pattern is colloquially defined as a set of relationships between a set of entities. Roles describe the set of entities within a pattern, between which those relationships will be described. As such the Role is a required association in a PatternDefinition. A Role is analogous to an item listed and discussed in the Participants section of a design pattern following the format template of Gamma et al. in *Design Patterns*. At a structural level, a Role is simply a name that will be associated to from a Binding within a PatternInstance, both of which are defined in the Observations package. Semantically, a Role is a 'slot' that is required to be fulfilled for an instance of its parent PatternDefinition to exist. Conceptually, this is little different than the purpose of a role in a play or script. The role is independent of the actor that will play that part and it exists within the context of the script. The same script (PatternDefinition) has roles (Roles) that are filled by actors to produce unique productions of the play (PatternInstance).

Generalizations

PatternElement

8.5 PatternSection

A PatternSection is a description of a portion of a PatternDefinition. The description may be free-form prose in the provided String type, or the body property may be a simple URI that points to an external resource that contains the description of this PatternSection. It provides information about, among other possibilities, the structure, uses, counter-examples, application, or history of the pattern. A PatternSection corresponds to a part of a Pattern Specification as would be found in the patterns literature. There is no single consensus on how to describe a pattern, so there is no single suggested list of PatternSections provided here. For instance, the Hillside Group, a well-known and established patterns community centered around software design patterns, offers several example pattern templates. Pattern communities that prefer the template put forth by Erich Gamma et al in the seminal *Design Patterns* text will use a template with the following Sections: Name, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code and Usage, Known Uses, and Related Patterns. An alternative is the AG Template with Sections named Name, Aliases, Problem, Context, Forces, Solution, Resulting Context, Rationale, Known Uses, Related Patterns, Sketch, Author, Date, References, and Example.

In addition, there will be a wider variation among different pattern communities, and certain classifications of patterns, such as anti-patterns, have their own special needs such as Mitigation or Workaround sections. By offering pattern communities the opportunity to define their own collections of defined PatternSections, and standard templates of PatternSections for their own use, SPMS provides both the flexibility required to support multiple communities while offering a unified mechanism of definition and retrieval.

Generalizations

PatternElement

Attributes (Required)

body : String	The contents of the PatternSection, or a URI pointing to said contents in another resource.

9 Observations Classes

9.1 Introduction

The Observations package provides a suite of classes to describe observations of patterns as defined in the Definitions package. Just as classes in object-oriented systems describe the structure of object instances to be created from them, a PatternInstance represents an instance of a defined pattern as described by a PatternDefinition, which may be bound to an arrangement of model elements within a model. The PatternInstance binds the Roles from the PatternDefinition to elements in a model, using instances of the Binding class. A PatternInstance may have a PatternObservation, which describes how the instance was found, when it was found, and so on. A PatternObservation may have an association with a FormalizedDefinition from the Formalisms package for traceability.

Continuing the Pattern Descriptor notation from sub clause 8.1, the Decorator pattern can be expressed as the combination of two instances of other patterns: ObjectRecursion (Woolf, 1996) and ExtendMethod (Smith, 2005), in the following *Pattern Definition*, represented as a reduction rule:

ObjectRecursion(Object : *a*, Recurser : *b*, Terminator : *c*, Init : *x*)

ExtendMethod(OriginalBehavior : *b*, ExtendedBehavior : *d*, Operation : *e*)

Decorator(Component : *a*, Decorator : *b*, ConcreteComponent : *c*, ConcreteDecorator : *d*, Operation : *e*)

This states that a Decorator pattern is evident when instances of two sub-patterns, ObjectRecursion and ExtendMethod, are proven to exist, and in such a way that the design or implementation entity that fulfills the *Recurser* Role of the ObjectRecursion instance also simultaneously fulfills the *OriginalBehavior* Role of the ExtendMethod instance. Any appropriate element from an existing model, whether it is UML, KDM, GASTM, or other, can be used as a fulfiller for a Role. The exemplar PHORML described in Clauses 13 through 16 is a simple example of an appropriate and minimalist approach for unifying a number of approaches. Element instances may be subcomponents of a PatternDefinition as well, defining entities and reliances between them.

To create a PatternInstance, the variables represented by the Roles in a PatternDefinition are bound to concrete entities by a Binding. For instance, a pattern instance of ExtendMethod can be represented by binding the variables to code entities as in:

ExtendMethod(OriginalBehavior : `Alert`, ExtendedBehavior : `BeepAndMailAlert`, Operation : `beep`)

where Alert, BeepAndMailAlert and beep are respectively two classes and a method in a design or implementation.

Figure 9.1 shows the classes in the Observations package.

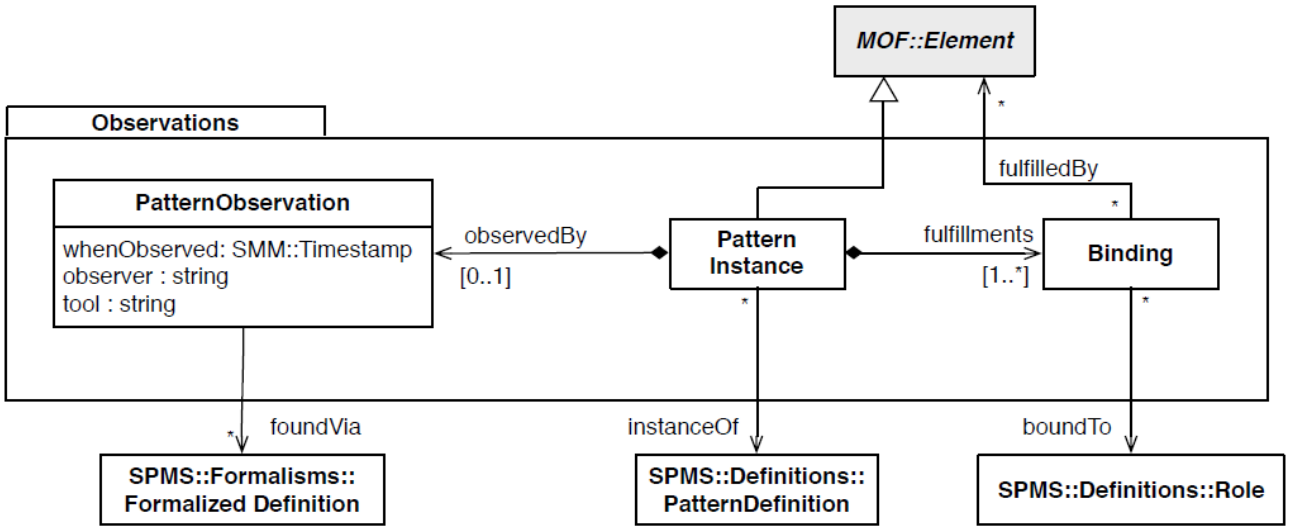

Figure 9.1 - Observations Classes

9.2 Binding

A Binding associates a Role with one or more entities that fulfill it for the particular PatternInstance that contains the Binding. The associated Role must be an associated element of the PatternDefinition pointed to by the PatternInstance that holds this Binding.

Associations

boundTo : SPMS::Definitions::Role	The Role being bound.
fulfilledBy : MOF::Element [*]	The entities within the model that fulfill the Role for this particular pattern instance. There may be more than one.

9.3 PatternInstance

A PatternInstance is a specific instance of a pattern, as expressed within a model. This instance indicates the existence of the associated PatternDefinition. Many PatternInstances may be associated with one PatternDefinition.

At least one Binding will be associated with each PatternInstance, one for each Role in the matching PatternDefinition.

Generalizations

MOF::Element

Associations

instanceOf : SPMS::Definitions::PatternDefinition	A reference to the definition for the pattern being instantiated.
fulfillments : Binding [1..*]	The set of bindings between the PatternDefinition's Roles and the Entities that express this particular instance of the pattern.
observedBy : PatternObservation [0..1]	How was the pattern determined to exist in the model?

9.4 PatternObservation

When a PatternInstance is determined to exist, regardless of the methodology used to uncover it, it is often useful to record how it was found, and by whom. This is accomplished via a PatternObservation, which provides information about the circumstances surrounding the detection of the PatternInstance. A PatternObservation adds an optional reference to a formalized definition of the pattern, to allow a reviewer to see which formalism was used by the detection method described in the PatternObservation. The PatternObservation shares much in common conceptually with the Software Metrics Meta-Model (SMM) Observation class, but it was determined that not tying SPMS to SMM was preferred. The core elements of SMM::Observation are therefore duplicated here.

Attributes

whenObserved : String	Identifies the "moment" when the PatternInstance was recorded.
observer: String	Identifies the observer of the PatternInstance.
tool : String	Identifies the method used to determine the PatternInstance. It may be an automated software tool, a consultant performing a manual inspection, a reference to a piece of documentation, and so on.

Associations

foundVia : SPMS::Formalisms::FormalizedDefinition [*]	A reference to the formal definition used for this particular observation.

This page intentionally left blank.

10 Formalisms Classes

10.1 Introduction

One goal of SPMS is to allow the community to share pattern specifications, including definitions of a more formal nature. These are of particular relevance to automated tool systems for the application, detection, or refactoring of patterns. Unfortunately there is no one mechanism or formalism that is agreed upon or suitable for all pattern domains or use cases. A developer of a static analysis tool for patterns support is going to require a different formalized view onto a pattern than will a developer of a dynamic analysis tool for patterns support, or than will a consultant looking for a UML model for verification against client documentation, and so on. With the immense breadth and depth of possible formal models for pattern definition, we feel that it is both efficient and prudent to allow practitioners, researchers, and developers to have a variety of models from which to choose for their particular needs, without being locked in, or locked out of, a specific modeling style.

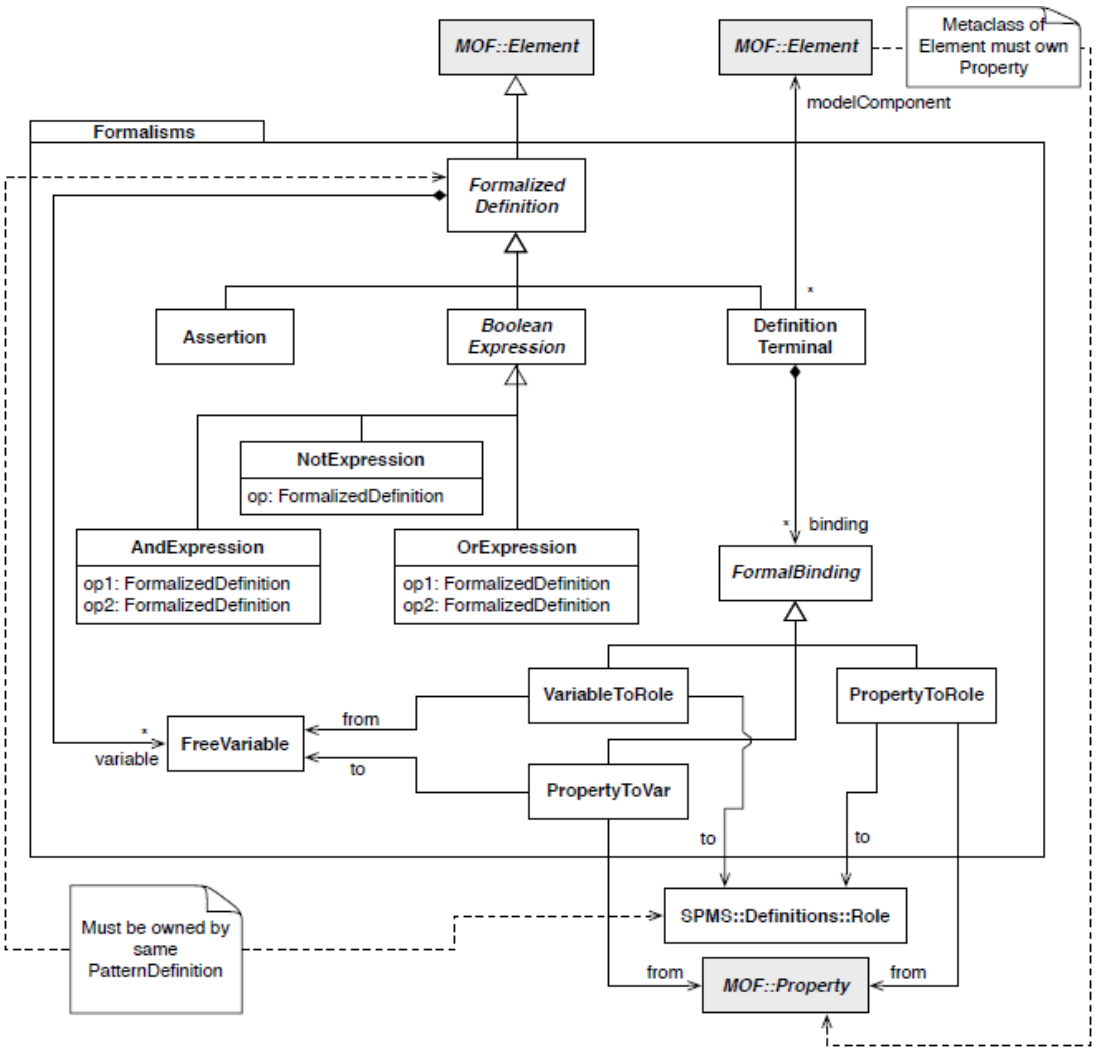

Figure 10.1 - Formalisms package

Because we desire multiple definitions of a pattern for a variety of use cases, many instances of FormalizedDefinition can be referenced by a single PatternDefinition. Each of these FormalizedDefinitions, in turn, can be composed of further instances of FormalizedDefinitions as sub-models, by using an extremely lightweight boolean logic mechanism defined here. This allows the composition of model fragments from a number of modeling domains into a comprehensive whole. This satisfies our need for a single pattern formal definition requiring multiple views to properly describe the pattern. Many patterns, for instance, have both unique structural forms and run-time behaviors. It is unlikely that a single OMG model is going to capture all the nuances of each, but a combination of ASTM and OCL models, for instance, or PHORML and KDM, may be sufficient. For this reason, SPMS defines a minimalist composition mechanism for those that wish to have a lightweight yet compliant composition model. For more complex needs, an OCL expression may be used by an instance of DefinitionTerminal referencing an OCL model. This Formalisms package is shown in Figure 10.1.

As an example of a minimalist modeling system for implementation patterns, Clauses 13 through 16 of this document informationally describe PHORML, a lightweight non-normative metamodel for representing object-oriented systems.

10.2 FormalizedDefinition (Abstract)

The base class for the Formalisms package. This simply provides an entry point for the SPMS PatternDefinition and PatternObservation classes. A FormalizedDefinition defines a set of variables that are binding points to act as a bridge between an element of a formal model such as a UML Class, and an element of a PatternDefinition, such as a Role. FormalBindings provide the bridge mechanism.

Associations

variables : FreeVariable — An owned variable to use in FormalBindings.

10.3 Assertion

There are times when we wish to track where a pattern instance was detected or asserted to exist, but no formal method was used, and no formal model exists for it. This may happen, for instance, when a consultant speaks with a development team, and they ascertain the existence of a pattern in a software system, but have no formal model. The Assertion class provides us with a way of expressing this. It is most useful in conjunction with the BooleanExpression class family when a portion of a pattern formalism is able to be formally modeled in one of the MOF expressible specifications, but another portion is not.

Generalizations

FormalizedDefinition

10.4 BooleanExpression (Abstract)

A superclass for simple lightweight composition in SPMS. Subclasses of BooleanExpression allow a practitioner to combine disparate model fragments from different modeling approaches, such as UML, OCL, KDM, ASTM, and so on. The And, Or, and Not Expression subclasses provide a full combinatorial expressiveness, as they are compositive in nature, and boolean trees can be trivially formed. Two examples of such trees are as follows. Assume that **W**, **X**, **Y** and **Z** are model fragments expressed in one or more metamodels that derive from MOF:

1. Or (And(**W**, **X**), And(**Y**, **Z**))Or indicates that either of the two paths may be used, while And indicates that both sub-paths must be present. Here, a human or automated tool has two possible models to choose from: one that includes both **W** and **X** model fragments (**W** && **X**), and one that includes both **Y** and **Z** model fragments (**Y** && **Z**).
2. And (Or(**W**, **X**), Or(**Y**, **Z**))Here, the And indicates that both paths must be satisfied, while the Or indicates that either sub-path may be chosen. This provides the human or machine consuming this formalism with four choices: (**W** && **Y**), (**W** && **Z**), (**X** && **Y**) or (**X** && **Z**).

As described in 10.10, FormalBindings are used to stitch the chosen model fragments together into a composite entity.

This is the only normative composition style that must be complied with for SPMS compliance. If adopters of SPMS wish to include more complex compositions of MOF derived entities and measurements, they are free to optionally use a logic or constraint system of their choosing, such as OCL. However, such techniques are well outside the scope of SPMS.

Generalizations

FormalizedDefinition

10.5 AndExpression

A simple logical conjunction of the two referenced definitions. It indicates that both sub-models are required to define the formalism.

Generalizations

BooleanExpression

Associations

op1 : FormalizedDefinition	A reference to a FormalizedDefinition.
op2 : FormalizedDefinition	A reference to a FormalizedDefinition.

10.6 OrExpression

A simple logical disjunction of the two referenced definitions. It indicates that either sub-model is applicable in the formalism. This is applicable when two alternate forms of a formalism fragment exist, and either may be used to model the pattern.

Generalizations

BooleanExpression

Associations

op1 : FormalizedDefinition	A reference to a FormalizedDefinition.
op2 : FormalizedDefinition	A reference to a FormalizedDefinition.

10.7 NotExpression

A simple logical negation of the referenced definition. It indicates that the sub-model must not exist in the pattern defined by the formalism. OCL, for example, can be used to model a constraint which is then required not to be found in an instance of the pattern. As most metamodels provide such a feature, this expression is rarely used, but included for completeness.

Generalizations

BooleanExpression

Associations

op1 : FormalizedDefinition	A reference to a FormalizedDefinition.

10.8 DefinitionTerminal

This class is a leaf on a Formalism composition tree. It will refer to a MOF::Element based model element. The type of metamodel is not specified. This allows any MOF based metamodel to provide elements for inclusion in a FormalDefinition. For instance, a definition may include an ASTM tree fragment representing a necessary source code representation, a KDM model representing a build scenario, or a constraint model specified in OCL.

Generalizations

FormalizedDefinition

Associations

modelComponent : MOF::Element	A reference to the MOF::Element derived model element that is to be included in this FormalDefinition.
binding : FormalBinding [*]	An owned binding between formal elements.

10.9 FreeVariable

A FreeVariable describes an unbound variable in a FormalizedDefinition. This allows any of the subclasses of FormalizedDefinition to expose elements of its internal definition for external binding to elements exposed by other definitions, including FormalizedDefinitions and PatternDefinitions.

10.10 FormalBinding (Abstract)

An abstract class that provides a common entry point for kinds of bindings between formal models and PatternDefinitions. The formalism model fragments that are stitched together by the BooleanExpression instances will have elements that need to be bound to the FreeVariables in a FormalizedDefinition, and the Roles in a PatternDefinition. For example, 'The FreeVariable Foo in the formalism is bound to the Role Factory in the PatternDefinition, and is fulfilled by the Element Bar in the included model fragment.' FormalBindings are the glue that compose multiple model fragments into a cohesive whole formal definition.

10.11 VariableToRole

This class is a specialization of FormalBinding that binds a FreeVariable from any of the FormalizedDefinition specializations to a Role in a PatternDefinition.

Generalizations

FormalBinding

Associations

from : FreeVariable	A reference to a FreeVariable.
to : SPMS::Definitions::Role	A reference to a Role in a PatternDefinition. The Role must be owned by the same PatternDefinition that owns the DefinitionTerminal which owns this binding.

10.12 PropertyToRole

This class is a specialization of FormalBinding that binds a MOF::Property owned by a MOF::Element associated as the modelComponent of a DefinitionTerminal, to a Role in a PatternDefinition. It is used in special cases where the model is self-contained enough (i.e., one fragment) to need no inter-fragment stitching. In those cases, a FreeVariable is not needed to act as an intermediary.

An example of such a binding would be from a UML::Operation instance within an instance of UML::Class, to a Role in a PatternDefinition.

Generalizations

FormalBinding

Associations

from : MOF::Property	A reference to an MOF::Property owned by an instance of the metaclass of MOF::Element associated with the DefinitionTerminal that owns this binding.
to : SPMS::Definitions::Role	A reference to a Role in a PatternDefinition. The Role must be owned by the same PatternDefinition that owns the DefinitionTerminal which owns this binding.

10.13 PropertyToVar

This class is a specialization of FormalBinding that binds a MOF::Property owned by a MOF::Element associated as the modelComponent of a DefinitionTerminal, to a FreeVariable from any of the FormalizedDefinition specializations.

Generalizations

FormalBinding

Associations

from : MOF::Property	A reference to a MOF::Property owned by an instance of the metaclass of MOF::Element associated with the DefinitionTerminal that owns this binding.
to : FreeVariable	A reference to a FreeVariable.

11 Relationships Classes

11.1 Introduction

The Relationships package defines classes to enable rich searching and semantic association in a repository or catalog of PatternDefinitions. The package overview is shown in Figure 11.1. InterpatternRelationship provides semantic linkages between PatternDefinitions. A PatternDefinition can have multiple InterpatternRelationship connections to allow for a rich connected network. KnownUse provides specializations of PatternInstances suitable as examples of a PatternDefinition in concrete situations.

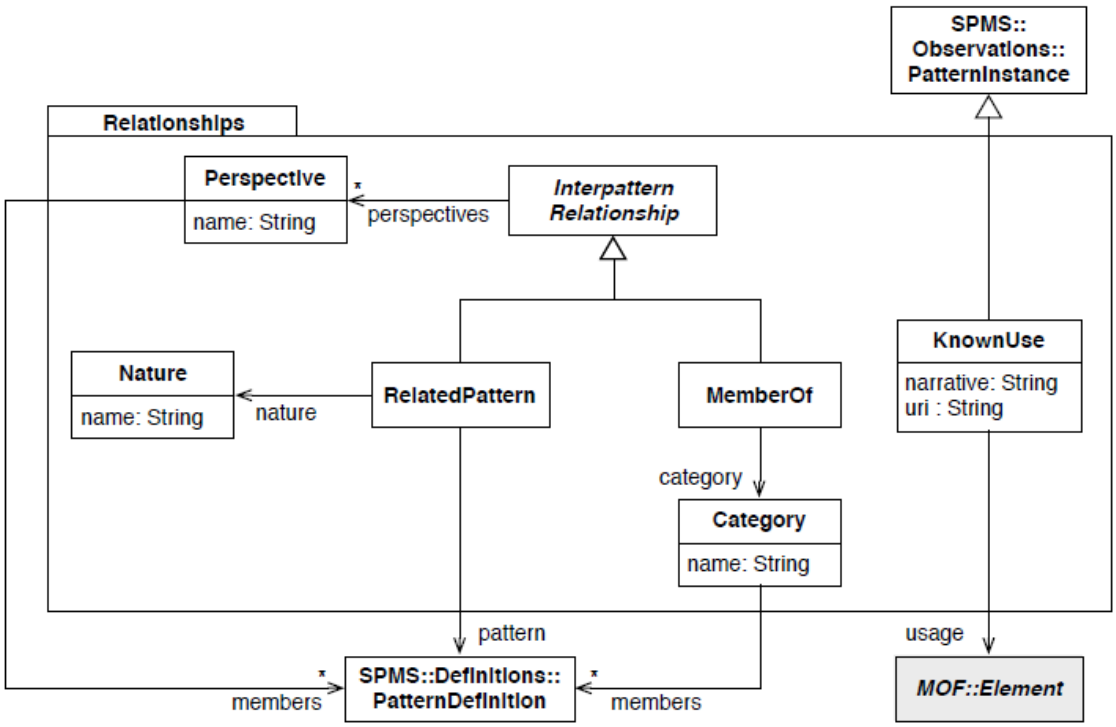

Figure 11.1 - Relationships package

11.2 InterpatternRelationship (Abstract)

A simple directed relationship between patterns. Each InterpatternRelationship in a system has a number of Perspectives for which the InterpatternRelationship is valid. For instance, a researcher and a developer may have different relevant concerns when searching or viewing a repository. By indicating which Perspective or Perspectives are of interest to them, they can be presented with only the data that is appropriate.

Associations

perspectives : Perspective [*]	Perspectives for which this relationship is valid.

11.3 RelatedPattern

An InterpatternRelationship specialized to point to a related pattern.

Generalizations

InterpatternRelationship

Associations

pattern : SPMS::Definitions::PatternDefinition	The pattern that this relationship points to.
nature : Nature	Descriptor of the relationship between the two PatternDefinitions.

11.4 MemberOf

An InterpatternRelationship specialized to indicate inclusion in a Category.

Generalizations

InterpatternRelationship

Associations

category: Category	The category that this pattern definition is a member of.

11.5 Perspective

Describes a perspective that defines an area of interest for a particular group of stakeholders. InterpatternRelationships are considered to be included in a Perspective if they reference a Perspective. A Perspective has a single string that indicates the name of the Perspective. Each community will form their own canonical set of terms. The following are one example of such a set.

Developer	Defines a perspective for Developer interest.
Research	Defines a perspective for Research interest.
Management	Defines a perspective for Management interest.

There should be one instance of each perspective kind in a system, with references to it.

Attributes

name : String	The name of the perspective.
members : SPMS::Definitions::PatternDefinition [*]	Members of this perspective.

11.6 Nature

A descriptor of the relationship between the source pattern definition and the target pattern definition. The value for a Nature is a simple string, and as with the Perspective, communities will select and define their own canonical sets of terminology. An example set might consist of:

ChildOf	The source pattern definition is a component of the target pattern.
	Converse of ParentOf.
ParentOf	The source pattern definition has the target pattern as a component.
	Converse of ChildOf.
PeerOf	The source and target patterns are peers. (Reflexive)
Requires	The source pattern requires the target pattern.
RequiredBy	The source pattern is required by the target pattern.
VariantOf	The source pattern is a variant of the target pattern. (Reflexive)
CanAlsoBe	
MitigatedBy	For Anti-Patterns: The source pattern is fully resolved by the target pattern.
	Converse of Mitigates.
Mitigates	The source pattern is a resolution for the target anti-pattern.
	Converse of MitigatedBy
CompensatedBy	For Anti-Patterns: The source pattern is worked around by the target pattern.
	Converse of Compensates.
Compensates	The source pattern can be used to work around the target anti-pattern.
	Converse of CompensatedBy.

Optimally, there should be one instance of each nature kind in a system, with references to it.

Attributes

name : String	The name of the nature of the relationship.

11.7 Category

A Category is a simple grouping element for gathering related PatternDefinitions into clusters. Unlike Perspectives or Natures, the names of Categories are not restricted. There should be one instance of each category kind in a system, with references to it.

Attributes

name : String	The name of the category.
members : SPMS::Definitions::PatternDefinition [*]	Members of this category.

11.8 KnownUse

The KnownUse class is used to describe known examples of patterns (i.e., pattern instances) in real world situations. Possibilities at this point include narrative descriptions, references to models, and links to source code repositories.

KnownUse represents an instance of a PatternDefinition, specializing PatternInstance as suitable for inclusion in a pattern repository's storage of a PatternDefinition.

Generalizations

SPMS::Observations::PatternInstance

Associations

narrative : String	A prose description of the known use, context, system, etc.
uri : String	A URI for web access to a source repository, website, etc.
usage : MOF::Element	A reference to a model for a KnownUse.

12 PIN Classes

12.1 Introduction

It is possible to use UML to graphically depict some definitions and instances of patterns using SPMS. It is, not, however, optimal in most cases. The Pattern Instance Notation (PIN) was developed to provide an alternative for when UML is either inappropriate or cumbersome. A full discussion of the background of PIN is beyond the scope of this document. For further details, please reference *The Pattern Instance Notation: A Simple Hierarchical Visual Notation for the Dynamic Visualization and Comprehension of Software Patterns*, Jason McC. Smith, The Journal of Visual Languages and Computing, Elsevier Publishing, October 2011.

The intent of PIN is to allow developers, architects, and consultants to quickly and naturally depict instances of patterns in a simple and clear format that is based on a rigorous formal foundation, without exposing the user to the underlying mathematical formalisms.

12.2 Overview

The Pattern Instance Notation is a simple graphical notation designed for informal and formally-backed use cases. It is comprised of two basic graphical elements: the PINbox, representing one or more individual SPMS::PatternInstances, and the BindingGlyph, representing one or more SPMS::Bindings.

PIN was developed to fix some deficiencies in using existing graphical notations for depicting individual instances of patterns in large-scale systems, such as UML Collaborations or Pattern::role tags. PIN represents instances of design patterns as first-class entities. They are not dependent solely on external entities, but can exist visually independent of other graphical notations, and can be used to illustrate and discuss interactions solely between pattern instances in a clean and concise manner.

PIN is also based firmly on the foundation of SPMS, using the same conceptual model. The entities in SPMS were not given their own graphical notation elements for three reasons. One, it provides a clean distinction between SPMS for analytical tools, and PIN for user visualization tools. Secondly, PIN offers enhanced support for multiplicities that SPMS does not. An automated tool will not benefit from multiplicity simplification, but a human viewer of a visualization will. Thirdly, PIN offers scalability through multiple detail-granularity control mechanisms, again, designed to assist human users.

The PIN metamodel is simple enough that it is shown in its entirety in Figure 12.1, along with the necessary interactions from the Definitions and Observations packages to provide an explanation of the inner workings of PIN. The PINbox will be described first, then the Equality and BindingGlyph classes. Then, and only then, will their interactions and use case scenarios be described, as a series of examples.

The PIN metamodel is normative, but the specifics of the graphical notation described here are not. A vendor is free to implement their own representation, the representation included here is an example notation that has been successfully used in various patterns related contexts. The contents of the symbols in the diagrams in this section are not normative, but only for explanatory purposes.

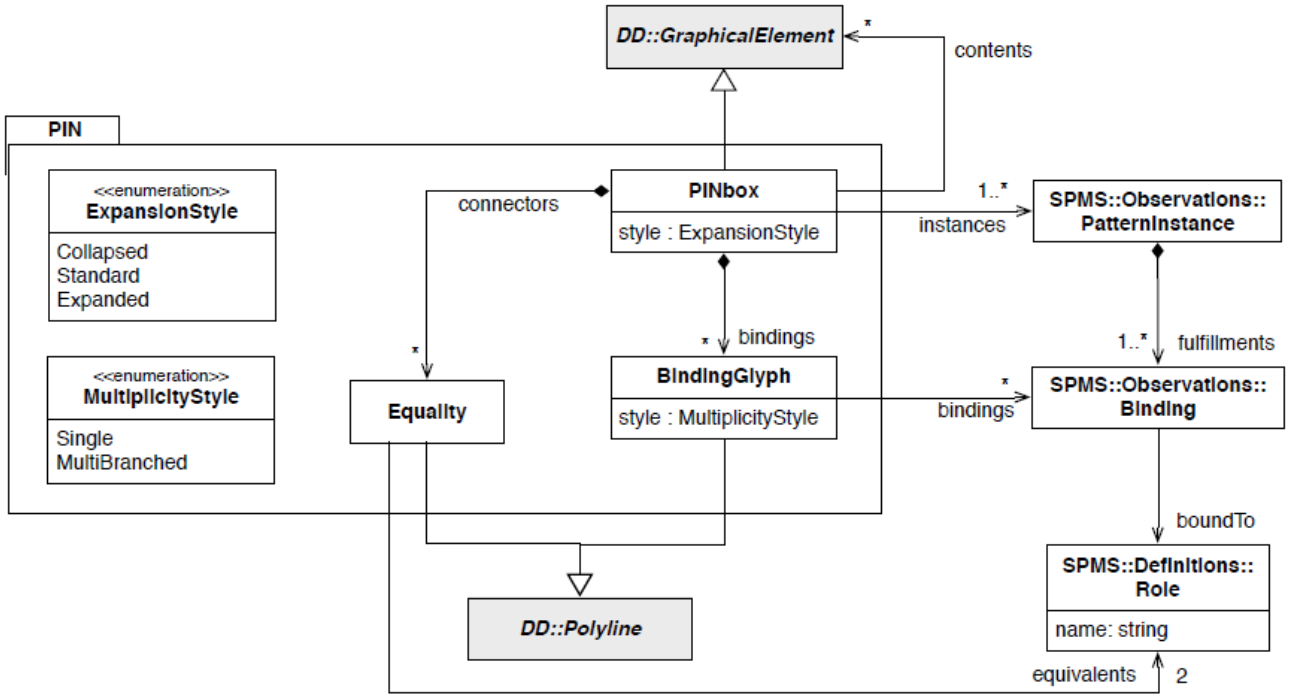

Figure 12.1 - PIN Module

12.3 PINbox Class

The PINbox is the basic visual unit in PIN. It represents one or more instances of patterns in a system. This section will concern itself with a PINbox which represents a single SPMS::PatternInstance.

The PINbox class is derived from GraphicalElement of the Diagram Definition 1.0 specification. It contains a container named instances, which holds one or more SPMS::PatternInstance instances. PINbox also contains two or more instances of the BindingGlyph class, which will be described below. Finally, a PINbox contains a style attribute, which indicates which of three states the graphical notation should be drawn in, Collapsed, Standard, or Expanded. These correspond to increasing levels of detail being exposed to the user. Each has utility in different scenarios.

Generalizations

DD::GraphicalElement

Attributes

style : ExpansionStyle	A tri-state value: Collapsed, Standard, Expanded which controls the amount of detail portrayed.

Associations

instances : SPMS::Observations::PatternInstance [*]	A collection of PatternInstances that this PINbox represents. All PatternInstances will be instances of the same PatternDefinition.
bindings : BindingGlyph [*]	The set of graphical binding elements associated with this PINbox.
connectors : Equality [*]	The set of inter-PINbox connectors associated with this PINbox.
contents : DD::GraphicalElement [*]	The graphical contents of this PINbox for Expanded mode.

The three style forms are described next.

12.3.1 Collapsed

A Collapsed PINbox, as shown in Figure 12.2, is a simple box containing the name of the pattern being represented by this instance. The border of the PINbox is drawn as a thick, shaded border with a rounded edge. This both distinguishes it from other common graphical elements, and provides the basis for further levels of detail, as shown in the following section. The name displayed comes from the SPMS::PatternDefinition associated from the SPMS::PatternInstance associated via instances.

This form is intended to be used as a quick mnemonic in informal use cases, or as a placeholder in a tool wishing to show the existence of a pattern instance, with minimal detail.

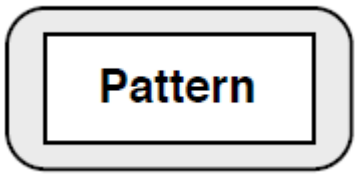

Figure 12.2 - Collapsed PINbox

12.3.2 Standard

The Standard PINbox form, as shown in Figure 12.3, shows the additional utility of the thick border – it is where the names of the Roles associated with the PatternDefinition are listed, while maintaining visual consistency with the Collapsed form. The Role names can appear in any order around the PINbox, the selection of which Role appears in which position is left to individual tools implementing PIN to decide. It is noted that re-ordering the Role names can result in vastly different optimal layouts of PIN annotated diagrams.

This is the most common usage of the PINbox, as it displays all of the necessary components of the pattern, and is suitable for addition to a UML diagram, or used in conjunction with other PINboxes for a more pure pattern-oriented illustration.

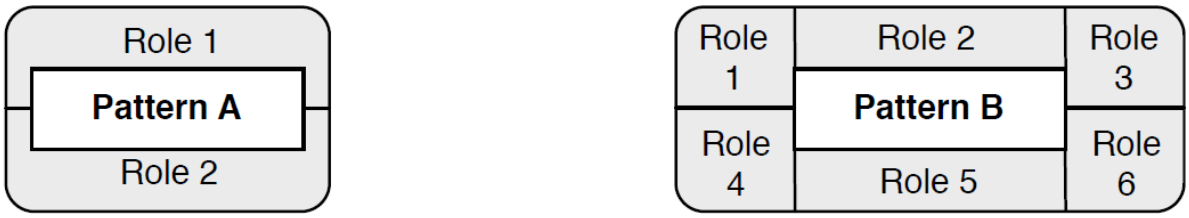

Figure 12.3 - Standard PINbox

12.3.3 Expanded

The Expanded PINbox form, as shown in Figure 12.4, literally expands the interior of the PINbox to create a new canvas on which graphical elements can be drawn. This uses the *contents* association of the PINbox. Any graphical notation may be drawn here, whether other PINboxes, UML such as Class or Sequence Diagrams, or other depictions that help illustrate the pattern being represented.

Use cases include, but are not limited to:

- Exposing the subpatterns of the external pattern's PatternDefinition, to provide further detail on the specific instance being shown.
- Providing a reference for the pattern in the form of the 'canonical' UML sample diagram provided in the Structure section of the design pattern literature specification.

Subsuming portions of a larger design within an enclosed frame to simplify a complex diagram into more easily understood abstractions.

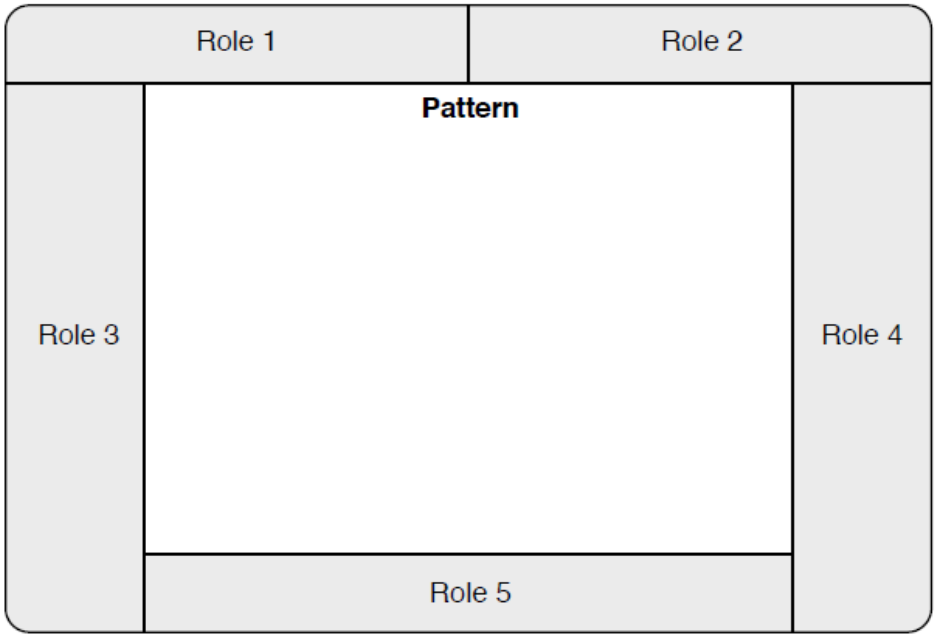

Figure 12.4 - Expanded PINbox

12.4 Equality Class

The Equality class represents a connection between two or more PINboxes, indicating an equivalence-link between a set of specified roles. It ties together multiple PINboxes, multiple instances of patterns, such that, whatever is bound to one of the Roles involved in the Equality, must be bound to the others in the set.

This PINbox-to-PINbox connector, independent of any other notation or entities, is what allows a PINbox diagram to illustrate the inter-pattern bindings involved in a PatternDefinition's subpatterns list. It enables a user to describe and depict interactions between individual pattern instances in the abstract, independently of a larger system or design.

An Equality is depicted as a simple line between two or more PINboxes, as shown in Figure 12.5. Line weight is non-normative. Here, it indicates that whatever eventually will fulfill Role 3 of Pattern A, must fulfill Role 1 of Pattern B as well. In other words, both roles are fulfilled by the same programmatic entity, and this entity is what ties together the two pattern instances to form a larger abstraction. This is shown in Figure 12.6 which also shows the first use case listed for the Expanded form of a PINbox: increased level of detail.

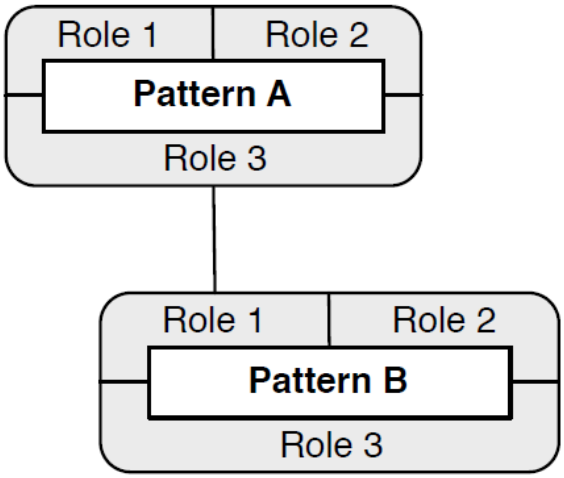

Figure 12.5 - Equality between two PINbox Rules

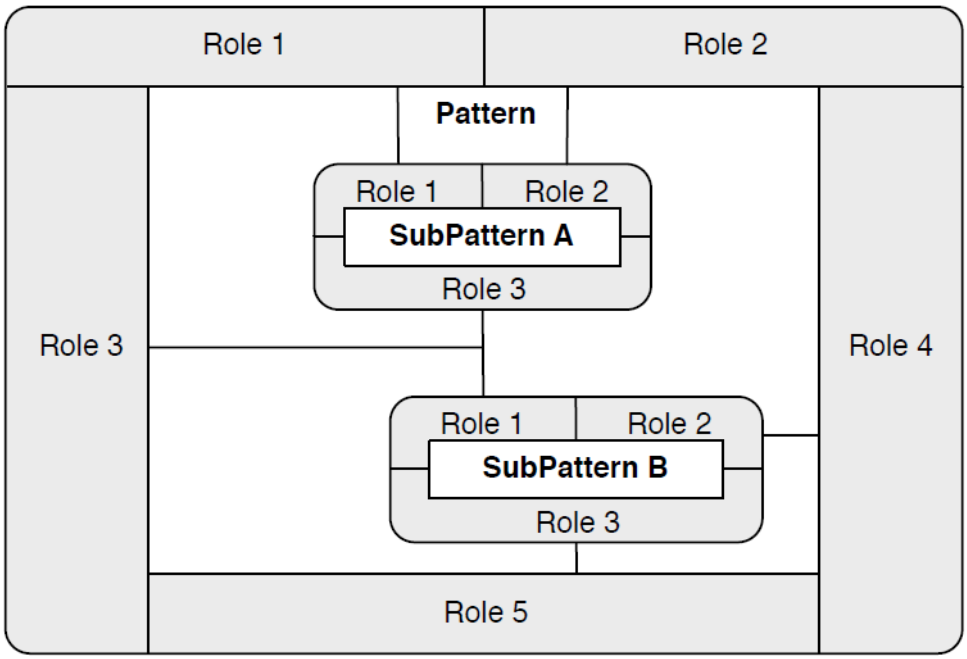

Figure 12.6 - Expanded PINbox illustrating internal PINbox diagram

To further illustrate this use case, Figure 12.7 shows the PIN diagram for the definition of the Decorator pattern that was provided in Clause 11. As in that formal case, the Recurser Role of the ObjectRecursion instance and the OriginalBehavior Role of the ExtendMethod instance are bound to the same entity. And, that entity is what is externally

bound to the Decorator Role of the overall Decorator pattern. The other Roles of the subpattern instances are tied to their equivalent Decorator Roles through further Equality glyphs. The thicker lines here are for presentational emphasis only and have no additional meaning.

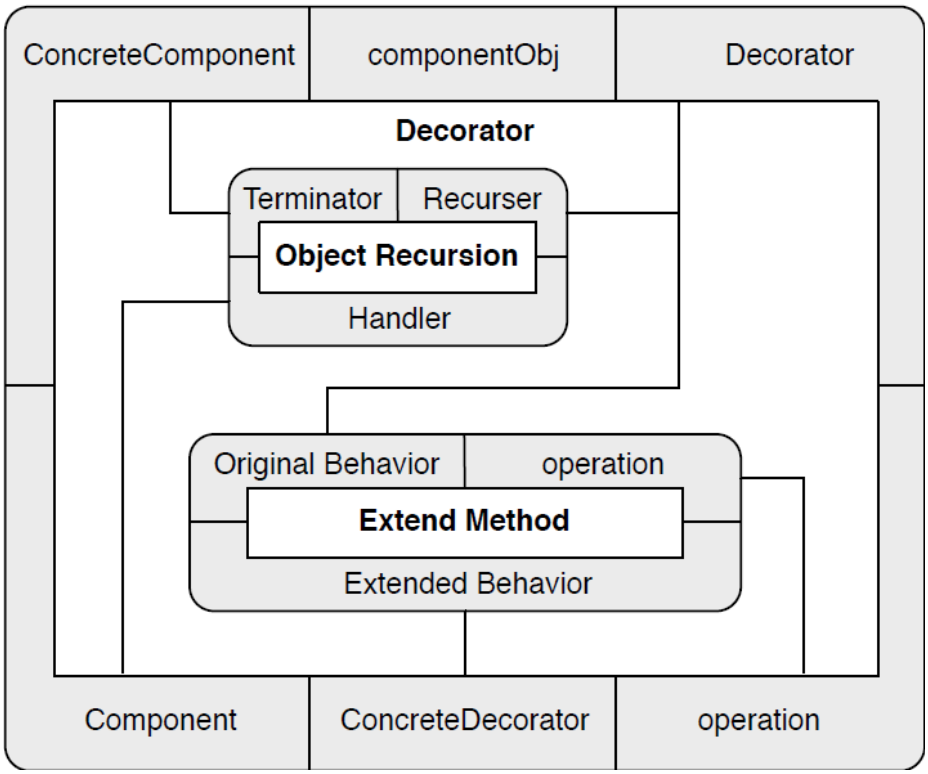

Figure 12.7 - Decorator as an Expanded PINbox with internal PINbox definition

Generalizations

DD::Polyline

Attributes

style : MultiplicityStyle A two-state value: Simple, or MultiBranched.

Associations

equivalents : SPMS::Definitions::Role [2..*]

12.5 BindingGlyph Class

The BindingGlyph represents a binding from a Role of a PatternInstance to an entity that fulfills that Role. Since PIN is designed to be used with a number of graphical notations that may represent those entities, including UML, the BindingGlyph does not directly associate to another graphical element. Instead, it contains a collection of associations to other SPMS::Bindings, and they provide the endpoints of the bindings to be depicted.

By decoupling the Bindings from the BindingGlyph, we enable two areas of flexibility. One, the Multiplicities which will be discussed in the next Clause. Secondly, we allow the BindingGlyph to be used with any number of appropriate other graphical notations.

A BindingGlyph is a directed relationship, with the source (tail) of the line starting at the PINbox being bound, and the target (head) of the line connected to the entity the Role is being bound to. The line weight is non-normative.

Figure 12.8 shows the simplest example of a BindingGlyph, indicating a Collapsed PINbox being fulfilled by a UML class. As stated in the Collapsed PINbox discussion in 12.3.1, this is used in cases where the context is obvious, such as an instance of a Singleton pattern.

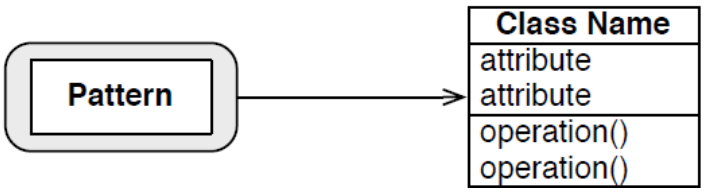

Figure 12.8 - Collapsed PINbox with BindingGlyph

A more common use is with the Standard PINbox, where the BindingGlyph is used to connect each Role with the entity that fulfills it, from a larger diagram. An example of this is shown in Figure 12.9, where an instance of a Flyweight design pattern is bound to the elements of a simple UML Sequence diagram.

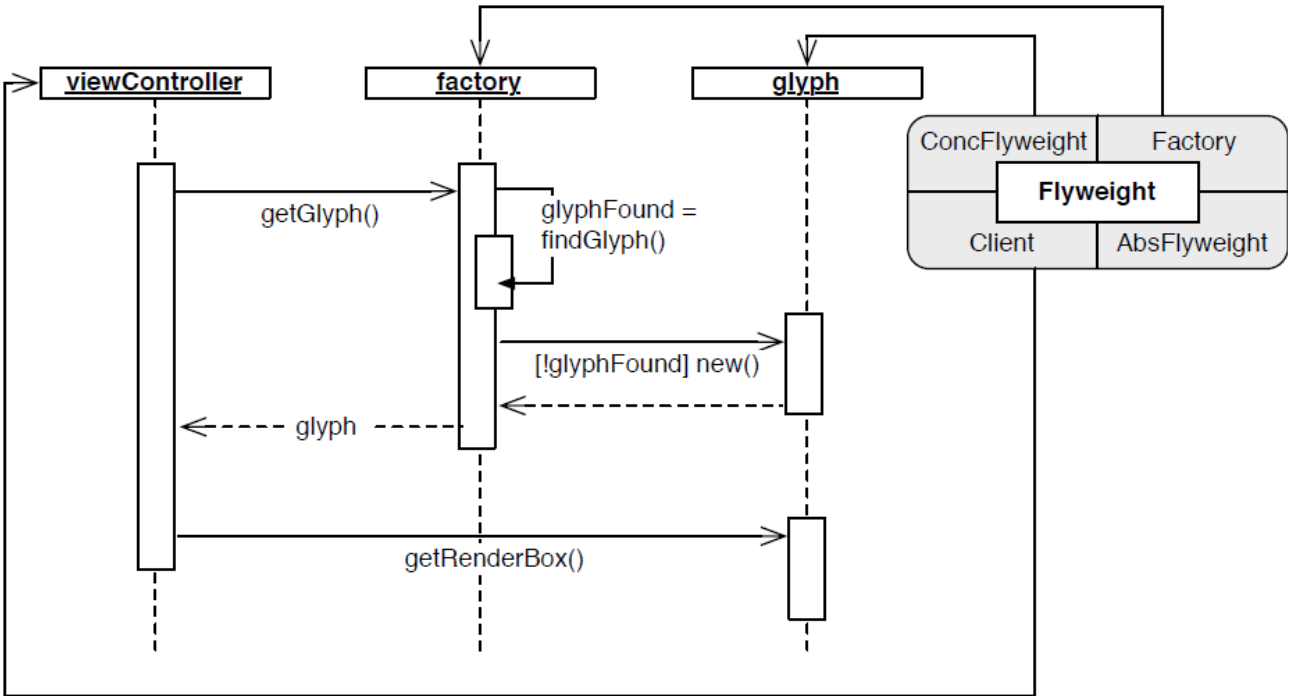

Figure 12.9 - BindingGlyphs used with PINbox in UML Sequence diagram

Figure 12.10 shows another example of BindingGlyphs being used with a UML diagram. In this case, one of the two instances of Decorator is being shown in the UML Class diagram provided as the example Structure in the canonical Decorator write-up in *Design Patterns*. This clearly shows the bindings between the Roles of the pattern, and the elements in the UML diagram, and is suitable for annotating the UML diagram of a system with instances of patterns.

This binding capability can also be used to illustrate bindings internal to a PINbox. Much as the Equality connectors were used in the Expanded PINbox form to show internal connections, we now see how any diagramming notation can be placed effectively within an Expanded PINbox's canvas, as in Figure 12.11. This encapsulation is appropriate for subsuming sections of a larger diagram into a PINbox for later revealing when the detail is desired, or for providing a 'reference' to an established definition for user education or reminding. The differing line weights inside the canvas are not normative, but merely suggested to differentiate BindingGlyphs from lines on the encapsulated diagram.

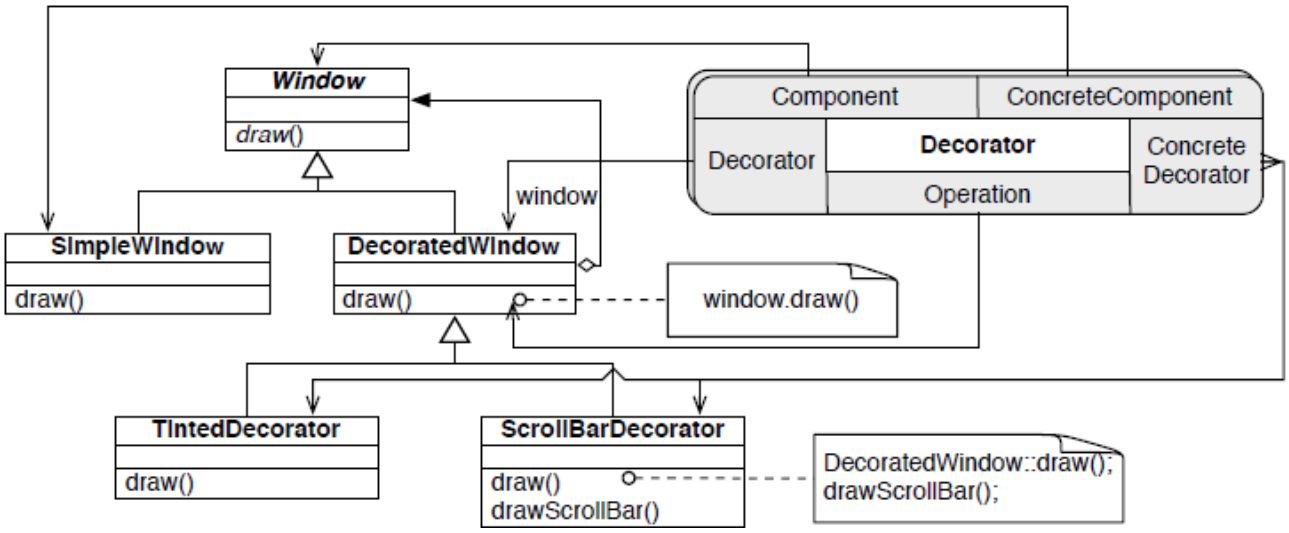

Figure 12.10 - BindingGlyph used to bind single Decorator instance to UML Class diagram

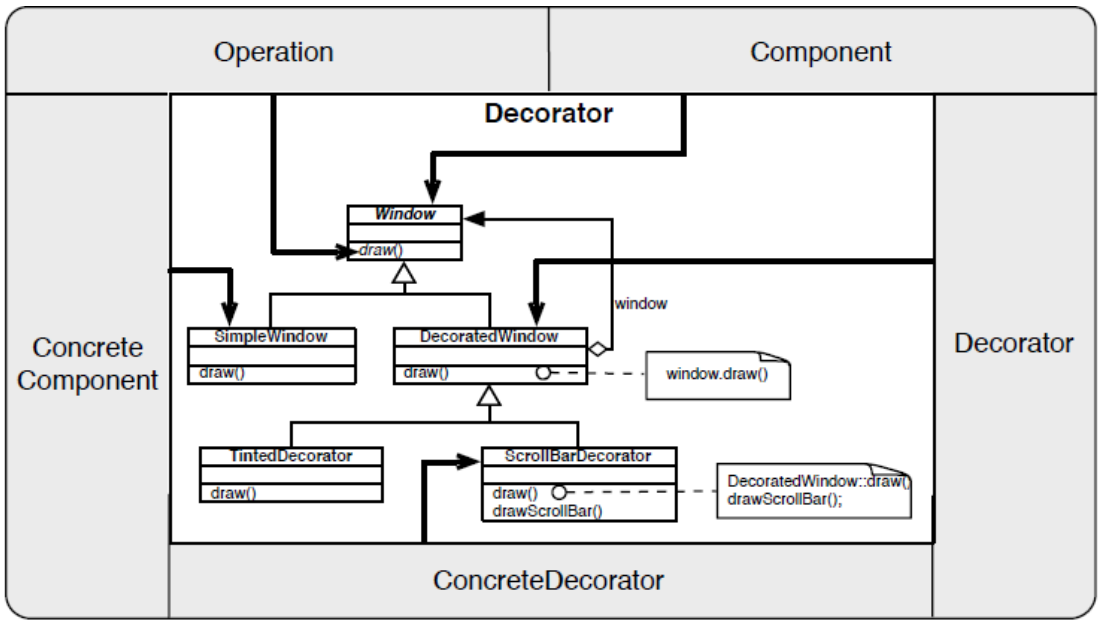

Figure 12.11 - Expanded PINbox for Decorator with internal binding to UML Class diagram

Generalizations

DD::Polyline

Attributes

style : MultiplicityStyle	A two-state value: Simple, or MultiBranched.

Associations

bindings : SPMS::Observations::Binding [1..*]	The set of conceptual binding elements associated with this BindingGlyph.

12.6 Multiplicities

PIN also supports multiplicities of pattern instances. In cases such as the Decorator pattern, it is common to have multiple overlapping pattern instances, one for each combination of classes and such that form one example of a Decorator. It can be cumbersome to try and manage multiple individual PINboxes in such cases, particularly when they share nearly all of their bindings and state. If a PINbox contains multiple PatternInstances through its *instances* association, then the Stacked form is triggered. If multiple instances share Equality connector or Bindings at one but not both, ends, then a MultiBranched annotation is used to annotate the Polyline used to depict the association.

12.6.1 Stacked PINbox

In such situations, PIN provides the *Stacked* PINbox. It is indicated by a secondary boundary offset to the upper left slightly, as shown in Figure 12.12. This provides a visual cue that this is rather like a three-dimensional stack rising off of the diagram. (There is no three dimensional aspect to the rendering of these graphics, they are considered to be in the same Z-ordering as all other instances in the same diagram.) There is no correlation between the 'depth' of the stack and the number of instances being represented. That information is usually discernible from the binding information. If it is not, then an appropriate annotation may be selected.

This is the reasoning behind having multiple PatternInstances being represented graphically by a single PINbox element. Practice has shown that if each and every PatternInstance is given its own PINbox, diagrams very quickly become unwieldy and difficult to work with. In this manner a single PINbox can represent multiple instances.

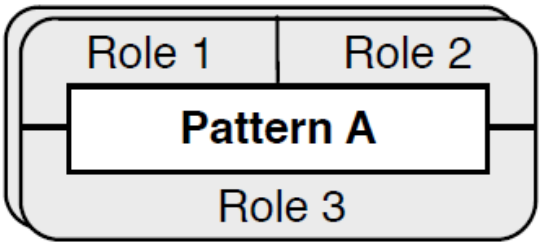

Figure 12.12 - Stacked PINbox

12.6.2 MultiBranched Annotation

A Stacked PINbox cleans up a diagram by minimizing the amount of redundant information. If multiple PINboxes representing multiple PatternInstances are in a diagram, and those PINboxes share the same bindings for multiple roles, then they can be effectively stacked to reduce the complexity of the diagram. Figure 12.13 shows an example of two PINbox instances that are of the same kind, Pattern A, and share an equality on Role 3 with Role 1 of Pattern B.

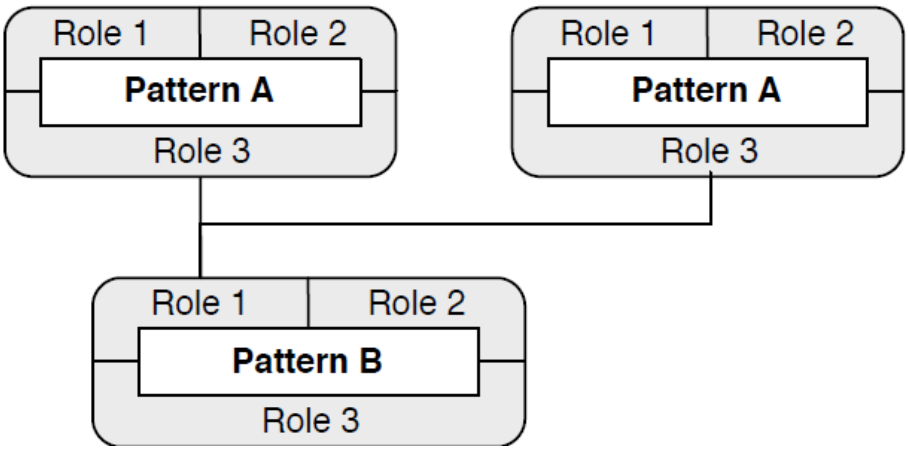

Figure 12.13 - Multiple instances of the same pattern

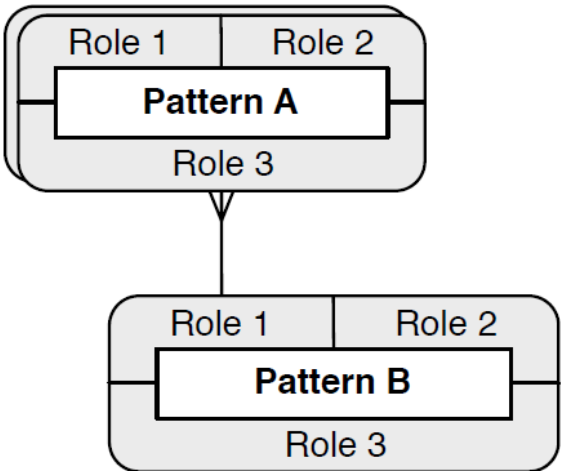

Figure 12.14 - Stacked PINbox with MultiBranch annotation on Equality glyph

Figures 12.15 through 12.18 following are more complex examples provided for informational purposes. Figure 12.15 demonstrates a MultiBranched form of a BindingGlyph being used to indicate multiple pattern instances sharing a bound entity, in this case, ConcreteDecorator. There are two instances of Decorator in this diagram, which is simply annotating the canonical Structure diagram from the Decorator description from *Design Patterns*. Since four of the five Roles in the PatternInstances are shared, only ConcreteDecorator needs a MultiBranch annotation.

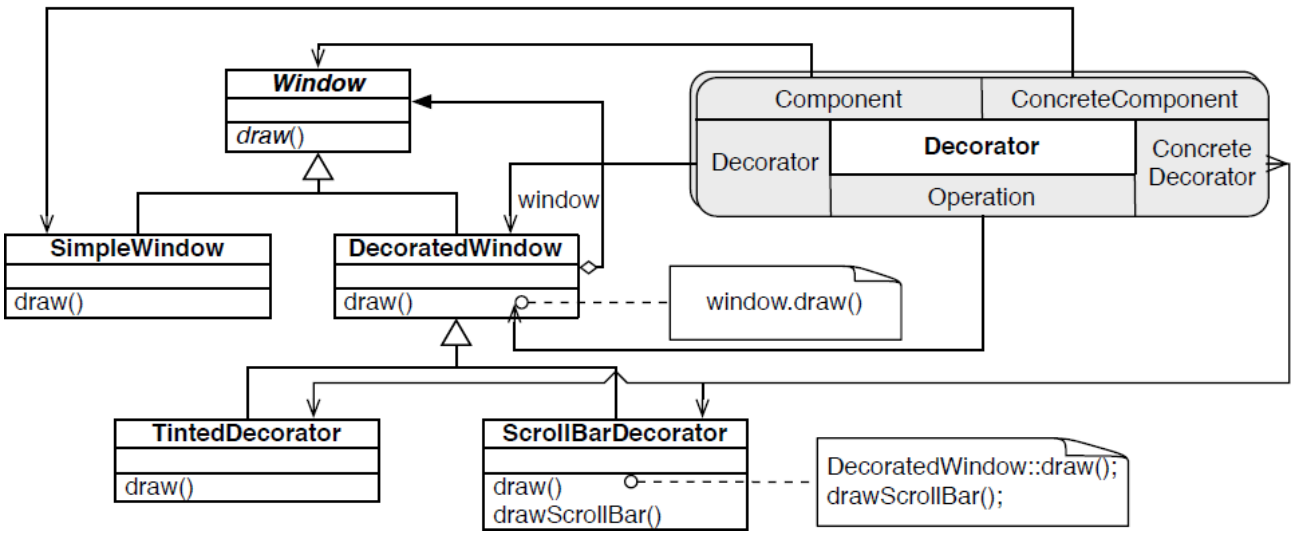

Figure 12.15 - Stacked PINbox of Decorator instances with MultiBranch annotation on Decorator Role

A more extreme version of a Stacked PINbox is shown in Figure 12.16. Here, eight formal individual instances of the Abstract Factory design pattern are coalesced into one Stacked PINbox.

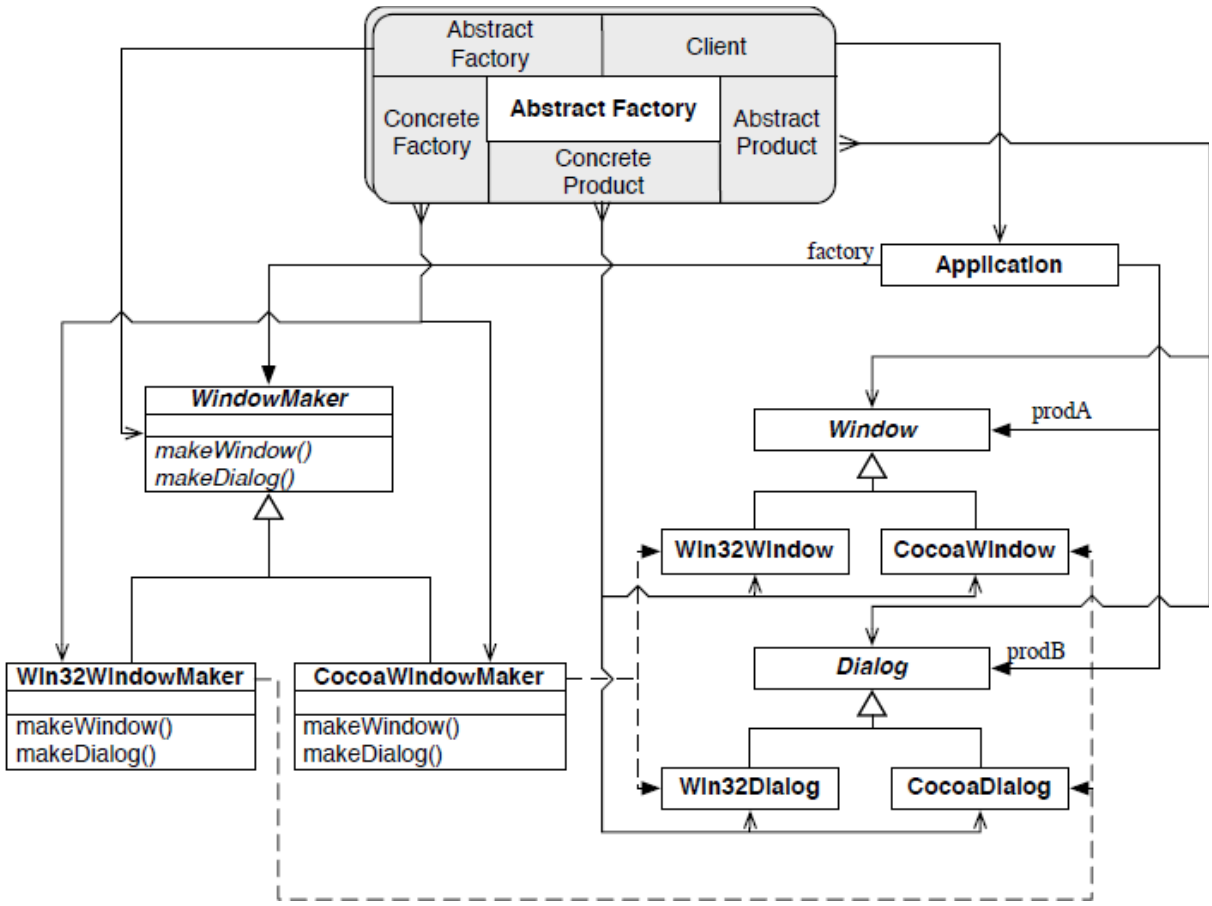

Figure 12.16 - Eight instances of Abstract Factory stacked into a single PINbox

The MultiBranch annotation can also be used within an Expanded PINbox, as shown in Figure 12.17. Now, we are showing both instances of the Decorator pattern, unlike in Figure 12.11 where we showed only one. Note however that here we are using the MultiBranch to indicate multiple satisfiers within the exemplar UML model of a specific role, not indicating a multiplicity of PatternInstances. The annotation is equally suitable for both.

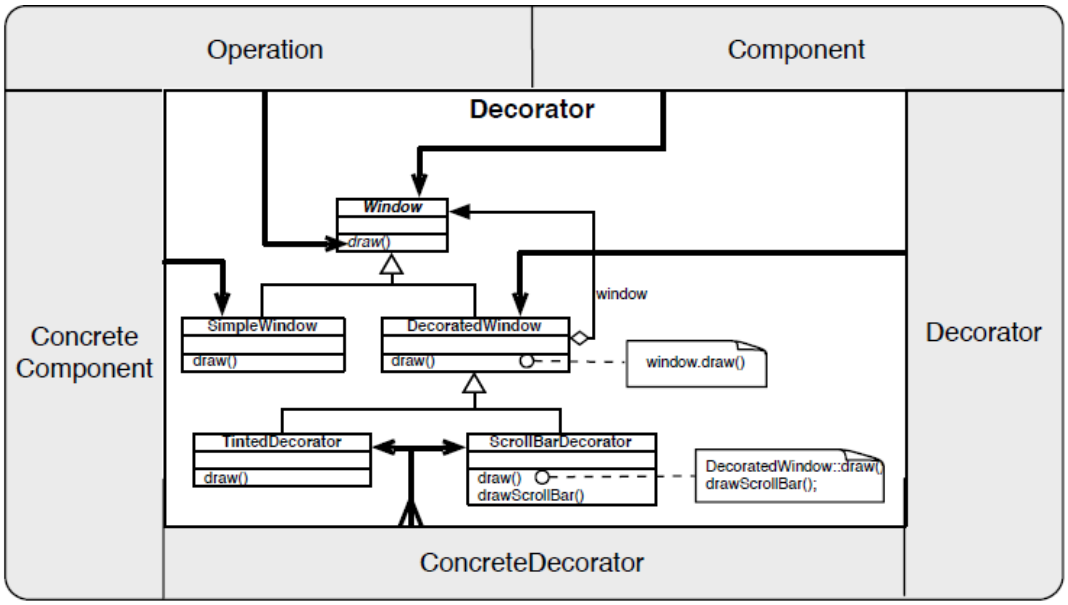

Figure 12.17 - Two instances of Decorator shown in one Expanded PINbox

12.7 Peeling and Coalescing

The canvas of an Expanded PINbox can be used to draw any number of diagrams on. One particularly useful use case is to use this canvas to 'pull in' or 'coalesce' elements of a larger diagram into a single conceptual unit, where applicable. This allows the PINbox to *reduce* the amount of complexity on a larger diagram, instead of adding to it.

Figure 12.17 above is the *coalesced* form of Figure 12.15. The external entities have been pulled inside the PINbox. Assuming that the UML structure in Figure 21 is part of a larger UML diagram, this PINbox can then be used in the Standard form, and the larger UML diagram is simplified. The Roles ringing the PINbox act as proxies for the original UML entities. Connections are propagated through the PINbox border via the Roles. At any time the PINbox can be expanded, and the original UML structure exposed.

The inverse of this behavior is *peeling*. By reversing the coalescing process, the original UML diagram can be reconstituted. This peeling off of the outer PINbox, much like the outer layer of an onion, exposes the internals to the larger diagram, allowing direct connections to take place once again.

One use case of peeling is to expose subpatterns involved with a single PINbox. For instance, in Figure 12.15, two instances of the Decorator pattern are shown via a Stacked PINbox. From the definition of Decorator, and the diagram in Figure 12.7, we know that Decorator is comprised of an instance of ObjectRecursion and an instance of ExtendMethod. We can peel off the outer PINbox and expose the inner patterns, as in informational Figure 12.18.

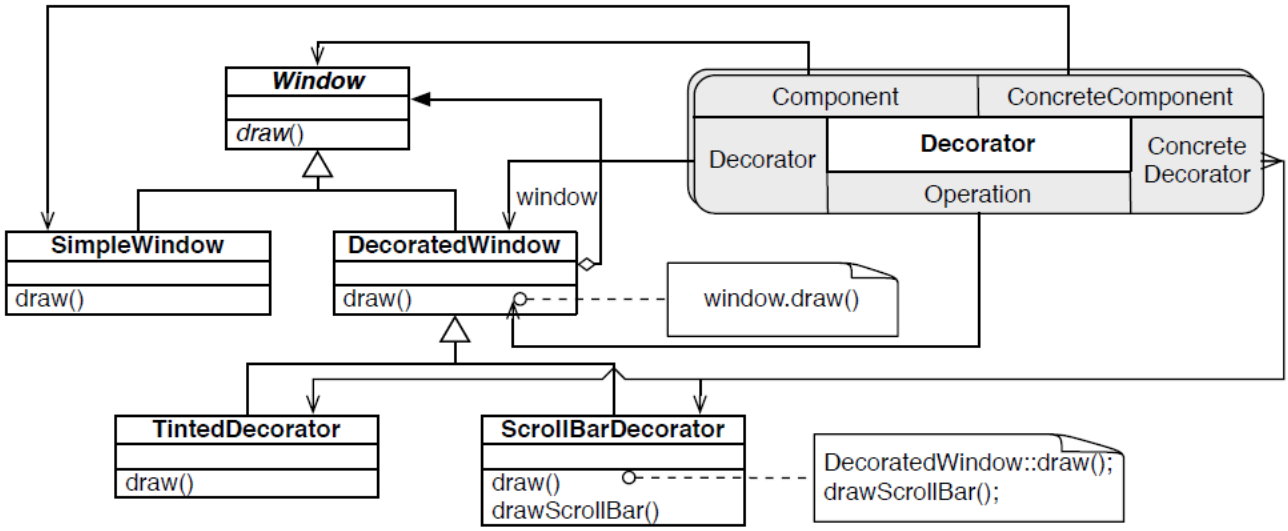

Figure 12.18 - Peeled Decorator instances

This is the same information as in Figure 12.15, but more detail is exposed. The PINboxes have been annotated with a small tab to indicate their ownership by an enclosing, but not shown, PINbox. A graphical tool may use these tabs as a connecting point for illustrating the set of PINboxes included in a peeled PINbox. Standard PINboxes are being shown here, but any of the three forms may be used, as with any other PINbox: Collapsed, Standard, or Expanded.

Note that this process may continue as needed, with more detail exposed through peeling, or less detail exposed through coalescing. In this manner, the granularity can be controlled to be precisely what is needed at that moment for human consumption, yet always have a formal underpinning due to the metamodel in SPMS.

13 PHORML Overview (Informative)

The Pattern-Hierarchy Object-Relation Modeling Language (PHORML) is an example modeling system for pattern definitions to be used in conjunction with SPMS. Figure 13.1 shows the overall structure and packages of PHORML. PHORML is a minimalist approach to modeling design patterns in software implementations, yet is expressive enough to embody the entirety of relationships available in object-oriented programming, from a formal foundation.

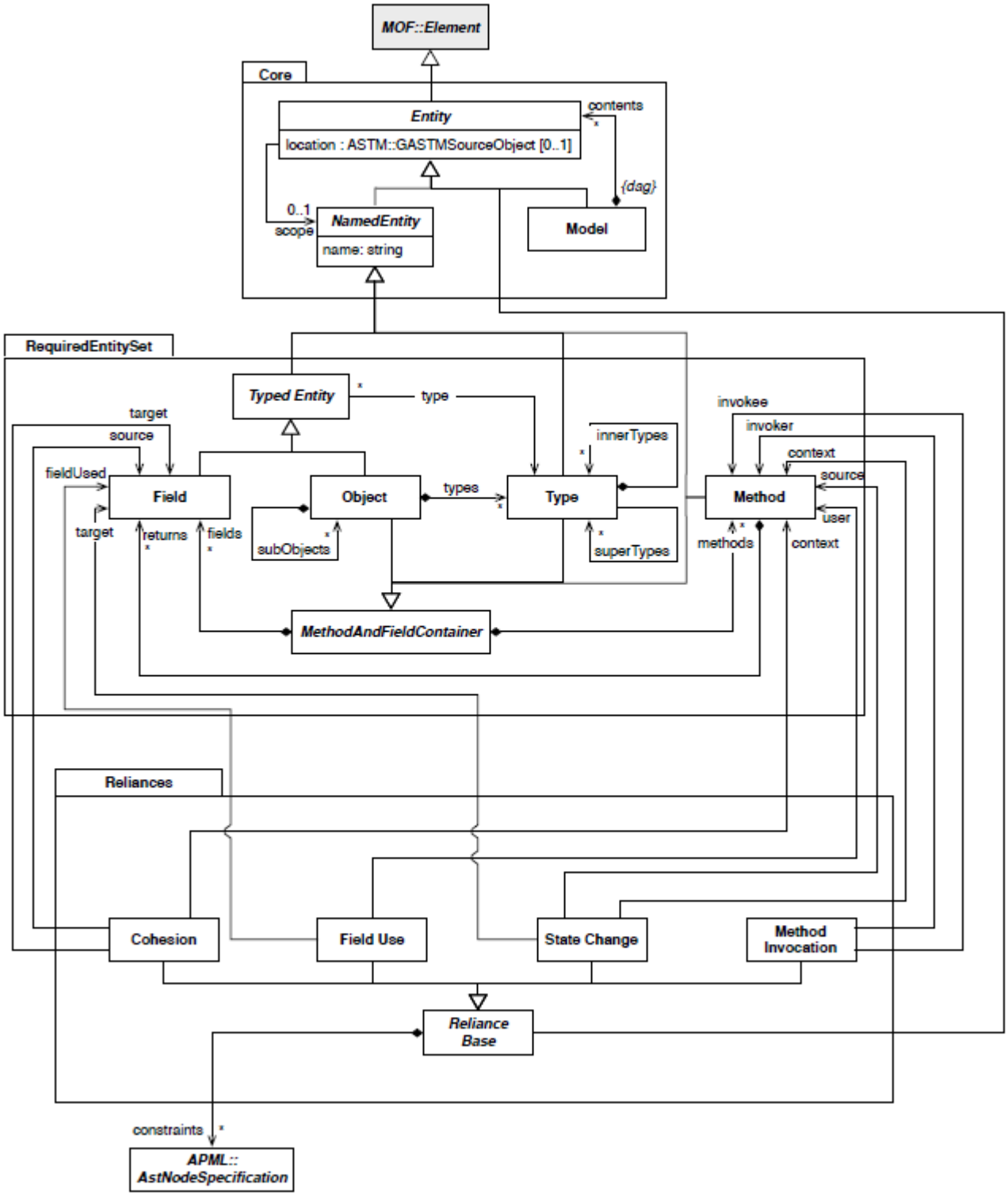

Figure 13.1 - Complete PHORML

The Required Entity Set and Reliances packages form the core of the pattern definition capabilities of PHORML. The Required Entity Set defines the minimum set of conceptual entities that conform to accepted structural entities in an implementation, or abstract entities in a design. This set is both necessary and sufficient to model any object-oriented element of any object-oriented language, and can be used to successfully model procedural systems as well. The Reliances package in turn defines the necessary and sufficient relationships that are not already defined within the Required Entity Set.

The formal foundations of PHORML are defined in the following documents. These are non-normative, however, and are informative only. Other relevant documents can be found in Annex D, Bibliography.

- *A Theory of Objects*, Martin Abadi and Luca Cardelli, Springer-Verlag, 1996.
- *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code*, Jason McC. Smith, Ph.D. Dissertation, University of North Carolina, 2005

By basing PHORML on a strong and rigorous formal foundation, simplicity is possible. PHORML avoids the *ad hoc* approaches of most patterns modeling frameworks, by providing all necessary atomic elements from which to describe the interactions among object-oriented programming and design elements. Without formality, it is impossible to describe software patterns rigorously, and without rigor the resulting software descriptions are equivalent to defining little at all.

PHORML works in concert with the ASTM metamodel and OCL constraint language through the optional APML package described in Annex C. PHORML does not attempt to replicate existing procedural or low-level source code execution. Instead it provides a higher level conceptual framework which can be annotated with ASTM tree fragments, OCL constraints, or other appropriate well-formed formal notations as necessary, at defined extension points. This continues the necessary rigor in ways that are flexible and extensible.

This page intentionally left blank.

14 PHORML::Core Classes (Informative)

The PHORML::Core package defines the necessary elements for the rest of PHORML. Figure 14.1 shows the primary three classes defined in this package suite.

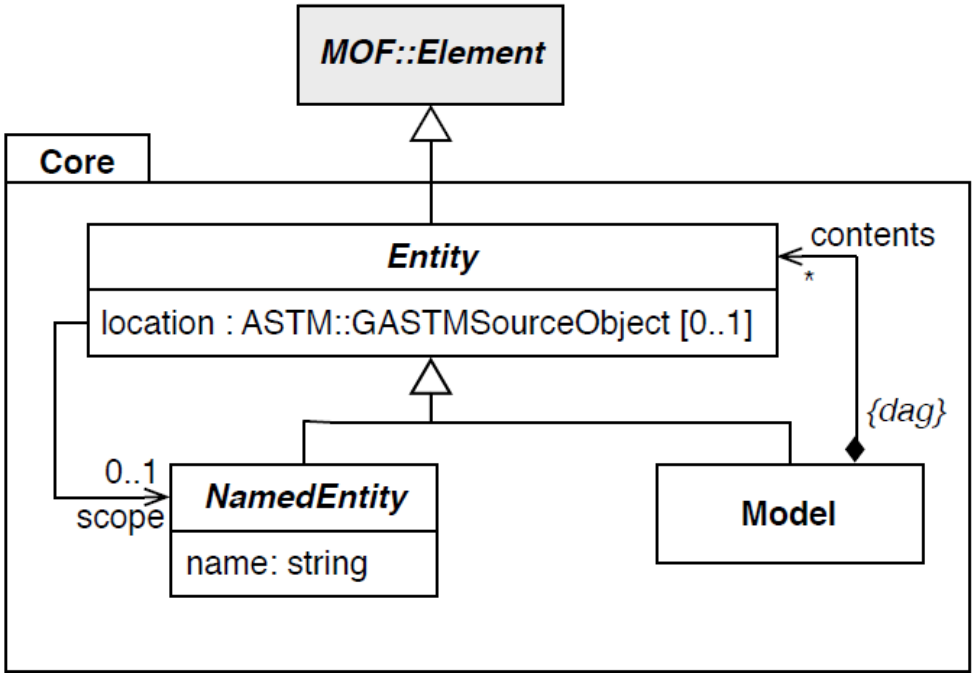

Figure 14.1 - Core Package

14.1 Entity (Abstract)

The base class for almost every class in the PHORML specification, Entity is an abstract subclass of MOF::Element. Entities have an associated instance of the Location class for traceability and tracking to and from concrete elements that they represent. PHORML::Entity instances also have an optional *scope* that provides a reference to an enclosing Entity that may contain them. Any enclosing Entity must be named for scoping to be logically consistent and accessible.

Generalizations

None

Associations

scope : Entity [0..1]	An optional scope that this Entity is defined within.
location : ASTM::GASTMSourceObject [0..1]	An optional location to assist with traceability.

14.2 Model

Model is an entity that represents a model expressed in PHORML, and contains zero or more PHORML::Core::Entity instances that form the definition of the model.

Generalizations

PHORML::Core::Entity

Associations

contents : Entity [*]	Contains the contents of the Model.

14.3 NamedEntity (Abstract)

NamedEntity is an abstract subclass of Entity with an associated name, provided as a value of String.

Generalizations

PHORML::Core::Entity

Attributes

name : String	Specifies the name of the Entity.

15 PHORML::RequiredEntitySet Classes (Informative)

15.1 Introduction

The RequiredEntitySet (RES), as shown in Figure 15.1, is that set of object-oriented programming concepts which are minimal, necessary and sufficient for portraying object-oriented language constructs. In an effort to keep the complexity of PHORML to an absolute minimum, the semantics of the sigma-calculus by Abadi and Cardelli have been adopted. These semantics provide four concrete entity concepts from which all aspect of object-oriented languages can be constructed. These entities are objects, methods, fields, and types. Classes are constructed from types and objects, namespaces and packages are analogous to objects, and so on. The proof of the necessary and sufficient nature of this required set is beyond the scope of this document but can be found in *A Theory of Objects*, Martin Abadi and Luca Cardelli, Springer-Verlag, 1996.

Tools and implementations may provide entities above and beyond those found in this set for efficiency of depiction to a user, or storage concerns. These extensions must, however, have a well formed and specific derivation from the entities defined in this section. Example extensions that are likely to be commonly requested can be found in Annex A.

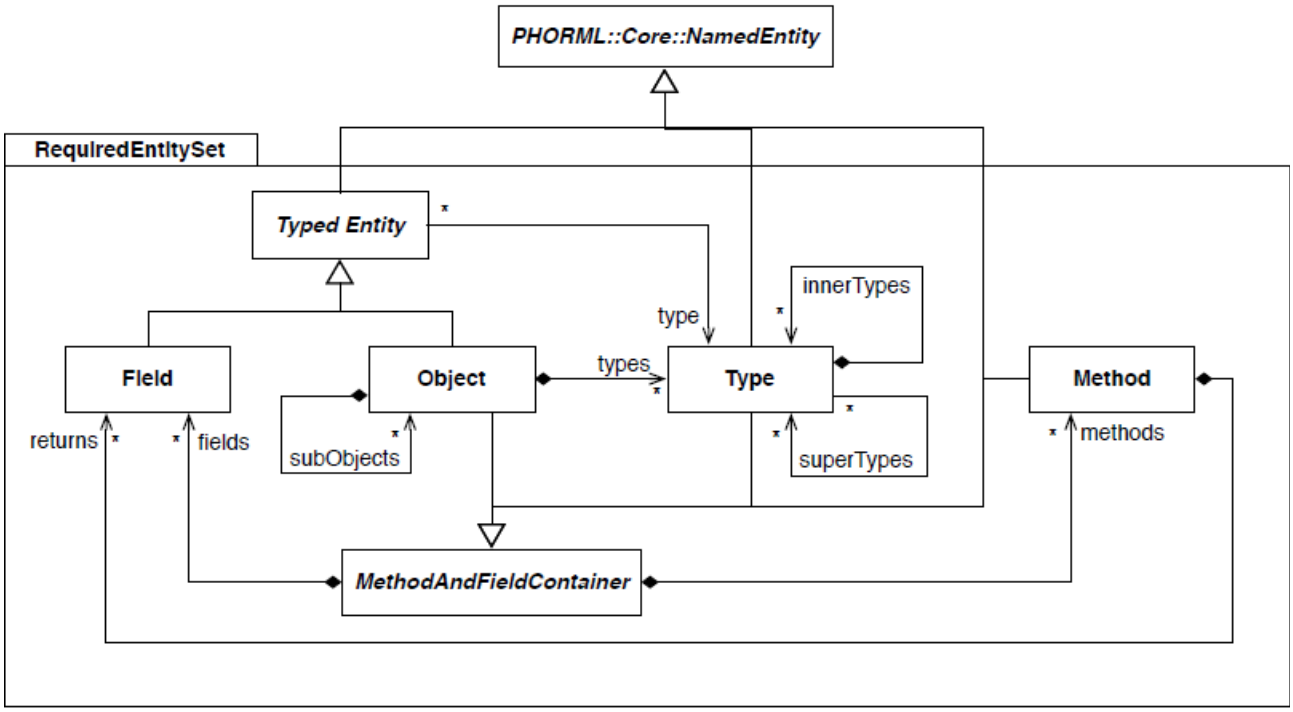

Figure 15.1 - Required Entity Set

15.2 TypedEntity (Abstract)

The TypedEntity class is an abstract class that provides a base concept for any entity that has a type. Any TypedEntity will necessarily be a NamedEntity and is subclassed from PHORML::Core::NamedEntity. TypedEntity defines a single attribute *type* which references an instance of the Type class.

Generalizations

PHORML::Core::NamedEntity

Associations

type : Type [1] A reference to the Type of the Entity.

15.3 MethodAndFieldContainer (Abstract)

MethodAndFieldContainer is an abstract class provided only as a convenience for the purposes of this document. It only defines a class that contains zero or more instances of the Method and Field classes below, and exists merely to simplify the diagram in Figure 15.1. The container semantics imply ownership through composition, and, in fact, this defines a scoping mechanism. The *scope* attribute of PHORML::Core::Entity is the reflexive form of this.

Generalizations

None

Associations

methods : Method [*] Methods defined within the scope of the Object.

fields : Field [*] Fields defined within the scope of the Object.

15.4 Object

The Object class describes a fully instantiated 'live' object in PHORML. Namespaces, packages, and the like are considered live objects at the time of runtime initialization. Objects are a subclass of both TypedEntity, and MethodAndFieldContainer. In addition to Methods and Fields, Objects may contain Type definitions, as well as other Object definitions. Objects are the most general kind of container Entity in PHORML. As with MethodAndFieldContainer, these 'contains' relationships are given ownership semantics indicated in reflexive form by PHORML::Core::Entity::scope.

Generalizations

TypedEntity

MethodAndFieldContainer

Associations

types : Type [*] Types defined within the scope of the Object.

subObjects : Object [*] Objects defined within the scope of the Object.

15.5 Method

The Method class is a subclass of MethodAndFieldContainer and NamedEntity. A Method may, in some languages, define inner methods, and almost all methods define private fields for data storage. A Method has zero or more *return* attributes which are instances of Field.

PHORML Methods are not TypedEntities, despite the usual assumption of the return type of a method being the 'type of' the method. This only holds true in general for procedural languages, and is not true of object-oriented languages, particularly as defined by sigma-calculus. The enclosing scope of the method definition, such as an object, or a type, also determines the 'type' of the method. In languages that support overloading, the types of the arguments to the method are also considered. It is for these reasons that the Method class is not a TypedEntity subclass.

Generalizations

PHORML::Core::NamedEntity

MethodAndFieldContainer

Associations

returns : Field [*] Fields used to return one or more values to a calling scope.

15.6 Field

The Field class is a TypedEntity. A Field is not an *in situ* definition of an object, as Object is. It is an instance of its Type class, instantiated during execution of a system, as opposed to Objects which are almost always instantiated *prior* to execution. Fields do not, by themselves, contain other Fields, Methods, Objects or Types. Their associated Type definition establishes these.

Generalizations

TypedEntity

15.7 Type

The Type class is a subclass of both MethodAndFieldContainer and NamedEntity. It may define child Methods, Fields, or other Types. Again, as with Object, these child definitions provided under ownership semantics, and reflected in PHORML::Core::Entity::scope. A Type may have supertypes that it inherits or subtypes from, as per IsA semantics.

Generalizations

PHORML::Core::NamedEntity

MethodAndFieldContainer

Associations

innerTypes : Type [*] Types defined within the scope of the Object.

superTypes : Type [*] Types that this type subclasses from.

This page intentionally left blank.

16 PHORML::Reliances Classes (Informative)

16.1 Introduction

Reliances are the core of PHORML in many respects. Where the RequiredEntitySet package establishes the structural arrangement of the Entities in a Model, the Reliance package defines the various non-structural non-scoping relationships that exist between them. As with the RequiredEntitySet package, this is designed to be a minimalist yet complete set of concepts.

Given our base assertion that the four concrete Entity classes defined in the RequiredEntitySet form a necessary *and* sufficient set of concepts for representing the constructs of object-oriented systems, then it can be quickly argued that we have, between the RequiredEntitySet and Reliance packages, a necessary and sufficient set of relationships between those concepts. This is far from a proof, for such a discussion, see *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code*, Jason McC. Smith, Ph.D. Dissertation, University of North Carolina, 2005.

There are four concrete Entity classes: Objects, Methods, Fields, and Types. Each concrete class can interact with each of the other classes. Table 16.1 shows the possible interactions, to be read as the Entity in the leftmost column has the relationships to the Entity in the topmost column defined by their intersection (i.e., a Type Defines a Method). All sixteen can be listed as either structural, i.e., scoping through the graph defined by their child and *scope* attributes, or relational. We have covered the scoping interactions above in Required Entity Set, eliminating six interactions listed in Table 16.1 as Defines. For instance, an Object can contain, and therefore Define, other Objects (think of nested namespaces), Methods, Fields, or new Types. A Type can contain, and therefore Define, Methods and Fields. Likewise, the TypedEntity class embodies the IsOfType aspects, such as a Field being IsOfType of a defined Type, or a Type being defined by an Object that is a prototype. Subtyping is handled directly within the Type class. The three entries listed as N/A are those that are simply not supported by the core semantics of sigma-calculus. This leaves us with four relationships, the ones in shaded boxes in Table 16.1. These are the classes represented in the Reliances package.

Table 16.1: Possible RequiredEntitySet Interactions

	Object	**Method**	**Field**	**Type**
Object	Defines	Defines	Defines	Defines
Method	N/A	Invocation	Field Use	N/A
Field	N/A	State Change	Cohesion	IsOfType
Type	IsOfType	Defines	Defines	Subtype

For the purposes of Figure 16.1 and the following discussion, RequiredEntitySet will be abbreviated as RES.

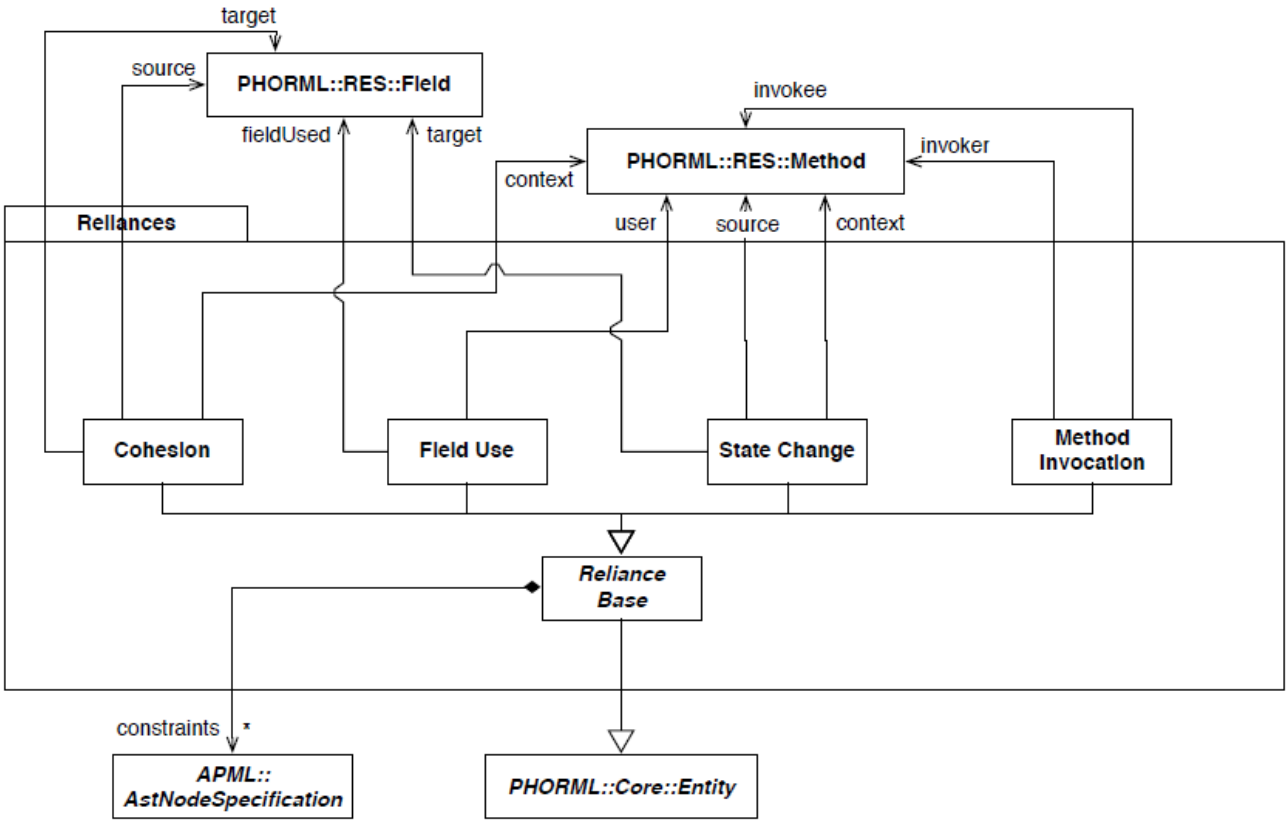

Figure 16.1 - Reliances package

16.2 RelianceBase

Base class for all Reliance classes, it is a subclass of Core::Entity, and defines a transitive relationship.

One or more constraints supply a conditional or constraint onto a Reliance, providing further information on where and when it is applicable. A constraint is an instance of an element from the APML described in Annex C.

Note on normative conformance: A tool may provide APML support, or not, and be baseline compliant with the core of PHORML. It will simply be less capable at expressing specific types of reliances. In such cases, the tool should just ignore the constraints. It is suggested that APML be adopted by most automated tools, but not normative to PHORML compliance.

Generalizations

PHORML::Core::Entity

Associations

constraints : APML::AstNodeSpecification [*] Fine-grained constraints imposed on reliances within SPMS.

16.3 Method Invocation

Method Invocation is in its degenerate form a direct method call. Since it is transitive, it is a convenient way to collapse entire calling chains into simple representations.

Generalizations

RelianceBase

Associations

invoker : PHORML::RES::Method The Method within which the method invocation is initiated.

Iinvokee : PHORML::RES::Method The Method being invoked.

16.4 Field Use

Field Use is when a Method uses the value of a Field that it has not defined, for instance through the argument parameters, or access to a global data pool.

Generalizations

RelianceBase

Associations

user : PHORML::RES::Method The Method within which the Field is being accessed.

fieldUsed : PHORML::RES::Field The Field being accessed.

16.5 State Change

State Change defines when the value of a Field relies on the behavior or value returned by a Method. The simplest form of this is an assignment such as `f = a();`. The Field `f` relies on the return value of the Method `a`. Since this necessarily requires a Method in which an executable statement to occur, State Change has a *context* attribute that specifies the Method and necessary scoping instance.

Generalizations

RelianceBase

Associations

context : PHORML::RES::Method The Method within which the StateChange is occurring.

source : PHORML::RES::Method The Method providing the behavior or value.

target : PHORML::RES::Field The Field whose state is being altered.

16.6 Cohesion

Cohesion is the process of one Field relying on another for its value. The simplest form is an assignment such as `f = g;`. The Field `f` relies on `g` through Cohesion. As with StateChange, this necessarily must occur within a Method body, and Cohesion has a *context* attribute to indicate this.

Generalizations

RelianceBase

Associations

context : PHORML::RES::Method	The Method within which the StateChange is occurring.
source : PHORML::RES::Field	The Field providing the behavior or value.
target : PHORML::RES::Field	The Field whose state is being altered.

Annex A: Entity Extension Examples

(normative)

A.1 Introduction

This Annex defines informative extensions to the Required Entity Set package. These are provided both as an example of how to define an EntityExtension, and because they are the most likely desired extensions.

A.2 Namespace of Package

A namespace in C++ or other languages can be emulated by simply using an instance of RES::Object. No additional semantics are required.

Static elements of a namespace in C++ are considered private to that namespace. Use an appropriate privacy control as required.

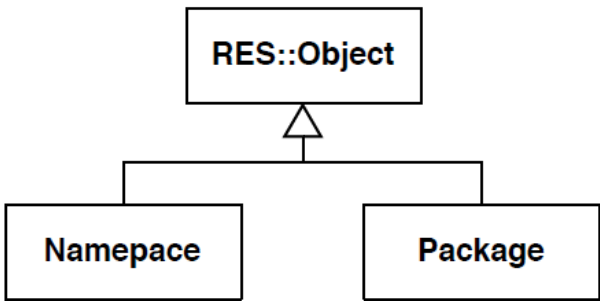

Figure A.1 - Namespace and Package Object

A.3 Class

A class in class-based object-oriented languages such as C++ and Java can be constructed by pairing two RES entities, one Type, and one Object. The instance members of the class are defined in a Type. The class-owned (such as indicated by static in C++ or Java) members of the class are placed in a corresponding *class object*. This class object is considered live at the start of execution of a system and is therefore an example of an Object. Use composition to indicate ownership of the class object by the Class.

Annex B: Procedural Language Modeling

(informative)

B.1 Introduction

SPMS is not limited to just object-oriented languages, despite the object focus. It has been successfully used to model pure procedural systems, such as those written in C, including semantic and idiomatic analysis identifying instances of design patterns from an appropriate library. This Annex outlines one way in which SPMS, leveraging the modeling system defined by PHORML, can be used to model a system implemented in C. No modifications to procedural source code need be performed to enable this modeling or further analysis. This is simply a unique view of the system.

B.2 Sample Code

Procedural languages have no enclosing objects surrounding functions. The functions are free-floating, and global in scope. This global scoping is easily modeling using an instance of Object named, simply, Global, in which all functions are placed. Similarly, global data can be placed within the same Global instance.

B.3 Directories as Namespaces

One common idiom in procedural programming is to use a directory layout within a file system as a way of partitioning code for human understanding. Functionality related to a specific topic will reside within a directory, related directories will be compiled and bound into a library, and so on. We can leverage this conceptual partitioning by recognizing that this approach is extremely similar to the use of namespaces and packages in language such as, respectively, C++ and Java. (Java packages even use the file layout explicitly.)

Namespaces, as defined in Annex A, can therefore be used in lieu of directory structures to provide insight to the partitioning used within an existing system.

B.4 File-static Functions and Data

Much as namespaces are used to model directory-specified conceptual partitioning, file-static elements in C, such as functions, and data, are used to hide them from the global scope. They are 'owned' within a particular file. This indicates that we can model this by providing an Object instance for each file or compilation unit within a system, responsible for ownership and scoping of file-static elements.

B.5 Structs as Classes

We can look to how C++ handles C-style structs for guidance. In C++, a struct is simply a class with no member functions. We model this the same way in our PHORML representation of C: a struct becomes a Class.

This page intentionally left blank.

Annex C: AST-Based Pattern Metamodel Language (APML)

(informative)

C.1 Overview

As part of PHORML, the AST-Based Pattern Modeling Language (APML) helps in defining and describing the code conditions at the body-level that are necessary provide a formal and complete definition of patterns. It is as independent as possible of software implementation, since it uses the Abstract Syntax Tree Metamodel (ASTM) and Object Constraint Language (OCL) standards. The AST-Based Pattern Modeling Language supports formal description of good and bad practices in programming.

APML is composed of two packages: Geometry package and Constraint package. The Geometry package describes the geometry of the expected tree, and the Constraint package constraint the content of the matched tree. Figure C.1 Illustrates these packages, and the relations with OCL and ASTM.

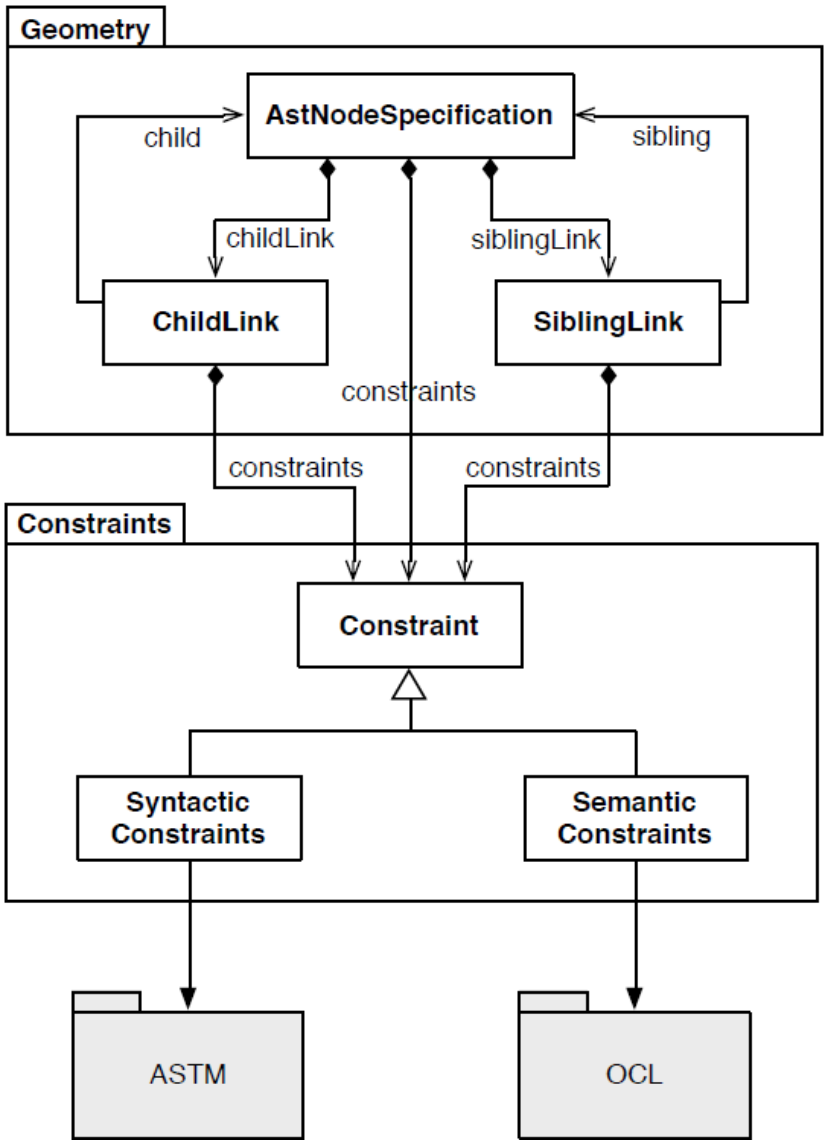

Figure C.1 - High Level (Composite) Diagram

C.2 Geometry

The geometry package describes the geometry of the expected tree.

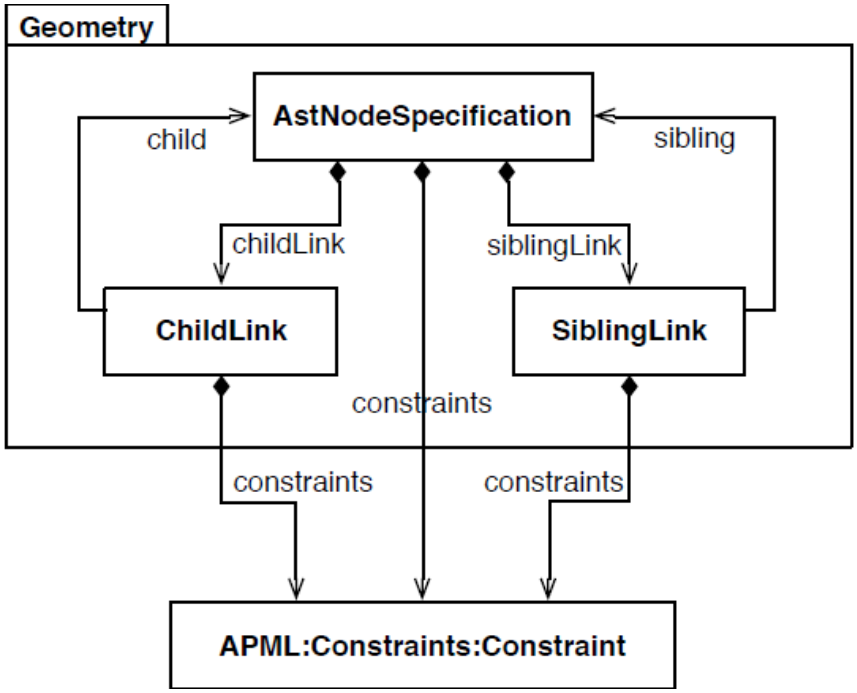

Figure C.2 - Geometry Package Diagram

Ast Node Specification Class

The AstNodeSpecification Class defines a node of a single AST node. The n-ary tree definition is made with two relations: ChildLink and SiblingLink.

Associations

constraints : Constraint [1..n]	Constraints on the current AstNode
childLink : ChildLink [0..1]	Relation to a child node
siblingLink : SiblinkLink [0..1]	Relation to a sibling node

ChildLink class

The ChildLink Class defines a relation to an expected child node. The relation concerns a direct or indirect parentship. In other words, the expected node could be a child, or a grandchild, and so on.

Associations

constraints : Constraint [0..n]	Constraints on all nodes in the partnership
child : AstNodeSpecification [0..1]	Relation to the expected child node

SiblingLink class

The SiblingLink Class defines a relation to an expected sibling. The relation concerns a direct or indirect sibling. In other words, the expected node could be the next sibling, or the next of the next sibling, and so on.

Associations

constraints : Constraint [0..n]	Constraints on all nodes in the partnership
sibling : AstNodeSpecification [0..1]	Relation to the expected sibling node

C.3 Constraints

The constraint package defines constraints on expected nodes.

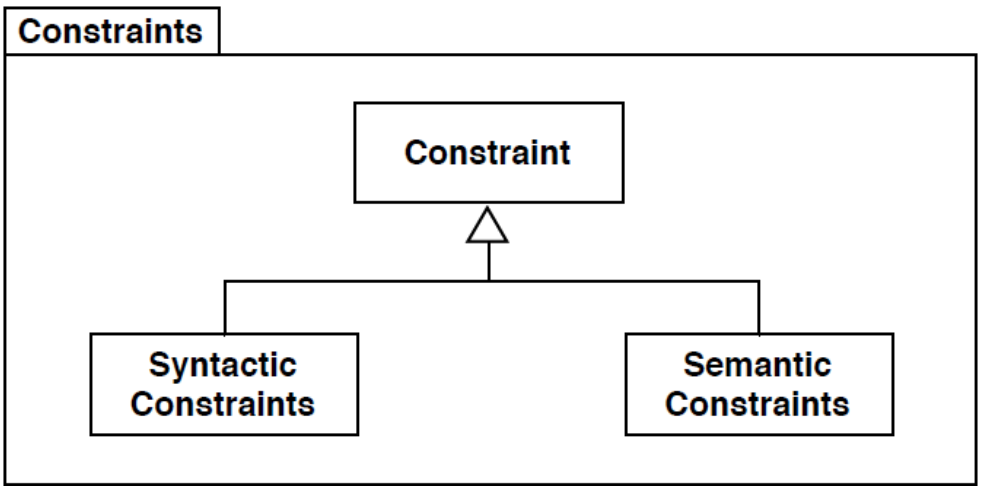

Figure C.3 - Constraint Package Diagram

The Constraint package is the minimal language to define what expected AST nodes have to fulfill. The goal is to avoid defining a whole language with statements and expressions.

Constraints are intended to be extensible. However, many of them are so frequent and useful that they can belong to a base library.

Types of Constraints:

- Syntactic constraints: constraints on the expected AST node
- Semantic constraints: constraints on the resolved symbol obtained on the decorated AST

C.3.1 Syntactic constraints

Constraints on the expected AST node.

AstNodeTypeMustInherits(Type t):

Means it must inherits a certain type. The expected type is based on GASTMSyntaxObject.

AstNodePropertyEquals(Property property, object value):

Means it must have a property equals to a certain value.

C.3.2 SemanticConstraints

Constraints on the resolved symbol obtained on the decorated Ast.

ResolvedSymbolInheritsOrEquals(Type t):

means it must inherits a certain type. The expected type is based on UML::Core::Datatype.

ResolvedSymbolOverrideOrEquals(Method m):

means it must inherits a certain type. The expected type is based on UML::Core::Constructs::Operation.

ResolvedSymbolPropertyEquals(PropertyInfo property, object value)

Means it must have a property equals to a certain value.

This page intentionally left blank.

Annex D: Bibliography

(informative)

- *A Theory of Objects*, Martin Abadi and Luca Cardelli, Springer-Verlag, 1996.
- *Patterns-Based Engineering: Successfully Delivering Solutions via Patterns*, Lee Ackerman and Celso Gonzalez, Addison-Wesley Professional Publishing, 2010.
- *Notes on the Synthesis of Form*, Christopher Alexander, Harvard University Press, 1964.
- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm. Ralph Johnson, and John Vlissides, Addison-Wesley Professional Publishing, 1994.
- *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code*, Jason McC. Smith, Ph.D. Dissertation, University of North Carolina, 2005
- *The Pattern Instance Notation: A Simple Hierarchical Visual Notation for the Dynamic Visualization and Comprehension of Software Patterns*, Jason McC. Smith, The Journal of Visual Languages and Computing, Elsevier Publishing, Vol 22, Issue 5, Oct 2011.
- *Elemental Design Patterns*, Jason McC. Smith, Addison-Wesley Professional Publishing, Mar 2012.
- *The Object Recursion Pattern*, in: N. Harrison, B. Foote, H. Rohnert (Eds.), *Pattern Languages of Program Design 4*, Bobby Woolf, Addison-Wesley, 1998
- *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Frank Buschmann et al, Wiley & Sons, 1996.
- *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Douglas Schmidt et al, Wiley & Sons, 2000.
- *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*, Michael Kircher and Prashant Jain, Wiley & Sons, 2004.
- *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Frank Buschmann et al, Wiley & Sons, 2007.
- *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, Frank Buschmann et al, Wiley & Sons, 2007.

This page intentionally left blank.