Developing Web Applications With Object-Oriented Approaches and Object-Process Methodology

Iris Reinhartz-Berger

Developing Web Applications With Object-Oriented Approaches and Object-Process Methodology

Research Thesis

Submitted in Partial Fulfillment of the

Requirements for the

Degree of Doctor of Philosophy

Iris Reinhartz-Berger

Submitted to the Senate of

the Technion – Israel Institute of Technology

TAMOOZ 5763 HAIFA JULY 2003

The research thesisw as done under

the supervision of Prof. Dov Dori and Prof. Shmuel Katz

in the faculty of Industrial Engineering and Management.

The generous financial help of the Technion is gratefully acknowledged.

List of Content:

List of Figures:

List of Tables:

Abstract

The exponential growth of the Web during the last two decades and its expected spread in the next years has set the stage for increasing use of Web applications. Web applications, which can be classified as hybrids between hypermedia and information systems, have a relatively simple distributed architecture from the user viewpoint but a complex dynamic architecture from the designer viewpoint. They need to respond to operation by an unlimited number of heterogeneously skilled users, address security and privacy concerns, access heterogeneous, up-to-date information sources, and exhibit dynamic behaviors that involve such processes as code transferring. Common system development methods can model some of these aspects, but none of them is sufficient to specify the large spectrum of Web application concepts and requirements. The main reason for this is that these techniques are either structure- or behavior-oriented, but not both. This work presents the development and evaluation of OPM/Web, which is an extension to the Object-Process Methodology (OPM) that satisfies the functional, structural and behavioral Web-based information system requirements in a single frame of reference.

The work consists of five parts. The first part reviews Web applications modeling needs and critiques existing development techniques in this field. It also define the research goals and methodology. The second part presents OPM/Web extensions. The main extensions of OPM/Web with respect to OPM are: (1) adding properties to links to express such requirements as encryption; (2) extending zooming and unfolding facilities to increase modularity within a single model and to improve component reuse; (3) defining components and weaving rules for their composition; (4) separating declarations of process classes (representing code) from their instances to model code migration; and (5) adding global data integrity and control constraints to express dependence or temporal relations among (physically) separate modules.

The third part evaluates OPM/Web by experimentally comparing it to a Web extension of UML, the standard object-oriented modeling language. The experiment, whose subjects were third year undergraduate information systems engineering students, included comprehension and construction questions about two representative Web application models. OPM/Web was found to be more comprehensible when answering questions about the system dynamics and distribution and easier to use when constructing or extending existing models. No significant differences were found with respect to comprehension of the system's structure.

In the forth part of the work, OPM/Web is formally metamodeled using OPM ontology and notation. This metamodel captures the ontology and notation of OPM/Web, as well as a generic OPM-based development process. This capability of OPM (and OPM/Web) to model itself is indicative to its expressive power.

In the fifth, final part of the work, I summarize the contribution of this research, its application in OPCAT, the Object-Process CASE Tool, and refer to implementation issues.

Symbols and Abbreviations List

Code	Full Name
AOD	Aspect-Oriented Design
AOP	Aspect-Oriented Programming
CAME	Computer Aided Method Engineering
CASE	Computer Aided Software Engineering
CBD	Component-Based Development
COD	Code On Demand
CORBA	Common Object Request Broker Architecture
COTS	Commercial Of-The Shelf
CS	Client-Server
DB	Database
DBMS	Database Management System
DFD	Data Flow Diagram
ECA	Event – Condition – Action
EER	Extended Entity Relationship
EORM	Enhanced Object Relationship Model
ER	Entity-Relation
GUI	Graphical User Interface
HDM	Hypertext Design Model
HTML	HyperText Markup Language
MA	Mobile Agent
MDA	Model-Driven Architecture
MMF	Meta Modeling Facility
MOF	Meta Object Facility
NIAM	Nijssen Information Analysis Method
OCL	Object Constraint Language
OML	OPEN Modeling Language
OMT	Object Model Technique
OO	Object-Oriented
OOHDM	Object-Oriented Hypertext Design Model
OPCAT	Object-Process CASE Tool
OPD	Object-Process Diagram
OPEN	Object Process, Environment, and Notation
OPL	Object-Process Language
OPM	Object-Process Methodology
ORM	Object Role Model
RDF	Resource Description Framework
REV	Remote Evaluation
RMM	Relationship Management Methodology
RPC	Remote Procedure Call
RUP	Rational Unified Process
UML	Unified Modeling Language
WebML	Web Modeling Language
WSDL	Web Services Description Language
XML	eXtendible Markup Language

Part 1. Introduction and Background

1. Introduction

The exponential growth of the Web and the progress of Internet-based architectures have set the stage for the sprouting of Web applications. Web applications are diverse and include electronic catalogs with secure and on-line transaction processing, real-estate listing services, inventory management systems, private membership services, job matching applications, interactive discussion groups and chat rooms, interactive training, internet commerce solutions, and many more. Built on the foundations of the World Wide Web, Web applications, which provide a rich interactive environment, are completely cross-platform and can be accessed anywhere in the world at any time. The only client-side software needed to access and execute Web applications is a Web browser environment. In recent years, the World Wide Web has become the platform of choice for developing distributed systems. This preference is based on the Web's powerful browsing communication paradigm and on its open architectural standards, which facilitate the integration of different types of content and systems [35].

Web applications can be classified as hybrids between hypermedia and information systems [66]. Like hypermedia, which is based on computer-addressable files that contain links to multimedia information (e.g., text, graphics, video, or audio), information on the Web is accessed in an exploratory way rather than through “canned" interfaces. The way in which the navigation is done and presented in Web applications is therefore of prominent importance. Like information systems, the size and volatility of data and the distribution of applications requires consolidated architectural solutions based on such technologies as database management systems and client-server computing.

Due to the hybrid nature of Web applications, methodologies and software tools from both hypermedia and information systems may greatly assist mastering the complexity of these

applications. The three major design dimensions that characterize many Web applications are:

1. The structure that describes the organization of the information managed by the application, as well as its user interface;
2. The behavior that concerns the facilities for accessing information, manipulating it, and navigating across the application content; and
3. The architecture that specified the static and dynamic distribution of the application.

Hypermedia authoring techniques [66], which help design hypermedia applications, focus on modeling the structure, content, and navigation of a system, but they tend to neglect the architecture and behavior of many complex, dynamic Web applications. Hence, these techniques are primarily suitable for modeling and developing content-rich applications that are published once and are hardly changed.

In the information systems area, the object-oriented paradigm and the Unified Modeling Language (UML) [68] are commonly used. UML, which has become the industrial standard of the object-oriented methods, models the system structure through classes, while system behavior is expressed through services and message passing between objects. The static system architecture is modeled in yet another view. The use of the multi-view in UML and the lack of clear, separate mechanism for specifying processes weaken UML models integrity and expressiveness [24, 78, 87].

Hypermedia authoring techniques and information system development methods model some Web application requirements, but each one of them on its own, or even a combination of them, is not sufficient for specifying all the aspects of Web applications. This work extends the Object-Process Methodology (OPM) to respond to Web application domain needs. The motivation for this extension, which is called OPM/Web, its features, its

evaluation and comparison to the standard UML, and its formal metamodel are the focus of this research.

2. Literature Review: Web Application Features and Modeling Methods

2.1 Web Application Features

Although the variety of Web applications is very wide, several features, which are listed below, are common to most Web applications [13, 35, 58].

Complex dynamic and distributed architecture

The general architecture of Web applications is client-server, which consists of three components: one or more Web servers, a network connection (mainly http), and client browsers. A contrast exists between a simplistic user perspective, which views Web applications as resident on a server and accessed remotely, and a relatively complex designer's viewpoint of architectural reality with many structural and behavioral aspects [1]. Unlike stand-alone systems, in which architecture is known at compilation time and does not change, most Web applications feature dynamic architecture, which can change and evolve during run-time. Therefore, a specification method for Web applications should support modeling logical distributed elements, their bindings to physical elements, and algorithms for detecting system architecture and its dynamic changes during run-time.

Unlimited, heterogeneously skilled users

The Web allows universal access to its applications for an unlimited number of users of all skill levels. In addition, the competition among different solutions (Web sites) for the same user segment emphasizes user interface design as one of the first priorities. Any modeling technique for Web applications should therefore meet two main user interface requirements: expressiveness and complexity management. Expressiveness means that the technique should be able to model all the applicable structures, such as complex GUI classes of Java or

navigational capabilities of HTML. Moreover, since the developer obviously cannot represent every single Web object in a flat user interface, a variety of techniques are required to reduce the clutter. Some useful techniques for complexity management are aggregation, summarization, filtering, and example-based representations [67].

Security and privacy support

Although the Web is reachable by any connected user around the world, Web applications potentially deal with private information or restricted user group information. Therefore, Web applications should prevent the users from causing (either intentionally or unintentionally) harm to the systems, by security and privacy management [31]. Recent attacks and defacing of Web sites underscore this need. ISO 7498-2 (1989) defines five main categories of security services: authentication, access control, confidentiality (privacy), data integrity, and non-repudiation (of the transaction by the performer). These types of security services should be specified in the early stages of system development, i.e. during the analysis and design phases.

Heterogeneity of information sources

Web applications must handle and integrate heavy, complex, hierarchical data, as well as unstructured or semi-structured data. These data can reside directly in documents and might be static (text, pictures, etc.), dynamic (Java applets, HTML forms, etc.), or linking components. The data may also be stored in different systems and distributed over multiple sites, possibly through a distributed DBMS, which increases security, provides for shared access, and boosts performance. A Web application modeling method should be able to specify this variety of information sources.

Up-to-date Information

Updates of data input into Web applications are done in real-time or near real-time. Moreover, some Web applications should run continuously, forcing the developer to be able

to add new constructs and functionality without disturbing the working version. One way of facilitating concurrent updating is to apply modularity, which is essential for maximizing conceptual clarity, modifiability, understandability, and reusability of the specifications. Modular programs are easier to develop and test, especially for a team of designers and programmers. They are also easier to understand and maintain, because certain changes do not require extensive modifications or re-testing of the entire application and can be implemented locally.

Dynamic behavior

Most Web applications are dynamic. They access data, manipulate it, ask the server for code, verify different types of constraints, and produce results for the users or updates for the server. The behavior of Web applications results from the trade-off between evolving functionality requirements and existing bandwidth limitations. The dynamic behavior can be exhibited through animations, dynamic presentations, or filling in interactive forms, where executing code at the client side is required. In order to be able to reuse specifications of Web applications, modeling methods should support specifying conceptual Web terms separately from the technical solutions. An example is modeling code transferring processes that can be implemented as Java applets or mobile agents.

2.2 Current Web Application Development Techniques

As noted, existing system and software development approaches can be divided into two groups: hypermedia authoring techniques and information system development methods. The later group can be further divided into object-oriented approaches, behavior-oriented ones, and hybrids of the two.

2.2.1 Hypermedia Authoring Techniques

Hypermedia authoring techniques are based either on the Entity-Relation (ER) model (e.g., HDM, RMM, and WebML) or on the object-oriented model (OOHDM or EORM). Employing the hypermedia authoring approach, the Web application structure, navigation, and presentation are designed.

The **Hypertext Design Model (HDM)** [38, 39] shifts the focus from hypertext data models as a means to capture the structuring primitives of hypertext systems, to hypertext models as a means for capturing the semantics of a hypermedia application domain. It prescribes the definition of an application schema, which specifies classes of information elements in terms of their common presentation characteristics, their internal organization structure, and the types of their mutual interconnections. Web structure is expressed by means of entities, sub-structured into a tree of components. Navigation can be internal to entities (along part-of links), cross-entity (along generalized links), or non-contextual (using access indexes, called collections).

The **Relationship Management Methodology (RMM)** [48] evolves HDM by embedding its hypermedia design concepts into a structured methodology, splitting the development process into seven distinct steps and giving guidelines for the tasks. The development cycle steps in RMM are ER design, slice design, navigation design, conversion protocol design, user-interface design, runtime behavior design, and construction & testing.

The **Web Modeling Language (WebML)** [13] enables specifying the core features of a site at a high level, without committing to low-level architectural details. The specification of a site in WebML consists of four orthogonal perspectives: (1) The structural model which expresses the data content of the site in terms of the relevant entities and relationships; (2) the hypertext model that describes one or more hypertexts that can be published in the site. Each different hypertext defines a so-called site view, each of which consists of two sub-models:

the composition model (i.e., the hypertext that compose the pages) and the navigation model (expressing how pages and content units are linked to form the hypertext); (3) the presentation model that expresses the layout and graphic appearance of pages, independently of the output device and of the rendition language. This model is represented by means of an XML syntax; and (4) the personalization model which defines users, user groups, and permissions.

The **Object-Oriented Hypertext Design Model (OOHDM)** [88, 89] is a direct descendant of HDM. It differs from HDM in its object-oriented nature, and in that it includes special-purpose modeling primitives for both navigational and interface design. OOHDM comprises four different activities: conceptual design, navigational design, abstract interface design, and implementation. During each activity, except for the implementation, a set of object-oriented models describing particular design concerns are built or enriched from previous iterations.

The **Enhanced Object Relationship Model (EORM)** [53] is defined as an iterative process concentrating on the enrichment of the object-oriented model by the representation of relations between objects (links) as objects. The technique is based on three frameworks of reusable libraries: one for class definition, one for composition (link class definition), and one for GUIs. EORM differs from OOHDM mainly in that OOHDM clearly separates navigational from conceptual design concerns by defining different modeling primitives for each step, while EORM combines them all together. The advantages of EORM over OOHDM are that relations become semantically rich as they are extensible constructs, they can participate in other relations, and they can be part of reusable libraries.

All the hypermedia authoring techniques emphasize content and presentation modeling but do not adequately describe complicated behaviors. Since these techniques do not support physical architecture representation and security and privacy management either, they may be

suitable for designing content-rich applications with a low to medium level of dynamic behavior, but not for high-volume, distributed Web-based systems.

2.2.2 Object-Oriented Modeling Methods and Techniques

Object-orientation is a general paradigm for developing systems that focuses on the objects that build the system. The most common object-oriented modeling language is the Unified Modeling Language (UML) [68]. **UML** is a general-purpose visual modeling language, which is applied mainly for specifying, constructing, and documenting the artifacts of software systems. The structural models of UML define the classes of objects important to a system and to its implementation and the relationships among them. The dynamic behavior defines the history of objects over time and the communications among them. Web applications, like other software-intensive systems, are represented by a set of UML models, including a use-case model, a class model, an interaction (collaboration) model, a state transition model, and a deployment model. There are several UML extensions for the Web application domain, including [6, 18, 97]. All of these extensions are based on the UML built-in extension mechanisms, which are tagged values, stereotypes and constraints.

The **Object Process, Environment, and Notation (OPEN)** [57, 75], which is a complete methodology that uses the OPEN Modeling Language (OML) [45] as its notation, offers a set of principles for modeling all aspects of software development across the system lifecycle. The development process is described by a contract-driven lifecycle model, which is complemented by a set of techniques and a formal representation using a modeling language (e.g., OML or UML).

The strength of object-oriented techniques is in modeling the structural aspects of a system. However, they are far less suitable for representing the dynamic and functional aspects of an application. In particular, they do not have a single concept or mechanism for specifying stand-alone processes, which do not belong to a specific object and do not follow the

encapsulation principle. This is a major hindrance to developing Web applications, which, as noted, typically feature complex dynamic behaviors.

2.2.3 Behavior-Oriented Modeling Approaches

The behavior-oriented approaches specify the conditions or constraints that are required to activate some (perhaps partly specified) functionality at a desired moment. Three of the behavior-oriented design techniques, which also address the development of Web applications and distributed systems, are the superimposition approach, Aspect-Oriented Design and the Event-Condition-Action paradigm.

The **superimposition approach** [10, 11, 49] introduces a procedure-like control structure, called a superimposition, that allows convenient expression of the superimposition of one algorithm on another. In this construct, both formal processes and schematic abstractions (called role-types) are declared, each with formal parameters and a sequential communicating algorithm using those parameters. The declaration captures a distributed algorithm that is designed separately, but intended to be executed in conjunction with other activities in the same state space. The construct is combined with an existing collection of communicating processes by instantiating the formal processes and associating each process of the collection with one role-type.

Aspect-Oriented Design (AOD) [2, 50, 93] is an abstraction principle intended to help software developers separate multiple concerns in their design and source code. An aspect modularizes the features for a particular concern and describes how those features should be integrated (woven) into the system design. In object-oriented methods, an aspect is typically spread across multiple methods in multiple classes. In AOD, an aspect can be separated from the classes to which it applies. The aspect-oriented approach emerged through the development of AspectJ [3], a programming language based on Java. This was followed by attempts to percolate this concept to earlier stages in the application development lifecycle,

i.e., to the design stage. Most of the methods relying on this approach use UML. Suzuke and Yamamoto [93], for example, have extended the UML metamodel with another classifier element (in addition to Class, Interface, Component, and Node), called **Aspect**. Kersten and Murphy [50] have introduced the aspect notion as a new notation – a diamond and an associated rectangle, which consists of four sections: attributes, operations, introduce members, and advice methods.

The **Event Condition-Action Paradigm (ECA)** [14] is used in information systems in general and database systems in particular in order to apply the concept of triggers in specifying the behavior of a system. A trigger is a procedural processing element, stored in a database and executed automatically by the DBMS server under specific conditions. A trigger is composed of three components: Event, Condition and Action. It may also take the form of an "Event Condition then Action else Action" (also called ECAA principle). In Web applications those principles can be used, for example, to model business rules [79], expressed by an XML-likeobject -oriented meta-language.

These approaches incorporate new functionality to existing models and define conditions for the execution of processes. However, they tend to pay less attention to the system's structural organization, the physical architecture representation, and the detailed design of the system dynamics.

2.2.4 Structure- and Behavior-Oriented Approaches

Proponents of both structure- and behavior-oriented approaches have reached the conclusion that focusing on just one aspect of a system while neglecting the other is counterproductive. To remedy this shortcoming, each approach adopted some technique of its counterpart. DFD-based techniques, for example, rely on ER or class diagrams for modeling the static data scheme. UML incorporates several process-oriented diagrams, including interaction and Statecharts, which are state transition diagrams.

The need for adequate representation of both the static and dynamic aspects of a system, while keeping the model as simple as possible, was a prime motivation for the development of Object-Process Methodology (OPM) [23]. OPM is an integrated modeling approach that uses objects, processes, and links to represent the system's structure and behavior in the same diagram type. This way, OPM attempts to solve two of the main deficiencies of the object-oriented model: the complexity management problem and the model multiplicity problem [24, 78].

Although OPM is a promising candidate for modeling distributed systems, such as Web applications, in its current state it is not fully suitable to accurately specify some of the mentioned Web application requirements, such as the dynamic architecture and security and privacy management.

3. Object-Process Methodology (OPM)

The Object-Process Methodology (OPM) [23] is a holistic approach to the modeling, study, and development of systems. It integrates the object-oriented and process-oriented paradigms into a single frame of reference. Structure and behavior, the two major aspects that each system exhibits, co-exist in the same OPM model without highlighting one at the expense of suppressing the other. Contrary to UML and its nine diagram types, OPM shows system's structure, behavior, and architecture in the same diagram type, enabling the expression of mutual relations and effects between them and reinforcing the understanding of a system as a whole.

3.1 OPM Ontology

The elements of OPM ontology are entities (things and states) and links. A *thing* is a generalization of an *object* and a *process* – the two basic building blocks of any OPM-based system model. At any point in time, each object is at some *state*, and object states are

changed through occurrences of processes. Analogically, links can be structural or procedural. *Structural links* express static relations between pairs of objects or processes. Aggregation, generalization, characterization, and instantiation are the four fundamental static relations.

The behavior of a system is manifested in three major ways: (1) processes can transform (generate, consume, or change) things; (2) things can enable processes without being transformed by them; and (3) things can trigger events that (at least potentially, if some conditions are met) invoke processes. Accordingly, a procedural link can be a transformation link, an enabling link, or an event link.

The complexity of an OPM model is controlled through three refinement/abstraction mechanisms: *in-zooming/out-zooming*, in which the entity (primarily a process or a state) being detailed is shown enclosing its constituent elements; *unfolding/folding*, in which the entity (primarily an object) being detailed is shown as the root of a structural graph; and *state expressing/suppressing*, which allows for showing or hiding the possible states of an object. These mechanisms enable OPM to recursively specify a system to any desired level of detail without losing legibility and comprehension of the complete system.

3.2 The Bimodal Graphic-Text Representation of OPM

Two semantically equivalent modalities, one graphic and the other textual, jointly express the same OPM model. A set of inter-related Object-Process Diagrams (OPDs), constitute the graphical, visual OPM formalism. Each OPM element is denoted in an OPD by a symbol, and the OPD syntax specifies correct and consistent ways by which entities can be connected via structural and procedural links. The Object-Process Language (OPL), defined by a grammar, is the textual counterpart modality of the graphical OPD set. OPL is a dual-purpose language, oriented towards humans as well as machines. Catering to human needs, OPL is designed as a constrained subset of English, which serves domain experts and system architects, jointly

engaged in analyzing and designing a system. Every OPD construct is expressed by a semantically equivalent OPL sentence or phrase. Designed also for machine interpretation through a well-defined set of production rules, OPL provides a solid basis for automating the generation of the designed application. This dual representation of OPM increases the processing capability of humans according to an accepted cognitive theory [60].

The OPD formalism goes hand in hand with the OPL in the following meaning: *Anything that is expressed graphically by an OPD is also expressed textually in the corresponding OPL paragraph, and vice versa.* OPCAT, a Java-based Object-Process CASE Tool, automatically translates each OPD into its equivalent OPL paragraph (collection of OPL sentences) and vice versa, as explained in Appendix F.

4. Research Objectives and Methodology

The main objective of this research has been to develop and evaluate a complete methodology that supports the entire life-cycle of distributed systems in general and Web applications in particular. As argued in Section 2.2, none of the hypermedia authoring techniques and the information system development approaches can specify all the features that Web applications require. The main reason for this is that these approaches pertain mostly either to the system's structure or its behavior. Table 1 presents an evaluation of the level of structuredness, modularity and reusability, physical architecture representation, user interface modeling, dynamic behavior modeling, and security and privacy management of these techniques and modeling languages. Based on these criteria, OPM was found as the most suitable basis for a complete Web application development methodology. However, in its basic configuration and scope, OPM still cannot address some of the distribution concerns, such as dynamic architecture, message passing, and consistency constrains, neither can it adequately specify code migration. This research extends OPM to OPM/Web, which

augments OPM with features required to model and develop distributed and Web applications.

Table 1. Comparison of existing modeling approaches to the Web application domain

Criterion	Hypermedia Authoring	Object-Oriented	Behavioral-Oriented	OPM
Level of Structuredness	Average (limited to ER concepts)	Good	Poor (built on top of existing structure)	Good (inherits the object-orientation)
Modularity and Reusability	Poor (does not concern reuse)	Average (concern reuse of structure)	Average+ (does not support partially specified component reuse)	Average+ (inherits the behavioral-orientated approach)
Physical Architecture Representation	Poor (does not concern architecture)	Average (concerns architecture structure only)	Poor (does not concern architecture)	Average (concerns architecture structure only)
User Interface Modeling	Average (concerns navigational interface of hypertext)	Average (concerns user interface classes)	Poor (does not concern user interface)	Average (concerns user interface classes)
Dynamic Behavior Modeling	Poor (concerns navigation, but not behavior)	Average (breaks behaviors into methods/services)	Average (models different functionalities at separate levels)	Good (enables stand-alone processes)
Security and Privacy Management	Poor (does not concern security and privacy, supports personalization)	Average (supports security and privacy through stereotypes)	Poor (does not concern security and privacy)	Average (supports privacy through authorization of agent links)

The research objective is achieved in the following steps:

1. Extending OPM to OPM/Web, which includes improving the complexity management mechanisms of OPM, enabling specifications of the static and dynamic aspects of the system architecture, enabling modeling of system integrity constraints, and improving the reuse of partially specified OPM components. Both OPD notation and OPL grammar are extended to OPM/Web, leaving existing OPM models valid in OPM/Web. OPM/Web

extensions are presented in Part 2 of this work and were published in the Annals of Software Engineering [82], the Computer Software and Applications Conference [84], CAiSE workshop on Data Integration over the Web [81], and IBM workshop on Programming Languages & Development Environments [83].

2. Evaluating OPM/Web expressiveness, comprehension, and adequacy for specifying Web applications and comparing them to the standard UML. Although OPM/Web does not add to OPM concepts, it is important to check its ease of use by untrained designers. The experiment and its outcomes are presented in Part 3.
3. Formalization of OPM/Web through a reflective-metamodel. This metamodel, which is expressed by an OPD-set and equivalent OPL paragraphs, includes OPM ontology and notation definitions, as well as a development process suitable for modeling centric and distributed systems in OPM/Web. The complete metamodel of OPM/Web is described in Part 4. A paper about the metamodel of an OPM-based development process was accepted to the International Conference on Conceptual Modeling ER'2003 [26].

Part 2. OPM/Web – OPM Extensions for Web Application Modeling

Chapter 5 presents an overview of OPM/Web extensions through representative examples, while chapters 6 and 7 elaborate on two main OPM/Web extensions: specifying code mobility and migration applications and improving OPM models reusability. The OPM/Web legend of all the examples is summarized in Appendix A.

5. OPM/Web Overview and Examples[1]

The OPM/Web extensions explained and demonstrated in this chapter concern complexity management, structural architecture representations, link characterization, integrative representation of architecture structure and behavior, and system integrity constraint definition.

5.1 Complexity Management

As noted, OPM manages complex system models through abstraction/refinement mechanisms, which address the main requirements from a methodology: completeness and clarity. Completeness means that the system must be specified to the last relevant detail, while clarity means that the resultant model must be legible and comprehensible. Hence OPM has two scaling (abstraction/refinement) mechanisms: unfolding/folding and in-zooming/out-zooming. The *unfolding/folding* mechanism uses structural relations for refining/abstracting the structural parts of a thing (object or process). For example, in Figure 1(a), the process **P1** is unfolded to expose its parts, processes **P1.1** and **P1.2**, and its feature, object **B1.1**. The *in-zooming*/out-*zooming* mechanism exposes/hides the inner details of a thing within its frame.

[1] An extended version of this section was published in Annals on Software Engineering (ASE) – Special Issue on Object-Oriented Web-based Software Engineering [82].

In Figure 1(b), the process **P1** is in-zoomed, showing the process flow – first **P1.1** is executed and creates object **B1.1**, and then process **P1.2** is activated, consuming object **B1.1**.

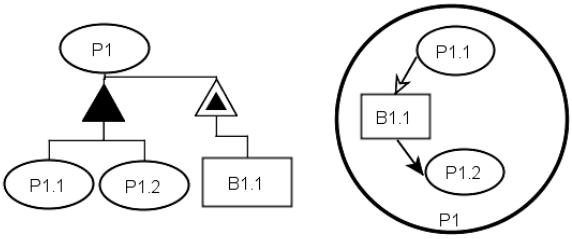


Figure 1. Applying OPM scaling mechanisms to process **P1**.
(a) Process **P1** is unfolded. (b) Process **P1** is in-zoomed.

OPM's abstraction/refinement mechanisms facilitate focusing on a particular subset of things (objects and/or processes), elaborating on their details by drilling down to any desired level of detail. The complexity of the entire system is managed by keeping each Object-Process Diagram (OPD) at a reasonable size and keeping track of the parent-child relationships among the various OPDs in the OPD-set.

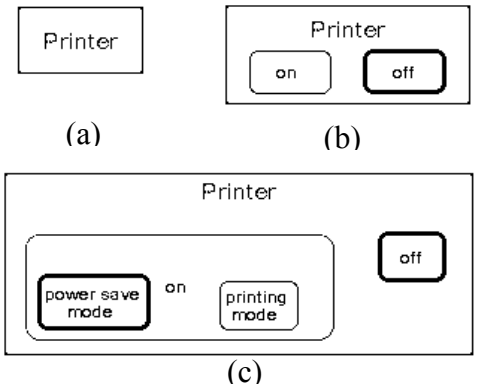


Figure 2. Suppressing and expressing of **Printer** states. State expressing of (a) yields (b) and state expressing of "**on**" in (b) yields (c). State suppressing moves from (c) to (b) to (a).

Following the Statechart approach [44], OPM/Web introduces a hierarchical *state expressing/suppressing* complexity management mechanism. This enables exposing and hiding of object class states and sub-states. For example, Figure 2 depicts the state suppression/expression mechanism for states of a printer. A **Printer** can be in "**on**" or in "**off**".

The state "**off**" is the initial state of the **Printer**. If the **Printer** is in "**on**", it can be in a "**power save mode**" sub-state (which is the initial state of the super state "**on**"), or in a "**printing mode**".

5.2 Structural Architecture Representation and Link Characterizations

OPM and hence OPM/Web can model the structural aspect of the physical architecture using a combination of physical and informatical things (objects and processes). A *physical object* consists of matter and/or energy, is tangible in the broad sense and can be detected by one or more of our senses, while an *informatical object* is a piece of information. A *physical process* is a change that a physical object undergoes. Similarly, an *informatical process* is some transformation (or manipulation) of an informatical object. Figure 3 models a typical architecture of a Web application, in which the **Server** and the **Client** are physical objects. The **Server** stores a database (**DB**), which is an informatical object, while the **Client** stores an informatical object, called **User Profile**, which gathers the system knowledge about the client (the browser type, configuration, etc.).

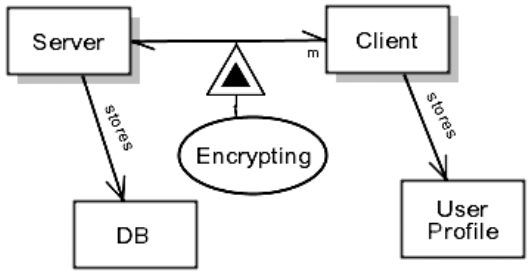


Figure 3. An OPD of a typical Web application architecture

The bi-directional structural relation between the **Server** and the **Client** is physical, since it connects two physical objects. In OPM, there are two different types of links: *structural links*, which specify static aspects of the modeled system and *procedural links*, which capture behavior (such as transformations and events). Both structural and procedural links are characterized only by their types (aggregation, agent, inheritance, etc.) and their multiplicity (e.g., one, many, between 2 to 5). This limits the ability to model certain link properties, such

as the encryption algorithm associated with a physical structural link and sets of possible user activities (a mouse clicking, a button pressing, etc.) associated with a procedural link. OPM/Web supports link characterizations that can associate with each link type any number of features (attributes, which are objects, and/or operations, which are processes). In Figure 3, for example, the **Encrypting** process class is an operation of the physical structural link between the **Server** and the **Client**. It can be further in-zoomed to specify a particular encryption algorithm.

5.3 Integrating the Representation of Architecture Structure and Behavior

As discussed in Section 2.1, modeling the static-structural aspects of a system is not enough. Essential dynamic aspects of the architecture, such as dynamic net programming constructs, must also be modeled. The best known and most prevalent such constructs are Java applets. An *applet* is a small program that is transmitted to be embedded inside another application and run within it. An applet can be sent to a user either separately or along with a Web page and run on the client computer without having to send a user request back to the server. Other known constructs are worms and cookies. *Worms* are computer programs that replicate themselves and are self-propagating. *Cookies* are special pieces of information about the user, which are stored in text files on the client hard disk and can be accessed by the server when the client connects to a Web site that requires this information, while associated code in the client can gather this information.

OPM/Web supports modeling of dynamic code and construct transmission by applying two principles:

- The thing class, be it an object or a process, is separated from its instances. The thing classes are denoted by rectangles for object classes and by ellipses for process classes. Following the UML notation of classes and objects, a thing instance is denoted in OPM/Web by a rectangle/ellipse within which the thing class name is

written as **:ThingClassName**, where the identifier of the instance can optionally precede the colon. The thing class is connected to its instances via an instantiation link (denoted by a black circle within a blank triangle). It serves as a template from which the individual instances are generated. Using this principle, the designer can model separately the system classes, which are usually located on the server side, and their instances, which are generated on the server side and transferred to the client side.

- Objects and processes are defined as dual things that can be used by processes in order to perform other activities. The developer is able to use either object instances or process instances as inputs and outputs for performing an operation.

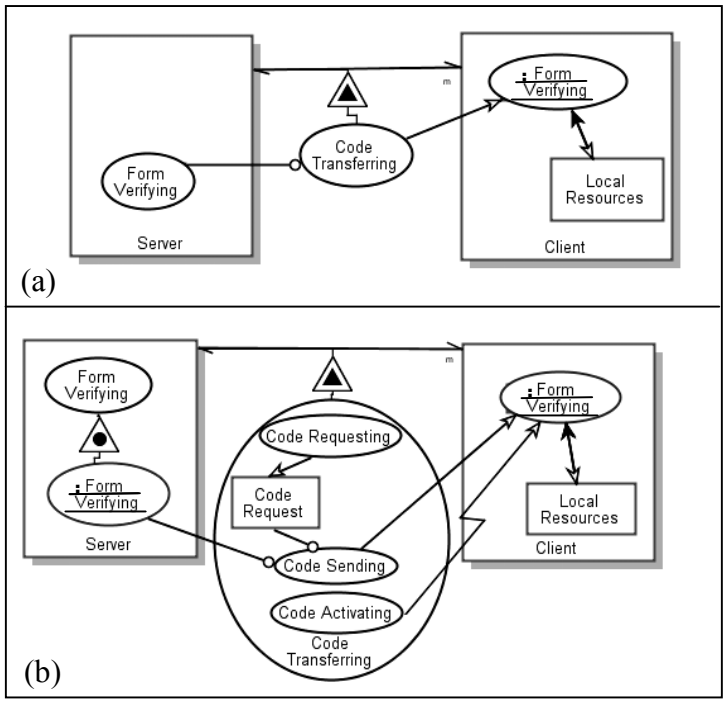


Figure 4. Modeling the **Form Verifying** transferring process. (a) The top-level version. (b) The detailed version – **Form Verifying** is unfolded and **Code Transferring** is in-zoomed.

As an example for the above extensions, Figure 4(a) models **Code Transferring** from the **Server** to the **Client**. The **Form Verifying** process class, which can be implemented as a Java applet, should verify the information within a form already filled by the user. This process class resides in the server; hence it appears within the in-zooming frame of the physical

object **Server**. The client side instance of this process is created through the **Code Transferring** process, which is an operation that characterizes the relation between the **Server** and the **Clients**.

The **Code Transferring** process is divided in Figure 4(b) into three sub-processes:

1. **Code Requesting** – When the **Code Transferring** process is activated, for example after the user pushes the "OK" or "Send" button in the relevant form, the **Code Requesting** part starts executing. This subprocess creates a **Code Request** message, asking for an instance of the **Form Verifying** process.
2. **Code Sending** – This subprocess uses an instance of the **Form Verifying** process class and sends it to the **Client**.
3. **Code Activating** – The **Code Transferring** process finishes by activating the client side instance of **Form Verifying**, which affects the **Local** (client) **Resources**.

5.4 System Integrity Constraints

Integrity constraints of Web applications are distributed and global in nature, and they involve constraints that are related to such issues as security management and dynamic updating. Web application constraints can be divided into three categories: data integrity constraints over the objects, concurrency and distribution control constraints over the processes, and global status integrity constraints over the system states. The following subsections explain the nature of these constraint types and how OPM/Web supports their validation.

5.4.1 Data Integrity Constraints

Data integrity refers to the completeness of the system information, which is modeled in the object-oriented paradigm as object classes. The integrity is achieved by defining dependency relations between different pieces of information (i.e., objects). For example, the Object

Constraint Language (OCL) [99], which was designed to accompany UML models, is an expression language that enables describing constraints on object-oriented models and other object modeling artifacts.

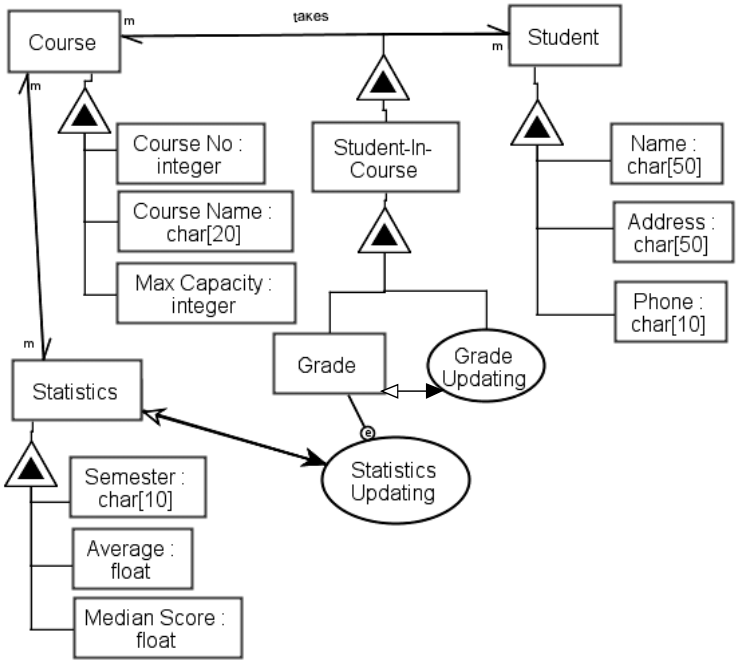


Figure 5. An OPD of the data integrity example

In OPM/Web, a dependency relation among objects is expressed in a procedural way by defining stand-alone processes. These processes automatically enforce certain constraints on the involved objects. For example, Figure 5 shows three top-level objects: **Student**, **Course**, and **Statistics**. The object **Student** stores the attributes **Name**, **Address**, and **Phone** of an actual student. **Course** stores the catalog information of the course (**Course Number**, **Course Name**, and **Maximum Capacity**). The relation between **Student** and **Course** is characterized by an object called **Student-In-Course**, which stores the student **Grade** in a specific course. The **Grade** is changed by the **Grade Updating** process, which is a service of **Student-In-Course**. The third top level object, **Statistics**, maintains statistical information about a course given in a specific semester (**Semester**, **Average**, and **Median Score**). One of the system responsibilities is that whenever a change (adding, deleting, or editing) in a student's grade occurs, the statistics

should be recalculated. In other words, the **Statistics Updating** process is activated whenever a change in a grade takes place. This is expressed by the event link between the **Grade** object and the **Statistics Updating** process. This process changes the attribute values of the **Statistics** object and thus expresses a dependency relation between those objects.

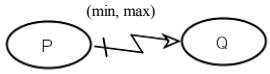
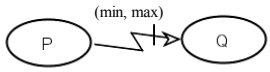
5.4.2 Concurrency and Distribution Control Constraints

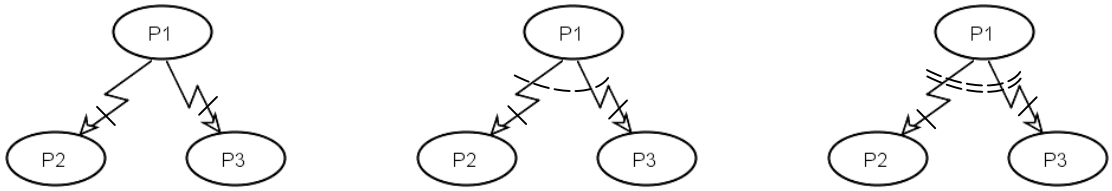
In OPM and OPM/Web the time axis is directed from the top of the diagram to its bottom within the frame of an in-zoomed process. This way, concurrent processes are located at the same height, while sequential processes are located one below the other. Constraints on the duration of an individual process can be specified in parentheses determining the lowest and uppest bounds of the process duration time.

In distributed systems, a problem arises when dependent processes occur in various physical locations. For example, a process executed at the server site may be required to terminate three seconds before another process begins at the client site. This time period is required for the central database to become stable. In order to support such concurrency control constraints, OPM/Web is augmented with timed invocation links between processes. An invocation link is denoted by a lightning arrow, as shown and explained in Table 2. Regular (round) parentheses or square brackets are used to denote open or closed time intervals. For example, (2s, 5s] means "a time interval greater than 2 seconds and smaller than or equal to 5 seconds". The default time constraints are [0, ∞).

This extension also provides the ability to express non-deterministic execution, using OPM xor and or relations between links. Figure 6 demonstrates the three possibilities of deterministic and non-deterministic executions.

Table 2. Dependency relations between distributed and concurrent processes

Symbol	Semantics
	Q must begin within a time period, which is between min and max time units after the **beginning** of P
	Q must begin within a time period, which is between min and max time units after the **end** of P



(a) (b) (c)

Figure 6. Deterministic and non-deterministic executions of processes P1, P2 and P3. (a) P2 and P3 are independently executed after the termination of P1. (b) P2 or P3 (but not both) are independently executed after the termination of P1. (b) P2 or P3 (or both) are independently executed after the termination of P1.

5.4.3 System Status Integrity Constraints

Constraints on the general status of the system involve relations among the states of the model. For example, a **Status** object, which is an attribute of **Client**, is in the state "**ok**" if and only if the **Database** object, which is on the **Server** side, is in its "**stable**" state. This implies that the system should not be concurrently at states "**ok**" and "**unstable**", and whenever this occurs, the system will take steps to correct the situation. OPM/Web handles this kind of constraints through error handling processes, as shown in Figure 7. **Wrong Status Error Handling** is triggered by **Status** being at the **wrong** state and changes **Database** from **stable** to **unstable**. An analogous chain of events occurs with **Unstable Database Error Handling**. When **Status** is wrong and

Database is **unstable**, another process, **Error Correcting** (not shown), is activated and upon its successful completion it restores **Status** to its **ok** state and **Database** to its **stable** state.

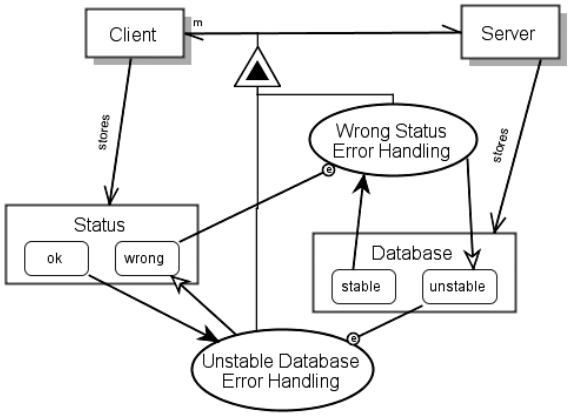


Figure 7. A status integrity constraint example

5.5 OPM/Web Vs. UML: The GLAP System Case Study

To demonstrate OPM/Web extensions, a Web-based glossary application, called GLAP, is used. The GLAP system [18] provides an online version of a software development project's glossary of terms. The project's team members can access the database of terms, using a common Web browser. Team members may also update, add entries to the database, and remove entries from it, using the same browser interface.

5.5.1 OPM/Web Model of the GLAP System

Figure 8 is the top level OPD of the GLAP system that shows the **Server** and the **Clients** as two physical object classes. The **Server** stores the **Server Components**, which consist of the **IGlossary** object, and the **Server Web Pages**. It also stores **Form Verifying** and **GLAP Server Executing** process classes. The last mentioned process is activated by **Client Requests** (as the event link from **Client Request** to **GLAP Server Executing** indicates), uses **Server Web Pages** as inputs, updates the **IGlossary**, and creates **Client Pages**. The **Client** stores **Browser Processing** and activates it according to the **User** commands (as the agent link from **User** to **Browser Processing** indicates)

and the **Client Pages**. The **Server** and the **Clients** communicate through the **Encrypted Transferring** process. This process transfers **Client Requests** from the **Client** to the **Server** (path **a**), **Client Pages** from the **Server** to the **Clients** (path **b**), and instances of **Form Verifying** from the **Server** to the **Clients** (path **c**). The **Encrypted Transferring** process can further be divided into sub-processes in order to specify its exact algorithm. For example, **Encrypted Transferring** is responsible (among other things) to transfer the instance of **Form Verifying** from the **Server** to the **Clients** as a response to a **Client Request** for a search or an edit form. In addition it activates the client instance which uses the relevant **Client Page** and creates a **Client Request** for page searching or edit confirming, respectively.

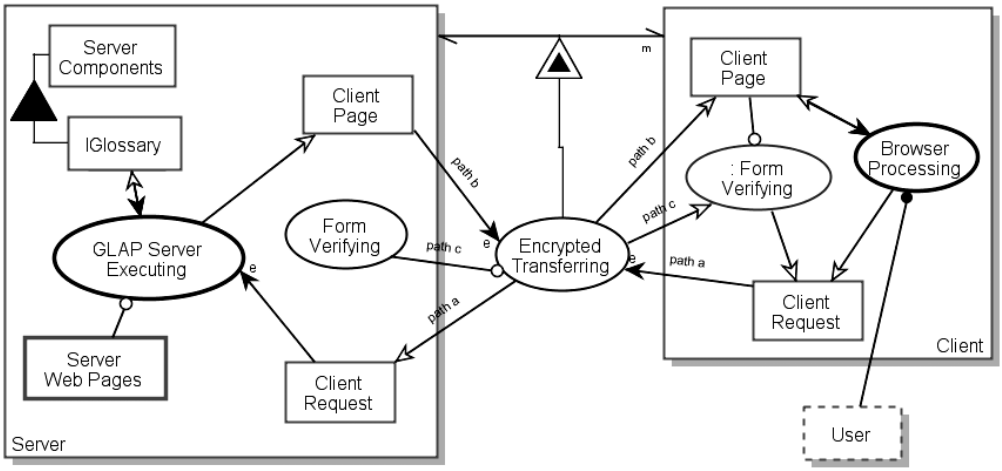


Figure 8. The top level OPD of the GLAP system

The OPD in Figure 9(a) elaborates the top level OPD by unfolding the **IGlossary** object to its constituents, **IGlossary Entry**. Each **IGlossary Entry** is characterized by one attribute (**MDSN**) and seven services (**Get Entries Starting With**, **Search Entries**, **New Entry**, **Update Entry**, **Remove Entry**, **DSN**, and **Message Text**).

The **Server Web Pages** object is unfolded in Figure 9(b) to its constituents **Process Search**, **Process Form**, **Get Entries**, and **SP Edit Entry**, each of which has its own set of attributes and services.

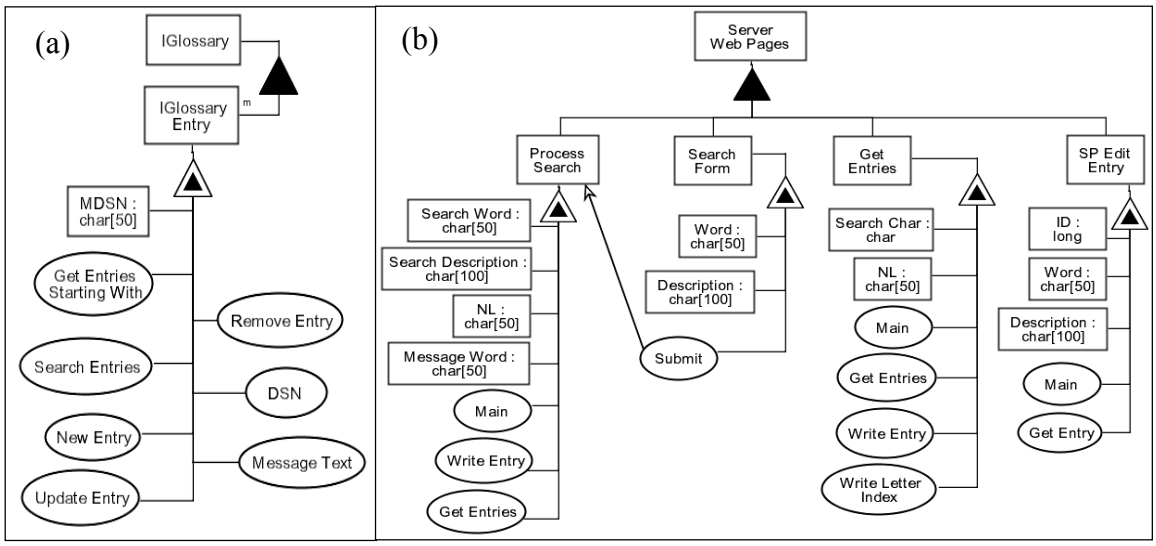


Figure 9. (a) **IGlossary**, unfolded. (b) **Server Web Pages**, unfolded.

In the OPM/Web model, separate processes describe the functionality of the client side (**Browser Processing**) and of the server side (**GLAP Server Executing**). This separation enables the developer to model the client-server communications, construct passing, and code transferring.

The server stores the stand-alone process, **GLAP Server Executing**, which is in-zoomed in Figure 10 into three sub processes: **Glossary Reading**, **Glossary Searching**, and **Glossary Entry Editing**. All three sub-processes use **Server Web Pages** but do not change them, as expressed by the instrument link between the **Server Web Pages** and the **GLAP Server Executing** process. The **Glossary Reading** and **Glossary Searching** processes use **IGlossary** as an instrument, while the **Glossary Entry Editing** process affects **IGlossary Entry** (of **IGlossary**). **Glossary Reading** is executed in response to a **read page Client Request**. **Glossary Searching** is activated in response to either a **search form Client Request** or a **search page Client Request**, and it creates accordingly either **search form Client Page** or **search page**. Similarly, **Glossary Entry Editing** is activated in response to either an **edit form Client Request** or an **edit confirmation Client Request**, and it creates accordingly either an **edit form Client Page** or an **edit confirmation Client Page**.

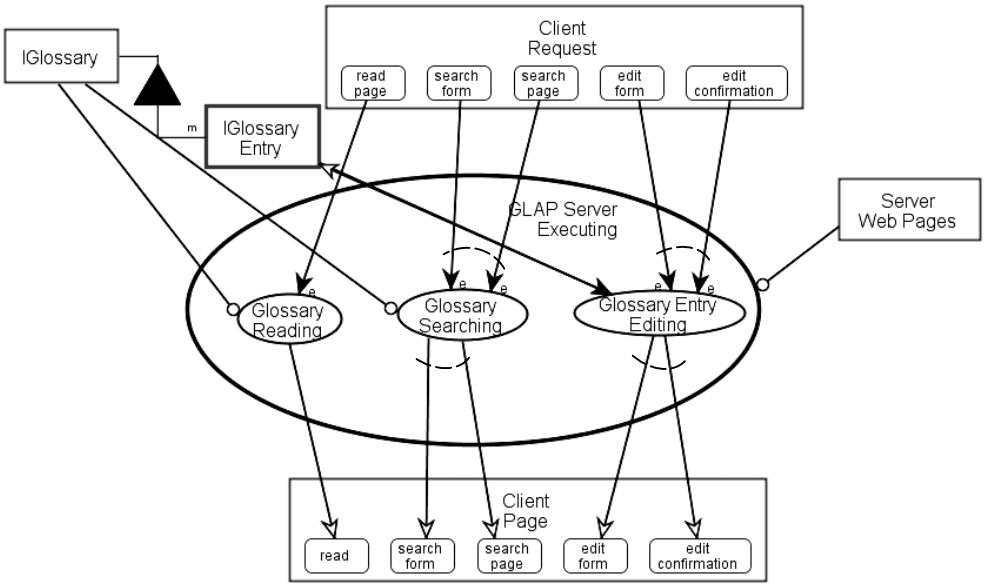


Figure 10. **GLAP Server Executing**, in-zoomed

The client side **Browser Processing**, shown in Figure 11, expresses the navigational functionality of the client computer and is executed in response to the command the **User** issues. The **Reader** can activate **Client Read Interface Managing** or **Client Search Interface Managing** to create **read page Client Requests** or **search form Client Requests**, respectively. The **Editor** can, in addition, activate **Client Edit Interface Managing** that creates **edit form Client Requests**.

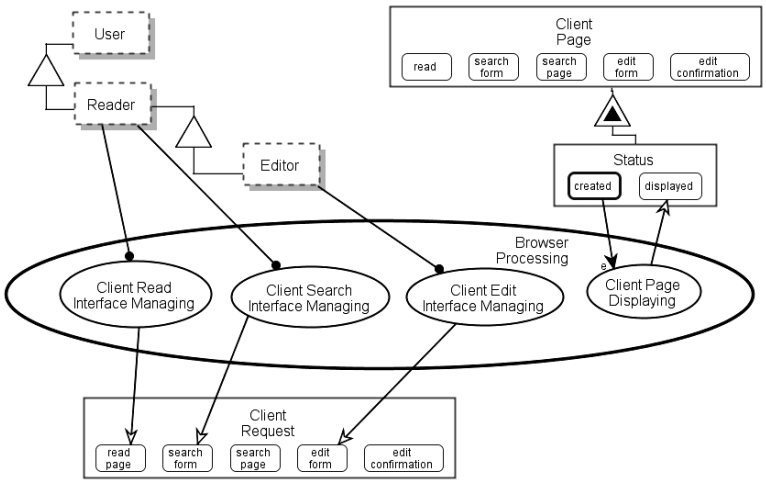


Figure 11. **Browser Processing**, in-zoomed

In the OPM/Web model, the client pages are not part of the server side and they are not statically linked with structural pointers to the creating pages of the server. Instead, they are

formed by the server processes, transferred to the client side, and displayed there by the **Client Page Displaying** process.

5.5.2 A Comparison to the UML Model of the GLAP System

Figure 12 and Figure 13 show part of an UML model of the same GLAP system, presented in [18]. The model includes a use-case diagram, a site map diagram, a package diagram, and the 'browsing detail' class diagram as a sample class diagram. The complete model includes, in addition to the ones shown here, four class diagrams and five sequence diagrams, which specify several system scenarios.

The use case diagram shows the main actors of the system: readers, who read the data, and editors, who change the data. The main use cases are Read Glossary, Search Glossary, and Edit Glossary Entry.

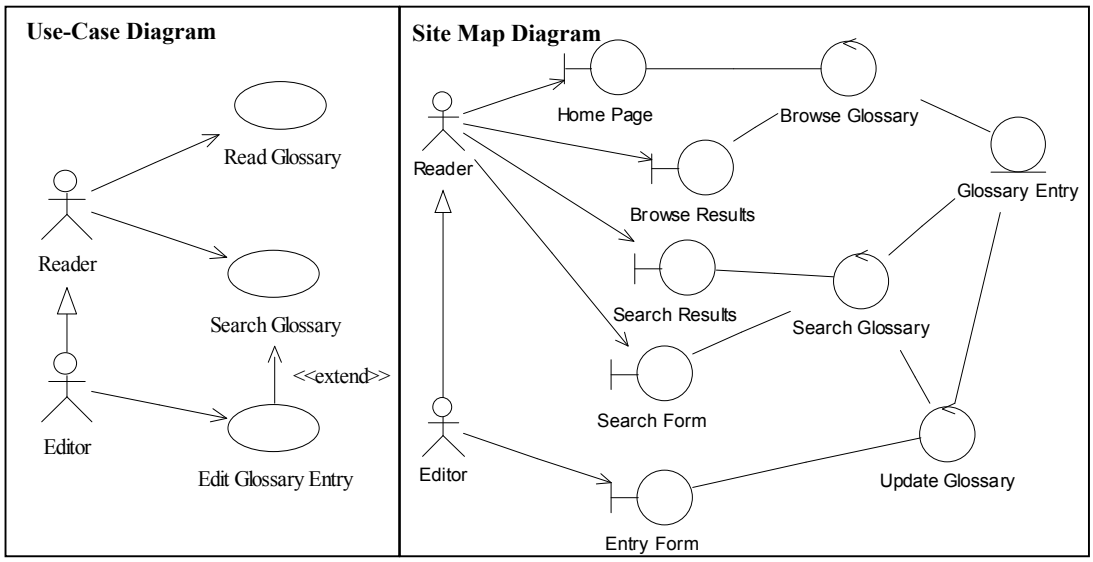


Figure 12. The use case and site map diagrams of the GLAP system

The site map diagram is a class diagram which shows an abstraction of the Web pages and navigation routes throughout the system. The GLAP system site map specifies that the Reader can view directly the home page, the browse results page, the search results page, and

the search form. The Editor can, in addition, view the entry form. The browse results page, for example, activates the browse glossary which accesses the glossary entry.

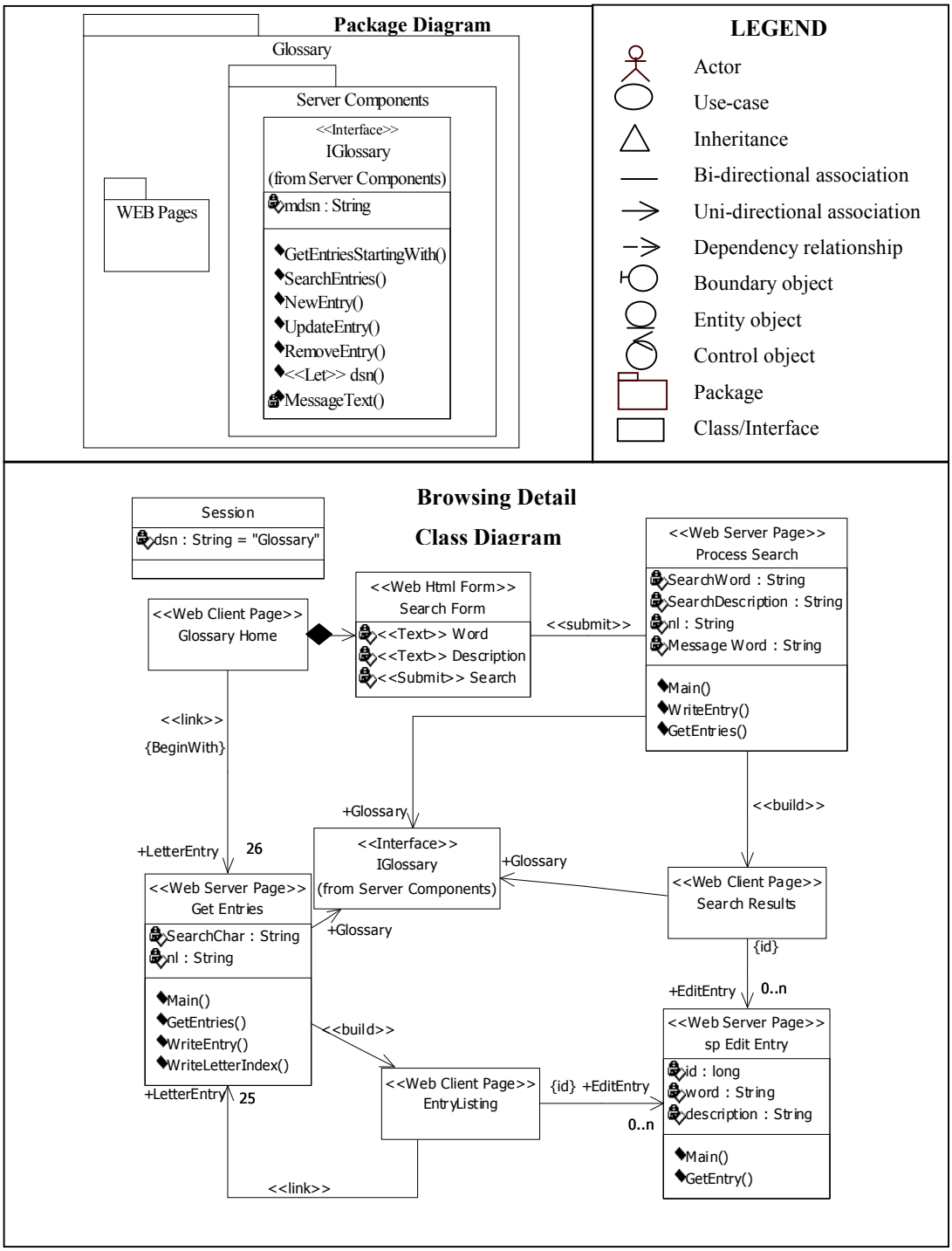


Figure 13. The package diagram and the 'Browsing Detail' class diagram of the GLAP System

The package diagram divides the system into two sub-packages: Server Components (which include the IGlossary interface object) and Web Pages. The Browsing Detail class diagram

includes the Web server pages and the Web client pages needed for modeling the glossary browsing functionality. It also specifies the structural relations among these classes, their inner structure and methods, and their constraints.

This simple example demonstrates some of the OPM/Web benefits over UML. A developer who uses UML to model the system has to remember many different symbols and to associate each symbol with the correct type of diagram. For example, to specify a string attribute in a Web HTML form, the developer has to use the stereotype <<text>> (e.g., the "Word" attribute of the "Search Form" class in the Browsing Detail class diagram). However, in other locations in the same class diagram, the reserved type "String" is used (e.g., the "Search Word" attribute of the "Process Search" class). Another example is using the "Search Results" class. In the site map diagram "Search Results" is used as a boundary object, which represents the interface between the actor and the system, while in the class diagram it refers to a Web client page. In contrast with UML, OPM/Web uses a single visual framework that makes it easier to understand the system as whole, enables the developer to grasp the system structure, behavior and functionality at the same time, and minimizes the likelihood of making design and integration errors.

The built-in scaling mechanisms of OPM/Web support automatic integration and compatibility checking, without losing the whole picture of a system. UML, on the other hand, requires the application of different diagram types for specifying distinct views of the same system, and the relationships among these views are not explicitly modeled or constrained.

While UML does not have a single clean mechanism for expressing processes, OPM/Web enables specifying the system's behavior by services or stand-alone processes, which can be described to any desired level of detail. Hence, in UML the designer should use a combination of sequence diagrams, collaboration diagrams and activity diagrams, which

describe possible scenarios, but do not capture the entire process framework in the context of the use cases or the class services. Thus, in the UML version of the GLAP example, the relation between a Web server page and its corresponding Web client page (in the 'browsing detail' class diagram) is structural. This means that there is a pointer from the server page to the client page, although the actual implementation is most likely dynamic. In the actual implementation, the server page builds the client page through a process, as the OPM/Web version explicitly expresses. Moreover, UML does not support modeling the code transferring processes, such as in Java applets, because it does not provide for specifying stand-alone processes.

6. Modeling Code Mobility and Migration Specification in OPM/Web[2]

The tradeoff between the functional requirements from Web applications and existing bandwidth limitations requires addressing issues related to code mobility and code migration. **Code mobility** is the capability of software systems to dynamically reconfigure the binding between the software components of an application and their physical locations (nodes) within a computer network [36]. **Mobile Code** is a piece of code that exhibits the mobility property, i.e., code that can be transmitted across a network and executed on another node. **Code migration** is the function which controls how code mobility is achieved [21]. Although most applications do not require mobile code, adding this capability to applications supports disconnected operations and can enhance system flexibility, reduce bandwidth consumption and total completion time, and improve fault tolerance [36].

The code migration process involves determining the operation targets, transferring the code, and integrating it into the target system. In static system architectures, the targets can be determined at compilation time. If the system architecture is dynamic, the operation targets should be computed immediately prior to transferring the code. Following the target determination, the code can be transferred by applying one of the design paradigms for code mobility, which extend the traditional client-server paradigm from data to code. Once transferred, the code can be integrated with the local target system by activating an instance of it, connecting it to existing data or code, or continuing its transfer over the network to yet another target. Modeling the code migration process also includes defining the process

[2] An extended abstract of this section was published in the Israeli Workshop on Programming Languages & Development Environments (PLE), IBM [83]. An extended version of it is submitted to the International Journal on Web Engineering Technology (IJWET).

triggers, its preconditions and postconditions, and handling security issues and possible transfer errors.

Current techniques for modeling code mobility and migration support determining the operation targets separately from the transferring stage (e.g., by class services) and do not specify how the code is to migrate. The main reason for the insufficient expressive power of these methods is the fact that they are structure-oriented in nature. Hence, although most of the systems, such as mobile applications, usually involve structure and behavior in complex, intertwined ways, current methods model these systems in an unbalanced way, emphasizing their objects and structural relations. OPM/Web enables completely specifying mobile applications within a single view by considering objects and processes as two equally important classes of entities.

6.1 Modeling Code Mobility: Design Paradigms and Modeling Techniques

Applications that involve code mobility are defined in terms of components, interactions, and sites [12]. *Components* are the building blocks of system architecture. They are further divided into *resource components*, which are objects (architectural elements representing data, or physical devices), and *computational components*, which are programs that embody flows of control. A resource component is represented in object-oriented terms as an object with attributes and operations (services) that contain knowledge about how to execute a particular task, while a computational component, which contains code, may also be characterized by private data, an execution state, and bindings to other (resource or computational) components. *Interactions* are events that involve two or more components communicating with each other. *Sites* are nodes or execution environments – they host components and provide support for the execution of computational components.

6.1.1 The Client-Server Paradigm and Related Approaches

The **Client-Server (CS)** paradigm [85] is the traditional design approach for distributed communication among sites, in which messages are transferred from one site to another, but actual code is not. In a typical client-server interaction, site S_B, which acts as the interaction server, offers a set of services. It also hosts the resources and the knowledge needed for executing these services. Site S_A, which is the operation client, requests the execution of some service offered by S_B by sending it a message. As a response, S_B performs the requested service and delivers the result back to S_A in a subsequent interaction. If the server does not have all the data and knowledge required, it can act as a client in another client-server interaction.

The CS paradigm has been criticized as being too low-level, requiring developers to determine network addresses and synchronization points. CS interaction is also too specific, since the client must "know" the exact services that the server can provide [21]. The Remote Procedure Call (RPC) [8] tries to overcome these shortcomings by enabling the client to request a service to be executed on a server in the same way that it would make a local function call; the location of the server, the initiation of the service, and the transportation of the results are handled transparently to the client. The object-oriented approach attempts to make the CS paradigm more accessible and uniform by adopting reuse, inheritance, and encapsulation principles. OMG's Common Object Request Broker Architecture (CORBA) [69], for example, is a CS technology that is based on the object-oriented approach.

6.1.2 Design Paradigms for Code Mobility

Design paradigms for code mobility extend the CS paradigm by transporting computational components across a network. Four common design paradigms for code mobility are Remote Evaluation (REV), Code-on-Demand (COD), PUSH, and Mobile Agents (MA). These paradigms differ in their preconditions, postconditions, and triggers.

In the **Remote Evaluation** (REV) paradigm [92], a computational component, C, located at S_A, has the knowledge (represented by code) necessary to perform a service, but it lacks the required resource components, which are located at a remote site S_B. Therefore, C is transferred from S_A to S_B and is executed there. The results of this execution are delivered back to S_A in an additional interaction.

In the **Code-on-Demand** (COD) paradigm [12], site S_A can access the resource components needed for a service, but it does not have the knowledge required to process them. Therefore, S_A requests the service execution knowledge, i.e., the computation component C, from its hosting site, S_B. S_B delivers the knowledge to S_A, which subsequently processes C at site S_A on the resource components residing there. Contrary to REV, in COD the code is executed at the client.

In the **PUSH** paradigm [34], site S_B sends a (computational or resource) component to site S_A in advance of any specific request. This push-based operation is often preceded by a profiling operation, in which S_A specifies a profile that reflects its users' interests. The profile is sent to site S_B, saved there, and used by S_B to decide which components S_A should receive and when to send them. The advantage of this paradigm over COD is that the users do not have to know when to pull new components and where to pull them from. Rather, the system automatically sends necessary new components when they become available, and they are often used later by the receiving node.

In the **Mobile Agent** (MA) paradigm [41], site S_B owns the service execution knowledge, C, but some of the required resource components are located at site S_A. Hence, C migrates to S_A and completes the service using the resource components available there. The migration is usually initiated by the agent (C), but it might be requested by S_A or S_B. Contrary to the REV, COD, and PUSH paradigms, which focus on the transfer of just code between sites, the

mobile agent migrates to the remote site as a whole computational component, along with its state, the code it needs, and some of the resource components required to perform the task.

Discussing these design paradigms for code mobility, Carzaniga et al. [12] claim that none of them is absolutely better than the others and suggest choosing the most appropriate paradigm for a system under development on a case-by-case basis according to the application type.

6.1.3 Modeling Techniques for Specifying Code Mobility and Migration

Code mobility is supported by such programming environments as Java, Telescript, and D'Agents. However, current modeling techniques that are used in the analysis and design phases of Web applications do not address the code mobility concept, including triggers, conditions, and security issues, at a satisfactory level.

Hypermedia authoring techniques model the content and navigational aspects of an application, but not its functionality, physical architecture, or security requirements. Therefore, they do not explicitly address code-related issues, such as code migration.

Object-oriented development methods, notably UML, enable modeling of the application functionality through class services and message passing among objects. Concepts involving code mobility, such as Java applets, are modeled in separate views using pre-declared UML stereotypes. Therefore, modeling these concepts with an object-oriented method is technology-dependent (e.g., specific to the Java language and its applets). Moreover, UML does not handle the code migration process as a whole pattern, including its preconditions (e.g., the existence of a request in the client site and source code at the server site), postconditions (e.g., the existence of executable code at the client site), and triggers (e.g., a change in a server component). Trying to overcome these shortcomings, UML has been extended by various research teams, including the mobile agent extension [52], Agent UML (AUML) [74], and MASIF-DESIGN [65]. Even though the proliferation of such extensions

undermine and weaken UML standardization efforts, they still do not separate the execution knowledge (services) from the resource components (classes). It should come as no surprise that such separation is not possible, since doing so would work against the encapsulation of operations within object classes, which is a major principle in the object-oriented approach.

Behavior-oriented techniques model the system functionality separately from the application structure. They enable static binding of processes to sites, but do not support the modeling of dynamic configurations and the actual migration process.

OPM enables specifying the architecture of the system along with its structure and behavior. It also enables transfer of objects between sites. Nevertheless, code migration specification cannot be carried out in OPM.

6.2 OPM/Web and Mobile Components

As noted, OPM/Web clearly distinguishes between thing classes and instances. An *object class*(abbreviated as an object) is a set of object instances which exist, or at least have the potential of stable, unconditional physical or logical existence. A *process class* (abbreviated as a process) is a pattern of transformation of one or more object classes. A program, an operation, a procedure, and an algorithm are examples of process classes. An actual execution of a process (such as the carrying out of an executable version of a program or an algorithm) is a *process instance*. The code migration process can transfer process classes or instances.

6.2.1 Mapping Mobility Terms onto OPM/Web Concepts

The terms used in the various design paradigms for code mobility are mapped to OPM/Web concepts as follows.

- A *resource component* is an informatical or physical object. An informatical object is a piece of information, such as the data required for a process execution. A physical object is tangible in the broad sense, for example a device.

- A *computational component* is a process. It can own private data (objects) and include sub-processes. The migration process can transfer the computational component source code (i.e., a process class), which can be compiled at the target site and run there any number of times, or an executable version of the code (i.e., a process instance), which can run at the target site only a specified number of times.
- A *site*, which is analogous to a node in the UML implementation model, is a physical object in OPM/Web. This physical object can be in-zoomed to expose its resource and computational components.
- An *interaction* has both structural and dynamic aspects. The structural aspect of an interaction specifies how two sites can communicate with each other, irrespective of a specific point in time. This aspect is modeled in OPM/Web by a (unidirectional or bi-directional) structural link between the communicating sites, which, as noted, are physical objects. The dynamic aspect of an interaction is the ability to transfer data (objects) or code (processes) between two sites and is specified in OPM/Web by an event-driven process. Since interaction conceptually characterizes the communication between the sites, the interaction process is associated in the model to the structural link that connects the two interacting sites. The implementation of this interaction may still be carried out as two inter-related processes, one at each interacting site.

The basic code transferring operation is represented by the generic OPD in Figure 14. The computational **Component** on the left of Figure 14(a) and Figure 14(b), which is a process class denoted by an ellipse, is the input for the **Component Transferring** process, as the instrument link between them indicates. In Figure 14(a) the **Component Transferring** process transfers **Component**'s source code, while in Figure 14(b) **Component Transferring** transfers a process instance, i.e., only an executable version of **Component**, denoted by **:Component**. The semantics of the arrow with the white (blank) arrowhead from **Component Transferring** to the

right appearance of **Component** is of a result link, which means that **Component Transferring** creates (a copy of) the process class **Component**, as in Figure 14(a), or an instance of it, as in Figure 14(b). In the original OPM, processes are not connected, and, hence, there is no difficulty to determine which is the processing entity. To remove the ambiguity arising from connecting two processes in OPM/Web via consumption or result links, a consumption link is denoted as a black-headed arrow from the consumed entity to the consuming process, while the semantics of a white-headed arrow from a process to an entity remains a result link. The identical path labels[3] on the instrument and result links and the identical component names indicate that **Component Transferring** transfers **Component** as is rather than computing it from an input.

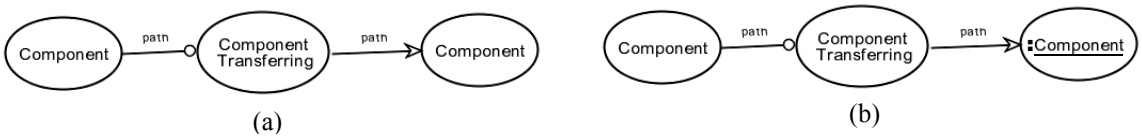


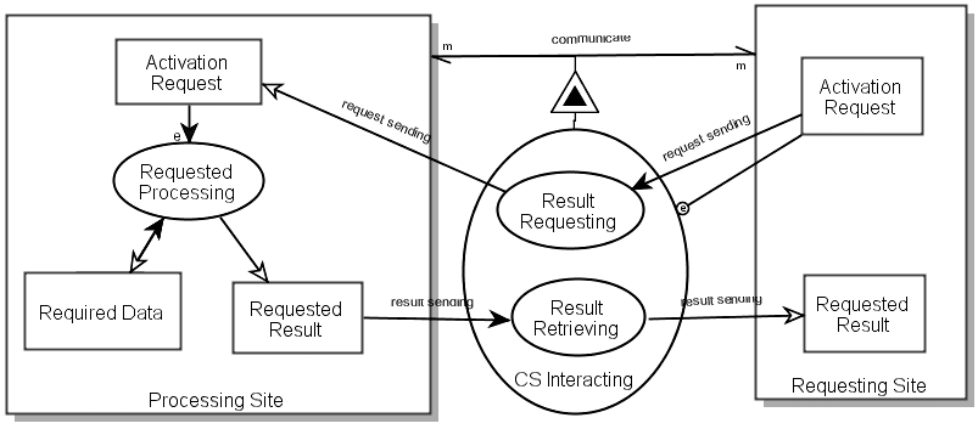
Figure 14. A generic OPM/Web model of a **Component Transferring** process.
(a) **Component Transferring** transfers **Component**'s code, leaving the original **Component** intact. (b) **Component Transferring** transfers an instance of **Component**, leaving the original **Component** intact.

6.2.2 Modeling the Client-Server Paradigm using OPM/Web

Based on the mapping of code mobility terms onto OPM/Web concepts, an OPM/Web model of the traditional client-server paradigm, presented in Figure 15, consists of two complementary modalities: graphical – the OPD in Figure 15(a), and textual – the OPL paragraph in Figure 15(b). The objective of this unique dual representation is to enhance the readability of the model by humans: graphics-oriented model readers, who are familiar with

[3] A *path label* in OPM is a label on a procedural link that removes the ambiguity arising from multiple incoming/outgoing procedural links. Here identical path labels on the incoming link to and the outgoing link

OPM and its diagrammatic notation, can relate to the OPD, while text-oriented readers, or those who are new to the OPM graphic notation, can refer to the OPL paragraph and learn the correspondence between each OPL sentence or phrase and its OPD construct counterpart.



(b) **Requesting Site,** which is physical**,** zooms into **Activation Request** and **Requested Result.**

Activation Request triggers **CS Interacting.**

Processing Site, which is physical**,** zooms into **Activation Request**, **Required Data**, and **Requested Result,** as well as **Requested Processing.**

Activation Request triggers **Requested Processing.**

Requested Processing consumes **Activation Request** of **Processing Site.**

Requested Processing affects **Required Data.**

Requested Processing yields **Requested Result** of **Processing Site.**

Many **Requesting Sites** and many **Processing Sites communicate**. This link exhibits **CS Interacting.**

CS Interacting zooms into **Result Requesting** and **Result Retrieving.**

Following path **request sending**, **Result Requesting** consumes **Activation Request** of **Requesting Site.**

Following path **request sending**, **Result Requesting** yields **Activation Request** of **Processing Site.**

Following path **result sending**, **Result Retrieving** consumes **Requested Result** of **Processing Site.**

Following path **result sending**, **Result Retrieving** yields **Requested Result** of **Requesting Site.**

Figure 15. An OPM/Web model of the Client-Server (CS) paradigm: (a) The OPD (b) The corresponding OPL paragraph

---

from the **Component Transferring** process are used to denote the transfer flow.

Examining Figure 15, one can see that **Requesting Site** (the client) and **Processing Site** (the server) are both physical objects (as denoted by shadowed rectangles). The computational component, **Requested Processing**, resides in the **Processing Site**, which also hosts the resource components required for that computation, **Required Data** and (later on) **Requested Result**. The two sites are connected via a bi-directional structural link, tagged **communicate**, which exhibits (i.e., is characterized by) the **CS Interacting** process. A change in (an instance of) **Activation Request** at the **Requesting Site** initiates the **CS Interacting** process, as the event link (the circle-headed link with the letter 'e' inside it) between the two things shows. Following the **request sending** path, the first sub-process of the **CS Interacting** process, which is **Result Requesting**, transfers a copy of **Activation Request** to the **Processing Site**. As soon as this copy is placed at the **Processing Site**, it activates the **Requested Processing**, as the consumption event link (the black-headed arrow with the letter 'e' next to it) denotes. This **Requested Processing** potentially affects the **Required Data** object and yields (produces) the **Requested Result** object. The creation of **Requested Result** enables the second stage of the interaction, executed by **Result Retrieving**. Following the **result sending** path, this process moves the local copy of the generated **Requested Result** from the **Processing Site** to the **Requesting Site**.

Table 3 summarizes the structure of **Requesting Site** and **Processing Site** before and after an activation of a **CS Interacting** process.

Table 3. The resource and computational components in **Requesting Site** (the client) and **Processing Site** (the server) before and after an activation of **CS Interacting**

Design Paradigm (**Process Name**)	***Time***	***Requesting Site***	***Processing Site***
Client Server (**CS Interacting**)	Before	**Activation Request**	**Requested Processing** (code) **Required Data**
	After	**Requested Result**	**Requested Processing** (code) **Required Data**

6.2.3 *Simulating Mobile Specifications with OPCAT*

Using the new version 2.1 of OPCAT (OPM CASE Tool), with which the OPM models in this work were generated, a system model can also be simulated by animation. In the CS paradigm, for example, the simulation starts by making the precondition set of the **CS Interacting** process true. This is done by enabling (through highlighting) all the components (objects and processes) which are not created by processes in the given model, i.e., the objects **Activation Request** of **Requesting Site** and **Required Data** and the process **Requested Processing**, as shown in Figure 16(a). While executing **CS Interacting**, the **Activation Request** at the **Processing Site** becomes highlighted, then the **Requested Result** at the **Processing Site**, and finally the **Requested Result** at the **Requesting Site**. After the transfer process has been completed, its postcondition set becomes true, i.e., **Requested Result** at the **Requesting Site, Required Data** at the **Processing Site**, and **Requested Processing** are highlighted, as shown in Figure 16(b). Using this simulation capability of OPCAT, design errors that were not detected in the static model can be spotted and corrected before starting the implementation. Appendix F elaborates on OPCAT 2 in general and its simulation capability in particular.

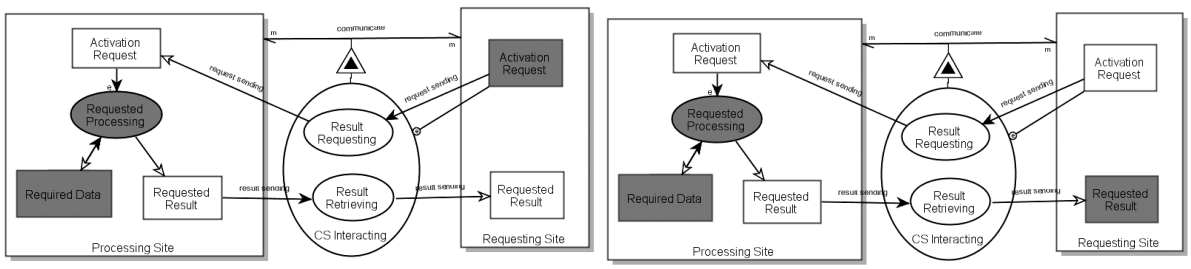


Figure 16. OPCAT 2 simulation snapshots before (a) and after (b) executing **CS Interacting**. Existing things in a snapshot appear in gray.

6.3 *OPM/Web Models of Code Mobility Design Paradigms*

OPM/Web enables precise modeling of the REV, COD, PUSH, and MA paradigms, which were explained informally in Section 6.1.2. In this section, generic OPM/Web models for these design paradigms are presented. In all of these models, **Requesting Site** is the transaction

client, and as such, it obtains a copy of the **Requested Result** and keeps it at the end of the process. **Activation Request** is the trigger for the code transferring process. The **Resource Site** is the transaction server, i.e., it hosts the **Requested Processing** (as in COD, PUSH, and MA) or the **Required Data** (as in REV). The COD, PUSH, and MA models describe transferring a one-time executable version of code (i.e., a process instance) from the **Resource Site** to the **Requesting Site**, and executing it in the remote site. The REV model specifies a process that transfers an executable version of code from **Requesting Site** to **Resource Site** and executes it there. Replacing the process instance with a process class supports transfer of source code that can later be instantiated, i.e., compiled and executed. Removing the execution part and connecting the transferred process class to other local things support mixins [30]. The various code mobility models can become generic components in specifications of mobile applications, as explained and demonstrated in Section 6.4.

Table 4 summarizes the components that reside at the **Requesting Site** and the **Resource Site** before and after the transfer of a process instance in each one of the four mobile code design paradigms. Note that the table reflects the situation before **Requested Processing** took place, so **Requested Result** does not yet exist. After this transfer, the executable code may be activated, creating **Requested Result**.

Table 4. The resource and computational components in **Requesting Site** (the "client") and **Resource Site** (the "server") before and after an activation of the transfer processes in each one of the four code mobility design paradigms.

Code Mobility Design Paradigm (**Process Name**)	***Time***	***Requesting Site***	***Resource Site***
Remote Evaluation (**REV Interacting**)	Before	**Activation Request** **Requested Processing** code	**Required Data**
	After	**Requested Processing** code	**Required Data** **Requested Processing** instance
Code-on-Demand (**COD Interacting**)	Before	**Activation Request** **Required Data**	**Requested Processing** code
	After	**Required Data** **Requested Processing** instance	**Requested Processing** code
PUSH (**PUSH Interacting**)	Before	**Required Data**	**Requested Processing** code **Profile** **Activation Request**
	After	**Required Data** **Requested Processing** instance	**Requested Processing**code **Profile**
Mobile Agent (**MA Interacting**)	Before	**Required Data**	**Requested Processing** instance **(+ Execution Status + Private Data)**
	After	**Required Data** **Requested Processing** instance **(+ Execution Status + Private Data)**	If clones: **Requested Processing** instance **(+ Execution Status + Private Data)**

6.3.1 Remote Evaluation

The OPD in Figure 17 is an OPM/Web model of the Remote Evaluation (REV) paradigm. The following OPL paragraph describes the same REV model textually.

Requesting Site, which is physical**,** zooms into **Activation Request** and **Requested Result**, as well as **Requested Processing.**

Activation Request triggers **REV Interacting.**

Resource Site, which is physical**,** zooms into **Required Data** and **Requested Result,** as well as **Requested Processing** instance**.**

Requested Processing instance affects **Required Data.**

Requested Processing instance yields **Requested Result** of **Resource Site.**

Many **Requesting Sites** and many **Resource Sites communicate**. This link exhibits **REV Interacting.**

REV Interacting consumes **Activation Request.**

REV Interacting zooms into **Code Sending, Code Activating**, and **Result Retrieving.**

Following path **code sending, Code Sending** requires **Requested Processing** of **Requesting Site.**

Following path **code sending, Code Sending** yields **Requested Processing** instance of **Resource Site.**

Code Activating invokes **Requested Processing** instance of **Resource Site.**

Following path **result sending, Result Retrieving** consumes **Requested Result** of **Resource Site.**

Following path **result sending, Result Retrieving** yields **Requested Result** of **Requesting Site.**

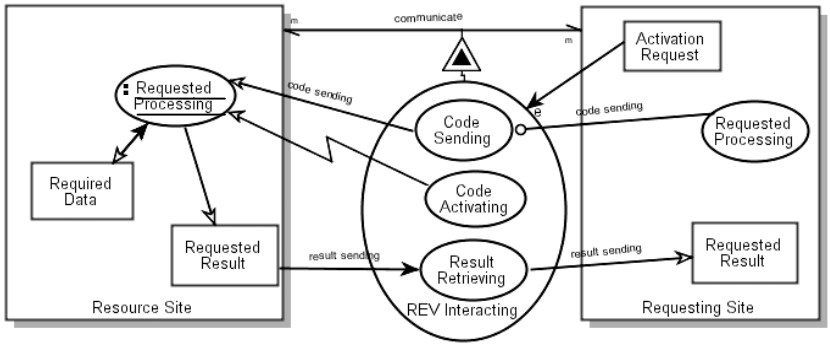


Figure 17. A generic OPD of the REV paradigm

6.3.2 Code-on-Demand

The OPD in Figure 18 is a generic model of the Code-on-Demand (COD) paradigm, while the OPL paragraph below is its textual counterpart.

Requesting Site, which is physical, zooms into **Activation Request**, **Required Data**, and **Requested Result,** as well as **Requested Processing** instance**.**

Activation Request triggers **COD Interacting.**

Requested Processing instance affects **Required Data.**

Requested Processing instance yields **Requested Result.**

Resource Site, which is physical, zooms into **Requested Processing.**

Many **Requesting Sites** and many **Resource Sites communicate.** This link exhibits **COD Interacting.**

COD Interacting consumes **Activation Request.**

COD Interacting zooms into **Code Retrieving** and **Code Activating.**

Following path **code sending, Code Retrieving** requires **Requested Processing** of **Resource Site.**

Following path **code sending, Code Retrieving** yields **Requested Processing** instance of **Requesting Site.**

Code Activating invokes **Requested Processing** instance of **Requesting Site.**

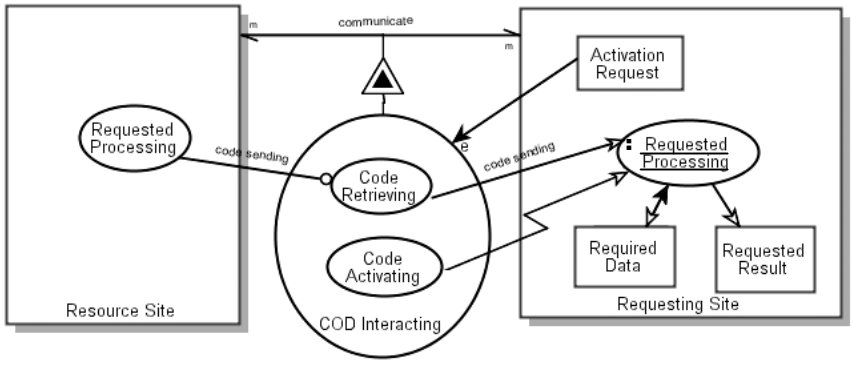


Figure 18. A generic OPD of the COD paradigm

Figure 18 clearly shows that processing (i.e., the activation of a **Requested Processing** instance) in the COD model occurs at the **Requesting Site**, whereas in the REV model, shown in Figure 17, the processing takes place in the **Resource Site**. The fact that **Requested Processing** is not initially at the **Requesting Site** is denoted in Figure 18 by the result link (the white arrowhead) whose destination is the **Requested Processing** instance at the **Requesting Site**, indicating that the **Requested Processing** instance was created there only after the first stage of **COD Interacting**, which is **Code Retrieving**, occurred. As described in Section 6.2.3, OPCAT 2

enables simulation of the behavior of this system, showing more vividly the sequence of occurrences. When the animated simulation is run, the **Requested Processing** instance appears only in the postcondition set of **Code Retrieving**.

6.3.3 PUSH

Figure 19 is a generic model of the PUSH paradigm. The following OPL sentences describe the model.

Requesting Site, which is physical**,** zooms into **Required Data** and **Requested Result,** as well as **Requested Processing** instance**.**

Requested Processing instance affects **Required Data.**

Requested Processing instance yields **Requested Result.**

Resource Site, which is physical**,** zooms into **Activation Request** and **Profile,** as well as **Requested Processing.**

Many **Activation Requests relates to** many **Profiles.**

Activation Request triggers **PUSH Interacting.**

Many **Requesting Sites** and many **Resource Sites communicate.** This link exhibits **PUSH Interacting.**

PUSH Interacting occurs if **Profile** of **Resource Site** is **requesting site.**

PUSH Interacting consumes **Activation Request.**

PUSH Interacting zooms into **Code Retrieving** and **Code Activating.**

Following path **code sending, Code Retrieving** requires **Requested Processing** of **Resource Site.**

Following path **code sending, Code Retrieving** yields **Requested Processing** instance of **Requesting Site.**

Code Activating invokes **Requested Processing** instance of **Requesting Site.**

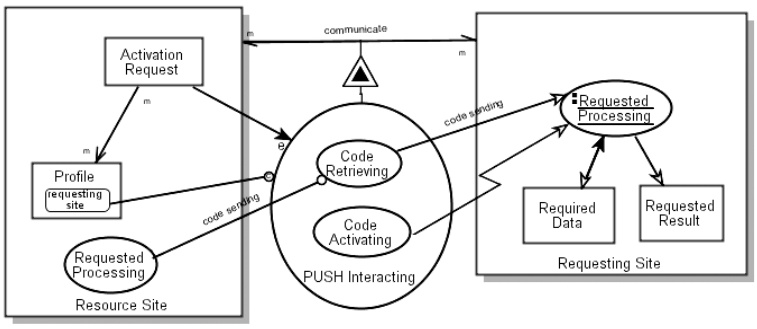


Figure 19. A generic OPD of the PUSH paradigm

The condition link from **requesting site Profile** to **PUSH Interacting** specifies that when triggered (by **Activation Request**), **Requested Processing** is transferred only to sites that were registered in the **Profile**.

6.3.4 Mobile Agents

Various definitions of an agent [33] agree that all software agents are computer programs, but not all programs are agents. Each agent definition indicates some properties that differentiate an agent from a "conventional" program. An agent is expected to be reactive, autonomous, goal-oriented, temporally continuous, communicative, learning, mobile, and flexible. Agents of the same class or of different classes can communicate with each other using objects. These definitions of an agent as a computer program with additional characteristics call for modeling an OPM/Web agent as a process instance, which belongs to a process class. These process instances (agents) initiate their own migration at specific points of their execution. Figure 20 and the corresponding OPL paragraph describe a mobile agent model for the case in which the agent is cloned from the **Resource Site** to the **Requesting Site**.

Requesting Site, which is physical, zooms into **Required Data** and **Requested Result**, as well as **Requested Processing** instance**.**

Requested Processing instance exhibits **Execution Status** and **Private Data**.

Execution Status can be **transfer** or **local**.

Requested Processing instance affects **Required Data**.

Requested Processing instance yields **Requested Result**.

Resource Site, which is physical, zooms into **Requested Processing** instance.

Requested Processing instance exhibits **Execution Status** and **Private Data**.

Execution Status can be **transfer** or **local**.

Execution Status triggers **MA Interacting** when it enters **transfer**.

Many **Requesting Sites** and many **Resource Sites communicate.** This link exhibits **MA Interacting**.

MA Interacting zooms into **Agent Migrating** and **Agent Activating**.

Following path **code sending**, **Agent Migrating** requires **Requested Processing** instance of **Resource Site.**

Following path **code sending**, **Agent Migrating** yields **Requested Processing** instance of **Requesting Site**.

Agent Activating changes **Execution Status** of **Requested Processing** instance of **Requesting Site** from **transfer** to **local**.

Agent Activating invokes **Requested Processing** instance of **Requesting Site**.

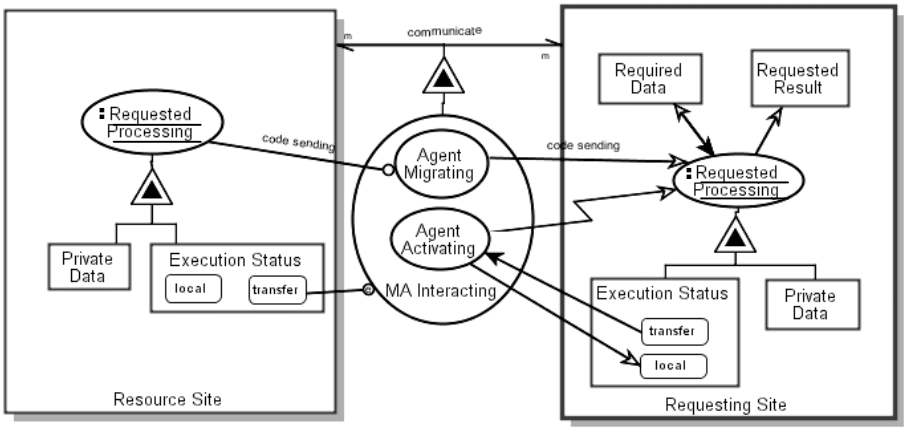


Figure 20. A generic OPD of the MA paradigm

The instrument link from the agent (the **Requested Processing** instance at the **Resource Site**) to **Agent Migrating** (within **MA Interacting**) denotes that this migration clones (i.e., makes a copy of) the **Resource Site**'s agent at the **Requesting Site**. Alternatively, **MA Interacting** might move the agent, in which case a consumption link from **Requested Processing** instance of **Resource Site** to **Agent Migrating** replaces the instrument link, implying that the agent at the **Resource Site** disappears.

6.4 Reusing OPM/Web Code Mobility Models: The QoS System Example

As noted, transferring a (resource or computational) component between sites involves determining the source and target sites, integrating the transferred component within the target sites, addressing network security issues, and handling errors that may occur in the process. These aspects can be incorporated in the single view of OPM/Web models by reusing one or more of the code migration design paradigms, presented in the previous section. To demonstrate the value of OPM/Web approach, a partial model of a Quality of Service (QoS) system is presented [52]. This system is a mobile application in which software components from multiple parties collaborate to provide a particular service to end users.

As the top level diagram in Figure 21 shows, the QoS system consists of three types of sites: **Client**, **ISP** (Internet Service Provider) **Agency**, and **Router Agency**, each of which may have multiple instances. Each site type is modeled as a physical object that inherits from **Site**, which represents a network node. At this level of abstraction, the **Client** is shown to include only the **QoS Interface Handling** process, with which the **Service User** interacts. The **Service User** is an actor using the system and is therefore modeled as an external (dashed) and physical (shadowed) object. Not knowing which routers provide the requested service, the **Service User** interacts via the **QoS Interface Handling** process, which the **Client** site hosts. This interaction is indicated in Figure 21 by the agent link from **Service User** to **QoS Interface Handling**. Each **Client** is connected to **ISP Agencies**, and each **ISP Agency** is connected to several sites of type **Router Agency**.

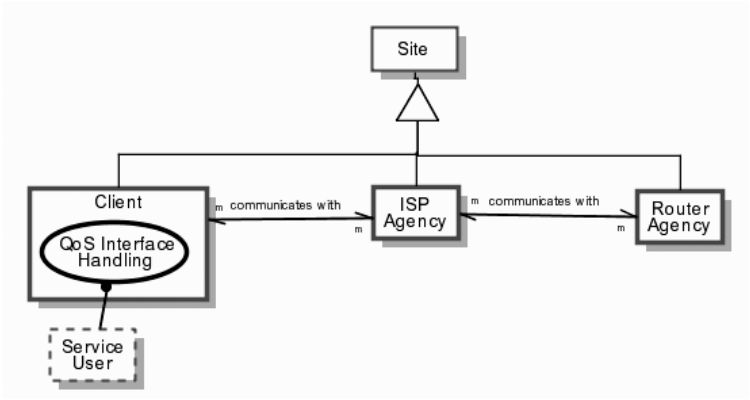


Figure 21. The top level OPM/Web diagram of the **QoS System**

In UML, this system would need three different types of UML diagrams: deployment diagrams to describe the system physical architecture, use case diagrams to describe the user-system interactions, and sequence diagrams to describe scenarios of the communication processes. However, even these three diagram types combined do not describe the details of the interaction processes, as do the next two OPDs in Figure 22 and Figure 23.

Refining the interaction between the **Client** and the **ISP Agency**, Figure 22 shows that their communication structural relation exhibits two operations: **CS Interacting** and **COD Interacting**.

The details of the models of the Client-Server (CS) and Code-on-Demand (COD) paradigms have been presented earlier. **COD Interacting**, for example, is the same as the process modeled in Figure 18, where **ISP Agency** is the server (**Resource Site**), **Parameter Check Request** is **Activation Request**, and **Parameter Checking** is **Requested Processing**. Therefore, **CS Interacting** and **COD Interacting** are not in-zoomed further here.

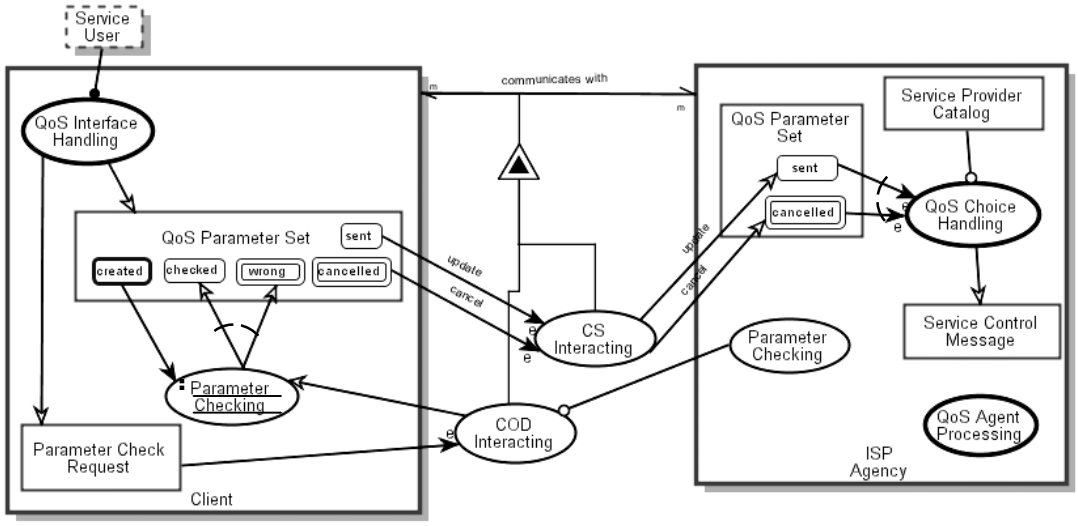


Figure 22. Detailing the **Client – ISP Agency** interaction

When weaving these models into a complete application, the combined model can be enhanced to handle security issues and possible transfer errors. Since the security and privacy algorithms are, most likely, pre-defined computational components, they can be modeled as OPM/Web processes, from which the transfer processes can inherit the functionality and interface. The different kinds of transfer errors, such as communication failures, unknown addresses, and timeout exceptions, can be traced using OPM event links. These links model a variety of events, including process timeout, process termination, state change, state entrance, state timeout, and external events [77].

In addition to showing the details of the interaction between the **Client** and the **ISP Agency** components, Figure 22 also zooms into the **Client** and the **ISP Agency** components, exposing a more refined view of their internal objects and processes. **QoS Interface Handling**, which is the

computational component of the **Client**, handles requests that the **Service User** submits. When activated by the **Service User**, **QoS Interface Handling** creates the objects **QoS Parameter Set** and **Parameter Check Request**. Upon its creation, **Parameter Check Request** activates **COD Interacting**, which transfers an instance (one-time executable version) of **Parameter Checking** from the **ISP Agency** to the **Client**. This **Parameter Checking** execution (at the client site) changes the state of **QoS Parameter Set** from **created** to either **checked** or **wrong** (as denoted by the dashed arc). Through the **QoS Interface Handling** process, the **Service User** can continue affecting **QoS Parameter Set**, in order to request services (via the **update** path) or to cancel them (via the **cancel** path). These requests are transferred to **ISP Agency** by the **CS Interacting** process, which does not need to wait for a response from the **ISP Agency**.

Unlike UML and its extension mechanisms, OPM/Web specifies the communication processes generically, regardless of their implementation technology. For example, the **COD Interacting** process specifies a common design paradigm for code mobility without limiting it to specific implementation language constructs (such as Java applets or C# in the .NET environment). As this example shows, OPM/Web also supports modeling the events which trigger the communication processes, as well as the conditions that enable their activations.

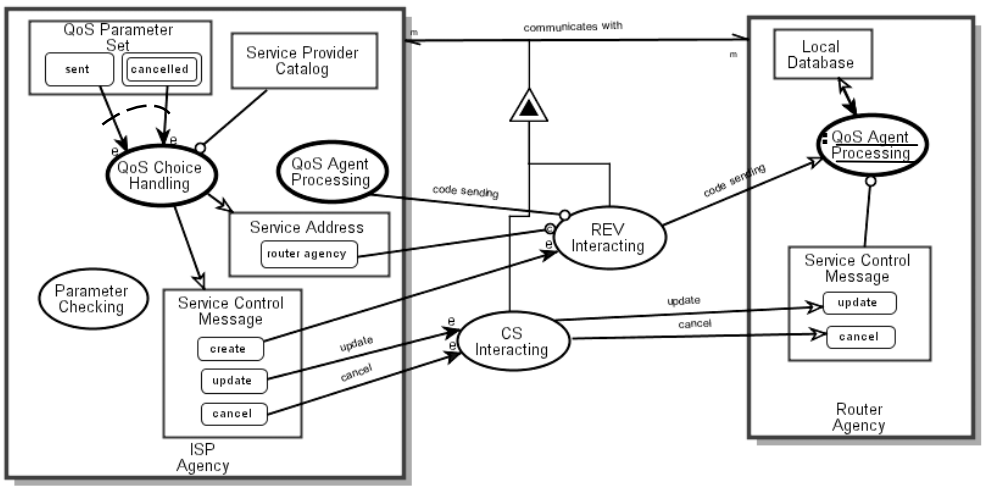


Figure 23. Detailing the **ISP Agency – Router Agency** interaction

Figure 23 is a refinement of the interaction between **ISP Agency** and **Router Agency**. Since not all the **Router Agencies** provide all the services, the **QoS Choice Handling** uses a **Service Provider Catalog** as an instrument for creating a **Service Control Message** and the **Service Address** object, which defines a **router agency** address for the required service. If the **Service Control Message** requests a new service (which is the case when its state is **create**), then the **REV Interacting** process is activated, transferring an executable version of **QoS Agent Processing** to the **Router Agency** according to the **Service Address**. If the **Service Control Message** is created in its **update** or **cancel** states, it is transferred as is to the **Router Agency** by the **CS Interacting** process, enabling the continuous running of **QoS Agent Processing** in the **Router Agency**, where it can use the **Service Control Message** and affect the **Local Database**.

Other OPM/Web code mobility models could be plugged and linked into the QoS application for specific purposes. For example, if the **QoS Agent Processing** is required to be able to move or clone itself among various **Router Agencies** according to the Mobile Agent (MA) paradigm, explained in Section 6.3.4, a structural relation between **Router Agency** and itself could be added. This relation would exhibit the **MA Interacting** process as its operation. The simulation of the system would provide for interactive verification of the system already at the early design stage, saving time and human resource efforts.

7. Component-Based Development with OPM/Web[4]

The last few decades have witnessed increasing interest in software reuse, i.e., the use of existing software artifacts or knowledge to create new software [32]. Software reuse aims at improving software quality and productivity by integrating existing components, such as commercial off-the-shelf (COTS) products or tested modules from other projects. Early software reuse concerned the combination of reusable source code components to produce application software [64]. The object-oriented paradigm has highlighted the importance of reusability as part of the entire software system development process by using classes, packages (modules), and the inheritance mechanism as primary linguistic vehicles for reuse [9]. The current definition of software reuse encompasses the variety of resources that are generated and used during the development process, including requirements, architecture, design, implementation, and documentation.

Software engineering approaches refer to reuse in various ways: plug-and-play software component technologies, design patterns, aspect-oriented and superimposition approaches, subject-oriented methods, etc. Many of these reuse approaches rely on the Unified Modeling Language (UML). While UML has undoubtedly contributed to the communicability of software designs during the system development process, two significant drawbacks reduce UML's suitability for component-based development: its model multiplicity and its confused behavior modeling [24]. A major problem with UML is the relatively large number of different views, which require nine diagram types, and the lack of integration and consistency validation among them. On top of this model multiplicity problem, UML lacks a coherent

[4] A short version of this section was published in the 26th annual international Computer Software and Applications Conference (COMPSAC'02) [84]; while an extended version of it is submitted to ACM Transactions of Software Engineering and Methodology (TOSEM).

single mechanism to specify system dynamics. The interaction of processes with external entities is shown in the use case view; their behavior is broken into services, which are distributed among different classes in the class diagram. Their effects on the system's objects are shown in the Statecharts view, while their flows of control are specified in the activity and interaction diagrams. This unbalanced object-oriented, multiple-view system representation complicates the reuse of behavioral components that cut across various object classes. OPM/Web remedies this major obstacle to modeling system dynamics in general and dynamics-rich components in particular by enabling reuse of partially specified components that combine structure and behavior. Web applications can serve as examples of complex structural and behavioral systems whose qualities and delivery times can be improved by reusing existing components.

In this section, an OPM component-based development (CBD), which encompasses the design of generic components and their complex integration with the system under construction, is introduced and exemplified. This process does not fit only for developing Web applications, but any system that requires reuse of existing, partially specified components.

7.1 Reuse of Design Components in Existing Modeling Techniques

Current object-oriented modeling techniques and languages, notably UML [56, 62, 68], emphasize the importance of reuse during the development process and facilitate it through classes, packages, and the inheritance mechanism. Plug and play software component technologies treat classes and packages as closed, black boxes with interfaces, through which other parts of the component or other components can communicate [5]. This approach hinders reusing generic components in different contexts, where they can be effectively woven into various locations throughout an application under development. Moreover, Mezini and Lieberherr [63] claim that object-oriented programs are more difficult to maintain

and reuse because their functionality is spread over several methods, making it difficult to get the "big picture." They suggest designing components that facilitate the construction of complex systems in a manner that supports the evolutionary nature of both structure and behavior.

To respond to this challenge, aspect-oriented programming (AOP) approaches [3, 20] modularize the features for a particular concern and enable these features to be woven, i.e., incorporated and integrated into the system model on the level of programming languages. Superimposition language constructs [4, 10, 11, 49] similarly extend the functionality of process-oriented systems, again cutting across the software architecture of process hierarchy. These techniques allow the imposition of predefined, but configurable, types of functionality on reusable components.

Recently, attempts have been made to extend the aspect notion from programming to software design [7, 19, 50] and to software engineering [2, 42]. Most aspect-oriented modeling techniques are based on UML and employ stereotypes to model the new aspect-oriented concepts. Catalysis [27], for example, is a methodology for component and framework-based development. It enables describing complex systems based on coherent perspectives and views. Catalysis also provides consistency rules across models and mechanisms for composing these views to describe complete systems. Troll [28] and Composition Patterns [16] suggest adding parameterization and binding capabilities to UML packages.

Design patterns [37] describe common design solutions that can be reused in different contexts. They are typically described using a combination of natural languages, UML diagrams, and program code. Since, as noted, UML visualization suffers from unbalanced representation of the static and dynamic aspects of the design patterns [59], UML is extended

using its built-in mechanism of stereotypes. These extensions increase the language vocabulary and complexity, and consequently hinder its comprehension.

Subject oriented design methods try to reduce the gap between the functional-oriented requirements and the object-oriented design models by enabling compositions of models, each of which specifies a user requirement. Clarke [15], for example, has extended UML with two types of composition relations: merge and override. They both involve an entire UML unit – attribute, method, class, etc. As noted by the author, this approach should be customized to each one of the nine UML diagram types, but for now the extension handles only class and sequence diagrams.

While these methods are intuitive for reusing complete structural units, such as classes, packages, or collaborations, they are limited and inconvenient in their support of behavioral and functional component reuse with more intricate relations to an existing system. Furthermore, most of them do not relate to subsequent phases of the system development that need to be accounted for after reusing generic components. Complete integration often requires that certain changes be made to parts of the original component units. This implies that components cannot be black boxes, but rather white or transparent boxes, whose contents can be accessed and modified. This kind of support is often essential for optimizing and enhancing the design of an entire system, a mission that goes beyond binding existing components together. OPM/Web enables weaving partially specified components and evolve them to complete system models by further enhancing and optimizing them during the analysis and design phases. The evolution is made possible through further specification and specialization of the original constituents of the generic components in a way that best suits the task at hand, while maintaining their original core function.

7.2 Weaving OPM/Web Components

The weaving of OPM/Web components is a three-step process, which includes (1) designing reusable generic and target components; (2) integrating them to create raw woven components; and (3) enhancing the raw woven components into complete systems or applications.

7.2.1 Designing Reusable Generic Components

Generic components are the building blocks of the weaving process. For each pair of components to be woven (combined), one component is defined as generic, and is partially specified, while the other is a specific target component. The components are each modeled using OPM/Web.

Each thing (object or process) in OPM exhibits the *affiliation* attribute, which determines if the thing belongs to the system or to the system's environment. A *systemic thing* belongs to (is affiliated with) the system, while an *environmental thing* is either completely external to the system (and interacts with it) or requires further specification in a target component to which it is bound. A compound environmental thing has a partially specified structure and/or behavior, containing at least one environmental element and possibly one or more systemic elements.

In OPM/Web, states and links also exhibit affiliation and hence may also be systemic or environmental. An environmental state is owned by an environmental object, while an environmental (structural or procedural) link connects a pair of environmental entities. As explained in Section 7.2.4 below, an environmental element in a generic component requires the existence of a corresponding element in the target component.

Figure 24 is a model of a generic reusable **Time Stamped Execution** component, which adds time recording capability to a process execution. This component attaches to each **Data Item** a timestamp, called **Recorded Time**. **Data Item** and **Recorded Time** characterize (i.e., are the

attributes of) the environmental object **Node**. **Node** must be environmental, because it exhibits the environmental **Data Item**. **Data Item** itself is environmental since it needs to be refined and adapted to the various contexts in which it is reused. Each object bound (assigned) to **Data Item** must have at least one state that corresponds to the state **created** of **Data Item**. Regardless of the context of **Data Item**, **Recorded Time** is systemic, since it requires no further refinement when woven into the target concrete component.

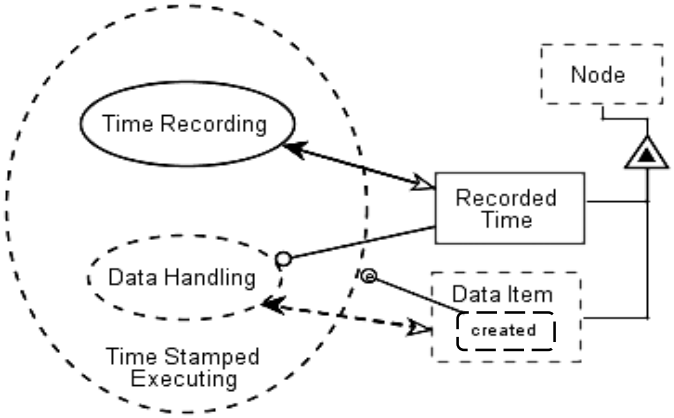


Figure 24. A reusable generic **Time Stamped Execution** component

Similarly, **Time Recording**, which is a sub-process within the **Time Stamped Executing** process, is systemic. **Data Handling**, which should be bound and adapted to a process in the target component, is denoted as an environmental process. The environmental effect link between **Data Handling** and **Data Item** means that in the target component bound to the **Time Stamped Execution** component, a systemic effect link must exist between the process bound to **Data Handling** and the object bound to **Data Item**. This implies that the **Data Handling** process in the **Time Stamped Execution** component must be bound to a process that affects **Data Item**, i.e., a process that changes its value. If **Data Item** and **Data Handling** were not linked in the **Time Stamped Execution** component, **Data Handling** could be bound to a process in the target component that is linked to the object bound to **Data Item** by any possible procedural link, or it might not be linked at all.

The systemic event link from the state **created** of **Data Item** to **Timed Stamped Executing** indicates that the process is triggered each time **Data Item** enters its **created** state. In other words, whenever a new data item is created, the process that records the time of its creation is invoked.

7.2.2 Intra-Model Weaving Rules

In addition to OPM consistency and legality rules, formally defined in Part 4 of this work, OPM/Web components are required to abide by two types of weaving rules: intra-model rules and inter-model rules. Inter-model weaving rules, which are discussed in Section 7.2.4, concern the weaving of two or more components. Intra-model weaving rules, which include the refinement and the link attachment rules defined below, state what can and what cannot be done within a single OPM/Web component.

The refinement rule: The refineables (parts, specializations, features, or instances) of an environmental thing can be either environmental or systemic, while those of a systemic thing can only be systemic. In other words, an environmental thing can be refined (unfolded, in-zoomed, or state expressed) by environmental or systemic entities, while a systemic thing is already fully defined and, hence, can only be refined by other systemic entities.

To see an application of this rule, consider Figure 24, in which **Time Stamped Executing** must be environmental, since it contains the environmental process **Data Handling**. Similarly, the object **Node** must be environmental, since one of its refineables, in this case the attribute **Data Item**, is environmental. The environmental **Data Item** object owns an environmental state, called **created**.

The link attachment rule: An environmental link connects two environmental entities, while a systemic link connects two entities that may be systemic or environmental.

Figure 25 shows the four possible variations of linking two entities in a generic OPM/Web component. Two systemic entities or a systemic entity and an environmental one can be

linked only by a systemic link (as shown in Figure 25(a) and Figure 25(b), respectively). Two environmental entities can be linked by either a systemic or an environmental link (as Figure 25(c) and Figure 25(d) show). A systemic link between two environmental entities (as in Figure 25c) implies that after the binding, the two systemic counterparts which are bound to the linked entities in the generic component must be connected by the systemic link, even though this link may not be present in the original target component. It should be noted that both A and B in Figure 25 can be any type of entity – object, process, or state, and the link between them can be any type of procedural or structural link that can legally connect the two entities.

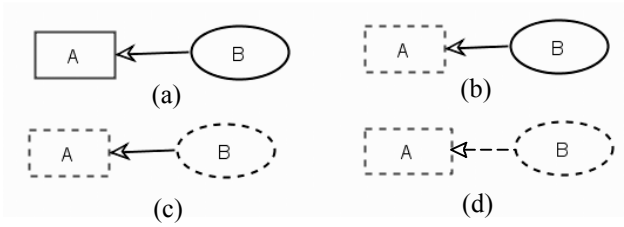


Figure 25. The possibilities of connecting two entities in OPM/Web. (a) Two systemic entities can be linked by a systemic link. (b) An environmental entity and a systemic one can be linked by a systemic link. (c) Two environmental entities can be linked by a systemic link. (d) Two environmental entities can be linked by an environmental link.

7.2.3 Creating Raw Woven Components

Having obtained or created a set of reusable generic OPM/Web components, the system architect should decide which ones are to be woven into the target component and how to weave each one of them so that the resulting specification meets the system requirements. A woven component includes one or more generic or target components, each of which is enclosed in a package. A package is graphically symbolized as in UML (see Appendix A) and is in-zoomed to expose its OPD-set. As noted, in each pair of components to be woven, one component is defined as generic, while the other is the target. A generic component in one combination can be a target in another, and vice versa. The result of the weaving of the

generic and target components is a single woven component, which can be entirely systemic (i.e., all of its entities are systemic, or concrete), or it may still contain one or more environmental entities, implying that it should be further woven with additional components until all the elements are systemic. Only then is the system specification considered complete.

While weaving, the designer has to link one or more environmental entities in the generic component with corresponding (environmental or systemic) entities in the target component. In Figure 26, for example, the generic **Time Stamped Execution** component of Figure 24 is woven into a target **Product Handling** component, such that the combined specification contains two components. Since each component may contain entities that are bound at different levels of refinement, the appropriate series of refining and binding steps needs to be applied in order to get the final specification. In Figure 26, for example, the generic **Time Stamped Execution** component shows **Time Stamped Executing** in-zoomed, while the target **Product Handling** component shows **Product Handling** in-zoomed. This way, the latter process can be bound as a specialization of the former. At the in-zoomed, more detailed level shown in Figure 26, **Product Updating** is bound to **Data Handling** as its specialization.

The generalization-specialization relation is the primary means for binding a thing from the generic component to its counterpart in the target component, although the components can be connected by any (structural or procedural) link. The generalization-specialization (gen-spec) relation in OPM/Web extends its object-oriented counterpart by providing not only for object inheritance, but also for process and state inheritance. As in the object-oriented paradigm, *object inheritance* implies that the sub-object class exhibits at least the same set of features (attributes and operations) and static relations as the super-object class. *Process inheritance* means that the sub-process class has at least the same interface (i.e., the set of procedural links) and behavior (i.e., sub-processes) as the super-process class. The interface and behavior of the inheriting process class may be extended. This way, things can

inherit not just complete classes, as in UML, but also partially specified behaviors. In *state inheritance*, the specialized state inherits the structure (i.e., sub-states) and the interface (i.e., the set of procedural links) in which the generalized state is involved.

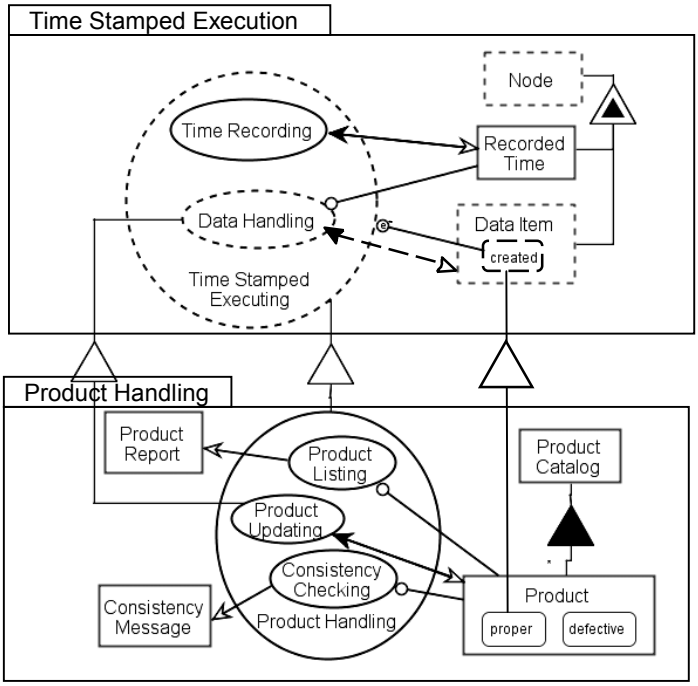


Figure 26. A raw OPM/Web woven component in which the generic **Time Stamped Execution** component (top) is woven into the target **Product Handling** component (bottom)

Each of the three gen-spec relations in Figure 26 binds an entity of the generic component to a corresponding one in the target component. These relations imply that (1) **Product Handling** inherits the systemic **Time Recording** process (which, in turn, affects **Recorded Time**), (2) **Product Updating** inherits the instrument link from **Recorded Time**, and (3) **Product Handling** inherits the event link and is thus triggered when **Data Item** is generated, i.e., when it enters its **created** state. The state gen-spec relation between **created Data Item** and **proper Product** implies that **proper** is a specialization of **created**, such that **Product Handling** is triggered whenever **Product** enters its **proper** state. This relation also implicitly connects **Data Item** to **Product** (implying that **Product** is a **Data Item**).

7.2.4 Inter-Model Weaving Rules

Each woven component is required to preserve the intra-model weaving rules, which are defined in Section 7.2.2, and three inter-model weaving rules, which apply to the integration of two components. The inter-model weaving rules are the mandatory binding rule, the hierarchy congruence rule, and the link abstraction rule, which are defined next.

(1) The mandatory binding rule: This rule concerns binding of three types of elements: entities, links, and states.

(1.1) Entity binding: Each environmental entity (object, process, or state) in a generic component can be bound to a corresponding entity in the target component, either explicitly or implicitly. Explicit binding applies a direct gen-spec relation. In implicit binding, which is applicable only to a compound environmental entity, the gen-spec relation is not visible directly. Rather, it is implied from the context, as follows. For each compound environmental entity **A** in the generic component, which is not explicitly bound to an entity in the target component, a default systemic entity, whose name is **Concrete A**, is automatically generated in the target component. This new entity has a refineables (parts, specializations, features, or instances) set which includes all the systemic entities bound to the environmental refineables of **A**.

As an example, for the unbound environmental object **Node** in the generic component of Figure 26, a default systemic object, whose name is automatically set to **Concrete Node**, is generated in the target component. The object **Concrete Node** exhibits **Product** as its attribute and inherits the systemic object **Recorded Time** from **Node**.

(1.2) Link binding: The binding of an environmental link is implicitly determined from the bindings of the entities it connects.

For example, the environmental effect link between **Data Item** and **Data Handling** in the generic component in Figure 26 is implicitly bound to the systemic effect link between **Product** and **Product Handling** in the target component.

(1.3) State binding: The binding of an environmental state implies the binding of the (environmental) object owning this state.

In Figure 26, for example, **Product** is implicitly bound to **Data Item**, since its **proper** state is bound to the **created** state of **Data Item**.

Figure 27 extends the raw woven component of Figure 26 with the above implicit bindings. One should bear in mind that Figure 27 is only drawn to explicitly illustrate the various bindings, but in practice the two added bindings are implicitly implied from Figure 26 and need not be explicitly specified as in Figure 27.

(2) The hierarchy congruence rule: The hierarchy structure of entities in a target component must be congruent with the hierarchy structure of the corresponding entities that bind them in the target component.

For example, in the woven component of Figure 27, **Node** is bound to **Concrete Node**, while **Data Item** is bound to **Product**. Hence, **Concrete Node** and **Product** of the target **Product Handling** component must preserve the exhibition-characterization relation that exists between **Node** and **Data Item** in the generic **Time Stamped Execution** component. The same congruence exists in the raw woven component in Figure 26, albeit implicitly.

As another example, the systemic **Product Handling** process is bound to the environmental **Time Stamped Executing** process, while the systemic **Product Updating** process is bound to the environmental **Data Handling** process. The in-zooming (containment) relation between **Time Stamped Executing** and **Data Handling** in the generic component is maintained in the target component, as **Product Handling** zooms into (contains) **Product Updating**. The hierarchy congruence rule forbids concurrent binding of **Time Stamped Executing** with **Consistency**

Checking and **Data Handling** with **Product Updating**, because this would violate the hierarchy congruence of the processes required by the generic **Time Stamped Execution** component. Similarly, the congruence of the relation between the **created** state and **Data Item** in the generic component is preserved in the target component by the **proper** state and the **Product** object, respectively.

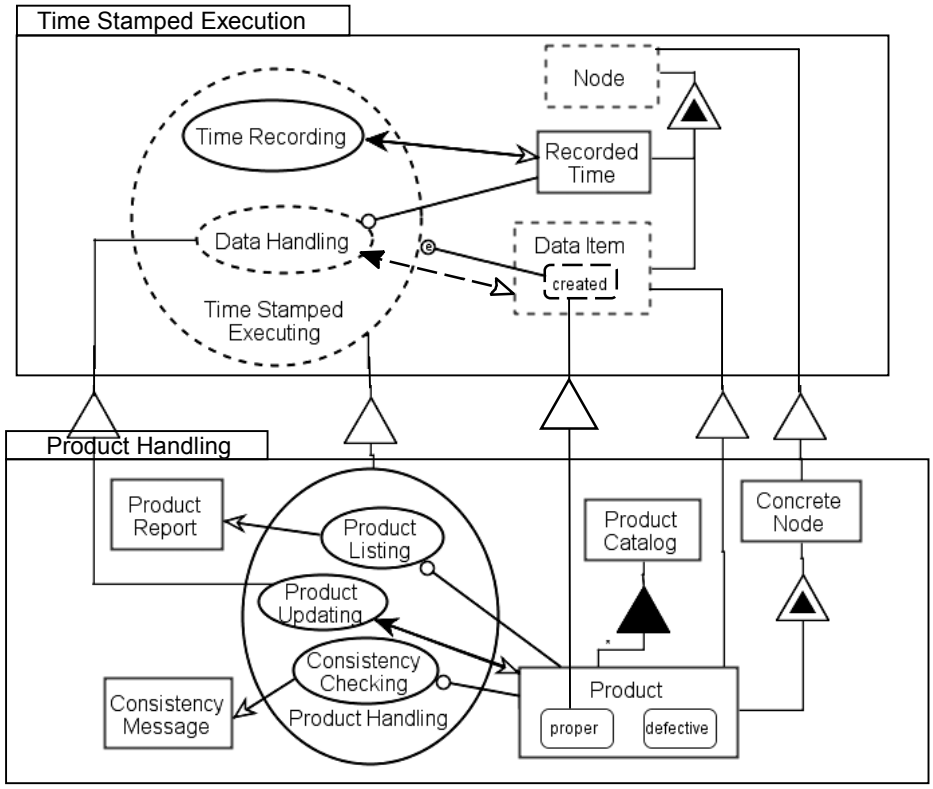


Figure 27. The raw woven component of Figure 26, in which **Concrete Node**, its binding with **Node**, and the binding of **Product** to **Data Item** explicitly appear

(3) The link abstraction rule: Environmental links can be bound to systemic links which are at least as strong as their environmental counterparts according to the link abstraction order. OPM's procedural link abstraction order is summarized in Setion 12.1.3.

As an example, the link abstraction rule implies that the procedural link between **Product** and **Product Updating** in the target component in Figure 26, must be at least as strong as an effect link, which connects **Data Item** and **Data Handling** in the generic component. Hence,

Product and **Product Updating** can not be linked, for example, by instrument or condition links, which are weaker than the effect link.

7.2.5 Merged Components

The semantics of the gen-spec relation between an entity in a generic component and its counterpart in the target component is similar to the semantics of this type of relation between entities within a single component: a target entity inherits the structure and behavior of a generic (environmental) entity.

The target component "absorbs" the generic one, resulting in a *merged component*, which is the single component constructed by applying the weaving rules and the semantics of the raw woven components explained above. By definition, the raw woven component and the merged one, which is derived from it, are equivalent.

Figure 28 is the merged component, which is derived from and equivalent to the woven component in Figure 26. In this merged component, **Concrete Node** exhibits two attributes, **Recorded Time** and **Product**, the latter being part of **Product Catalog**. Zooming into **Product Handling** shows that it consists of four sub-processes: **Time Recording**, **Product Listing**, **Product Updating** , and **Consistency Checking**. **Product Updating**, for example, has an instrument link from **Recorded Time**, required by the generic **Time Stamped Execution** component, and an effect link with **Product**, as the target **Product Handling** component implies. In other words, **Product Updating** uses **Recorded Time** as an input and affects **Product**. The **proper** state of **Product** inherits an event link to **Product Handling** from the environmental **created** state of **Data Item** in the generic **Time Stamped Execution** component.

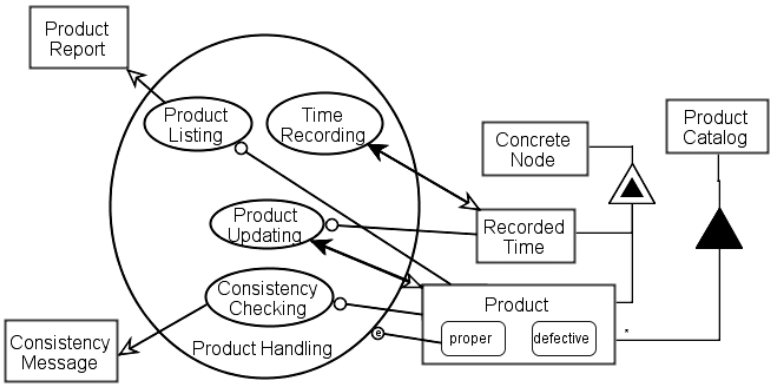


Figure 28. The merged component which is derived from the woven component in Figure 26 and is equivalent to it

The layout of the processes within an in-zoomed process in an OPD defines a partial execution order: two independent or concurrent sub-processes appear at the same vertical level, while sequential processes follow each other. The gen-spec relations between processes of different components merge the partial orders from each component into a single combined partial order. In Figure 26, for example, there is a total order in both the generic **Time Stamped Execution** component (first **Time Recording** and then **Data Handling**) and in the target **Product Handling** component (first **Product Listing**, then **Product Updating**, and finally **Consistency Checking**). The gen-spec relation between **Data Handling** and **Product Updating** in Figure 28 defines a partial order, in which **Time Recording** and **Product Listing** are independently executed first, followed by **Product Updating**, and finally by **Consistency Checking**[6].

If after absorbing the generic component into the target one, more than one link exists between two environmental entities in the generic component and their corresponding entities in the target component, then the more abstract link according to the link abstraction order,

[6] The sub-processes within an in-zoomed process are either parallel or sequential (but not both). Hence, the OPD in Figure 28 is equivalent to an OPD in which **Product Handling** is zoomed into 3 sequential sub-processes: **Pre-Handling**, **Product Updating**, and **Consistency Handling**, where **Pre-Handling** is further zoomed into 2 parallel sub-processes: **Product Listing** and **Time Recording**.

defined in Section 12.1.3, prevails. For example, if in Figure 26 there were a systemic instrument link between **Data Item** and **Data Handling** in the generic component, it would have been subsumed by the effect link between **Product** and **Product Updating** in the merged component, because an effect link is more abstract than an instrument link.

7.2.6 Weaving vs. Merging

A raw woven component can be maintained either as is, or as a merged component. Each option has its advantages and disadvantages. The raw woven component is more succinct and abstract. Its main advantage is the ability to maintain and develop each component separately. Once a generic component is improved, each target component automatically benefits from this improvement. The main advantage of the merged component is its explicit presentation of all the elements from both building components in a single model. However, once merged, this merged component loses its linkage with the generic component, so no change made to the generic component will be reflected in the merged component.

Components can be maintained in libraries that may be distributed over many nodes (computers). According to the flexible usage approach, each time an application is compiled, the most up-to-date components that comprise the application are imported, thereby enabling potential constant improvement in various performance aspects. In another configuration, only future uses of the component benefit from the new, improved version of the component, while existing bindings use an older version of the component with which they were originally complied. To increase flexibility and ensure that the application enjoys potential improvements in generic library components, it is recommended maintaining the woven component version and generating the merged components only to facilitate the readability of the application's OPM/Web model.

7.2.7 Enhancing Raw Woven Components

Having created the raw woven component, the system architect can refer to it as a single component, while further refining the system specification in a separate layer without affecting the generic nature of the composing components. This layer includes the gen-spec relations and additional links among the components. The refinement stage enables optimization of the combined component as a complete application, offering functionality that exceeds the sum of the individual component functionalities. The weaving process can be successively employed in order to reuse additional components for meeting new requirements or modeling various concerns or aspects, as demonstrated next.

7.3 Reusing OPM/Web Components: The Web-Based Accelerated Search Case Study

To demonstrate the weaving process described in Section 7.2, a case study of an OPM/Web component-based development of an accelerated search system is presented. The system implements an algorithm for improving the performance of a Web search engine, which employs time-consuming search algorithms. The design of the accelerated search system includes two components – the generic **Acceleration** component and the target **Multi Search** one. The **Acceleration** component [29] specifies a generic algorithm that reduces the execution time of an input-output part of a system by trying first to retrieve the output, which is determined by the input, from a database. It is assumed that the sought Web-based items rarely change, so they are relatively static. This implies that results of subsequent activations of a query with the same input remain valid and can therefore be stored to avoid executing the costly calculation each time a query with that input is submitted. If the entry is not already in the database, the algorithm activates a process that calculates the sought output and records it in the database to accelerate future executions of the query with the same input. The **Multi Search** component implements a new search engine that benefits from existing search engines by combining their results and ordering them according to a weighted score.

7.3.1 Designing the Acceleration and Multi Search Components

Figure 29 shows the **Acceleration** component. At the top level system diagram (SD) shown in Figure 29(a), **Accelerating** requires **Input** to produce **Output** and affects (updates) the database **DB**. According to Figure 29(b), the **Accelerating** process zooms into (consists of) the following sub-processes: **DB Searching**, which searches the **DB** for an input-output entry; **Output Retrieving**, which retrieves the output if it was found in the database; and **Full Process Activating**, which activates the full-blown process of **Output Computing** otherwise. As Figure 29(c) shows, during **Full Process Activating**, **Output Computing** first computes the output, and then **DB Updating** records the input-output pair in the database for future searches.

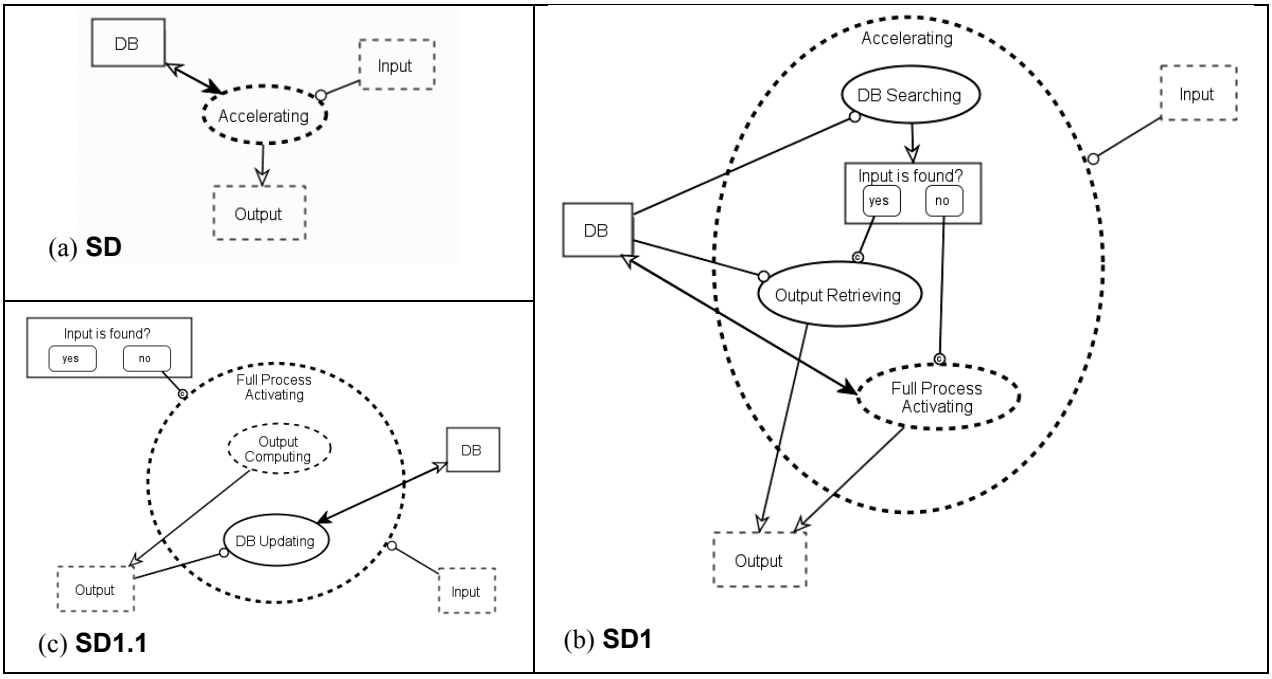


Figure 29. The **Acceleration** component. (a) **SD** is the top-level diagram. (b) **SD1** has **Accelerating** of **SD** in-zoomed. (c) **SD1.1** has **Full Process Activating** of **SD1** in-zoomed.

The **Acceleration** component contains two environmental objects, **Input** and **Output**, and one simple (atomic) environmental process, **Output Computing**. The refinement rule implies that **Full Process Activating** must also be environmental, since it contains the environmental process **Output Computing**. Following the same line of reasoning, **Accelerating**, which is at a yet more abstract level, must also be environmental. All the other things, including the object **DB**, the

Boolean object "**Input is found?**," and the processes **DB Searching** and **Output Retrieving**, are systemic. They are internal to the **Acceleration** component and would not change when this component is woven into a target component.

Figure 30 is a specification of the **Multi Search** algorithm, executed by the **Multi Searching** process. Figure 30(a) is the top-level diagram, which specifies the inputs, **Term** and **Query Result Message**, and outputs, **Query Message** and **Search Result**, of the **Multi Search** algorithm. Assuming that the algorithm operates concurrently on three different search engines, the diagram in Figure 30(b) shows that **Search Starting** requires **Term**, from which it creates three queries, one for each engine. **Query Sending** then creates from the three queries three **Query Messages**, which are sent in parallel to the relevant search engines (the sending part is not included in this model). **Result Collecting** waits until all three replies, called **Query Result Message 1, 2,** and **3**, arrive. It then outputs them as **Result 1, 2,** and **3**, respectively. Finally, **Result Merging** combines these three results to get the **Search Result**.

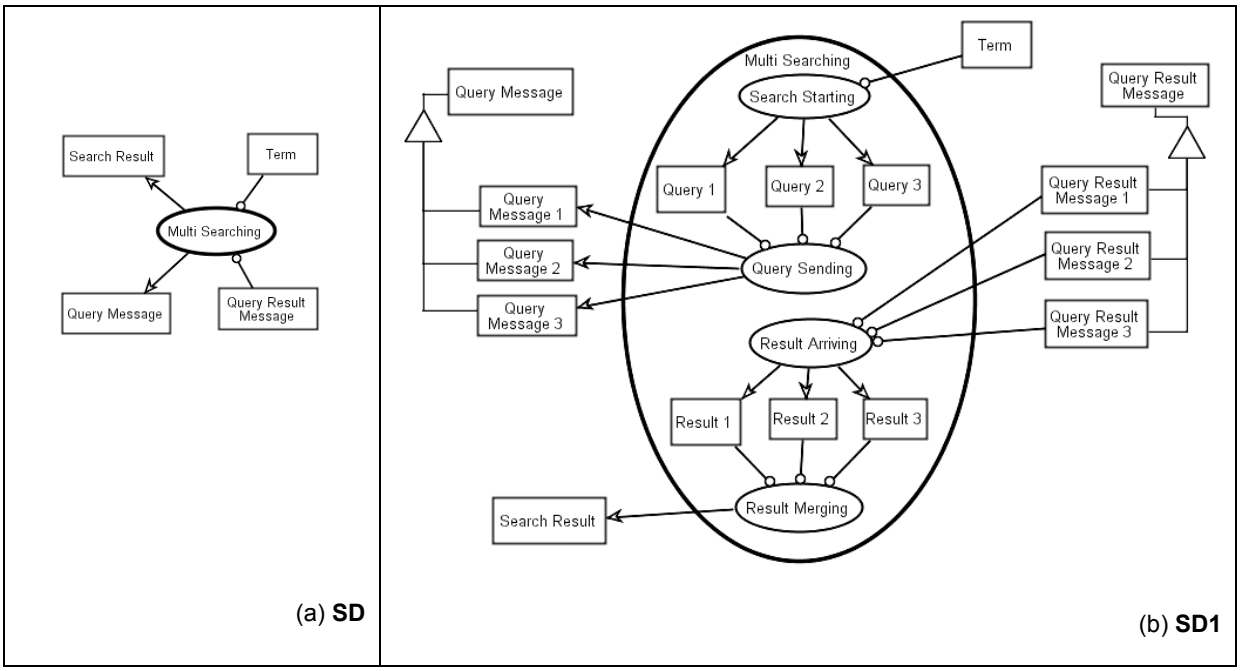


Figure 30. The **Multi Search** component. (a) **SD** is the top-level diagram. (b) **SD1** has **Multi Searching** of **SD** in-zoomed.

The **Multi Searching** process depends on the speed of each search engine, the network response time, and the number and size of results supplied by each search engine. Any combination of these factors can slow down the **Multi Searching** response time. To improve the system's performance, the **Acceleration** component (Figure 29) is woven into the **Multi Search** component (Figure 30). This way, the most recently searched terms and their corresponding results are saved in a local database, and, for each new query, the local database is searched before invoking the entire **Multi Searching** process. Only if the result is not found in the database, the system executes the **Multi Searching** process.

7.3.2 Weaving the Raw Accelerated Multi Search Component

Figure 31 shows the generic **Acceleration** component, derived from **SD1.1** of **Acceleration** in Figure 29(c), woven into **SD** of the **Multi Search** component in Figure 30(a) to create the raw woven component, called **Accelerated Multi Search**. Three gen-spec relations connect the things in the generic component to the corresponding things in the target one. Two of these relations are between object classes, specifying that **Term** is an **Input** and that **Search Result** is an **Output**. The third gen-spec relation is between two process classes, specifying that the systemic **Multi Searching** process specializes the environmental **Output Computing** process.

In each of the three bindings, an atomic thing in the generic component generalizes a corresponding thing in the target component: **Input**, **Output,** and **Output Computing** generalize **Term**, **Search Result**, and **Multi Searching**, respectively. According to the intra-weaving rules, the environmental process **Full Process Activating** is implicitly bound to a default systemic process, called **Concrete Full Process Activating**, which includes just **Multi Searching**.

Merging the **Acceleration** and **Multi Search** components would result in an explicit model, which is equivalent to the woven component modeled in Figure 31. However, this model is both specific to the problem and more complex than the woven component. Furthermore, in order to reuse the acceleration part of this merged component in another system, such as a

system that finds the shortest path between two nodes in a network, the system architect would have to remodel the acceleration functionality to fit the new problem. Conversely, the generic nature of the **Acceleration** component in Figure 29 makes the same core architecture reusable for a variety of related functions. Enhancements to the (non-merged) generic library **Acceleration** component can automatically be reflected in any model into which this component is woven. Physically, the **Acceleration** component may reside in any repository. Assuming constant update of the components is desired, new versions of the **Acceleration** component can be broadcast, published, or pushed to its customer applications whenever it is updated.

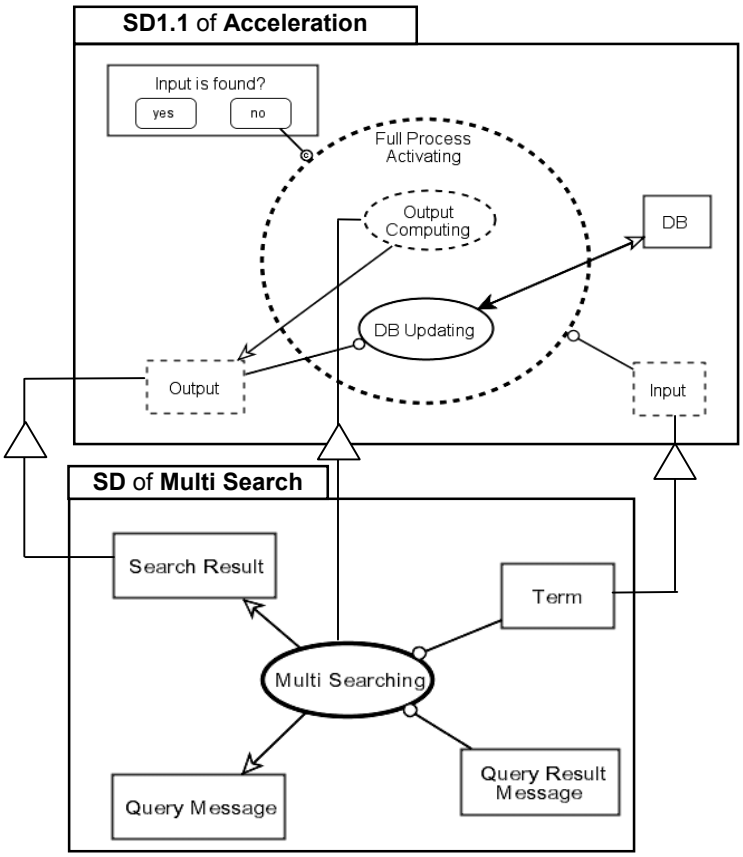


Figure 31. The **Accelerated Multi Search** component

7.3.3 Refining the Raw Woven Accelerated Multi Search Component

The equivalent semantics of the woven and the merged component make it possible to treat woven components as generic or as target components. The system architect can continue

specifying the system into which a component has been woven as a complete application. Two refinements that enhance the raw woven **Accelerated Multi Search** component obtained in Section 7.3.2 are demonstrated. The first refinement improves the **Result Merging** algorithm within **Multi Searching** by treating **DB** as an additional input, while the second one adds to the system an entire generic **Log Recording** component. Any combination of these two refinements can be part of the system design, and they may be incorporated into the system in any order.

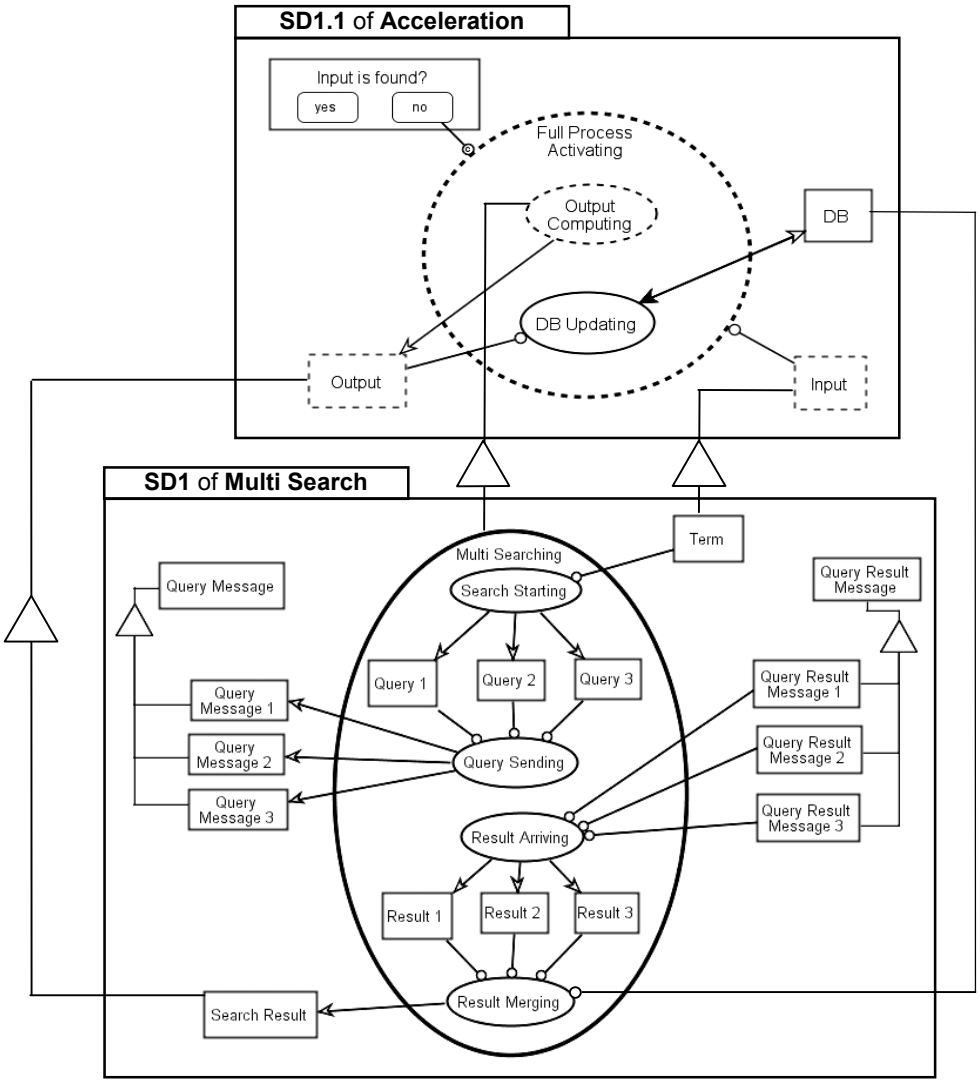


Figure 32. Improving the **Result Merging** algorithm of the **Accelerated Multi Search** component by linking it to **DB**

Improving the **Result Merging** process is achieved by adding the capability to retrieve related term-result pairs from the database and use them to decide how to score the new search results. To carry out this function, an extra instrument link from **DB** (of the generic

Acceleration component) to **Result Merging** (of the specific **Multi Search** component) is added in Figure 32. This makes **DB** an additional input to **Result Merging**. To be able to express this binding, the **Multi Searching** process is shown in **SD1** of the **Multi Search** component (see Figure 30(b)). Further zooming into **Result Merging** would contain details specifying how the database (**DB**), added by the **Acceleration** component, improves the merging of the query results.

Figure 33 shows how the **Accelerated Multi Search** component is enhanced with the ability to maintain a log file. The generic **Log Recording** component includes a systemic **Log File** along with its **Log Records** and a **Recording** operation. The only environmental thing in this component is the object **Input**. When weaving the **Log Recording** component into the **Accelerated Multi Search** component, **Term** is bound to **Input**. The two components are also connected with an event link and an invocation link, denoting the two possible triggers of **Recording**: a **DB** change event triggers **Recording** via the event link, while a **Multi Searching** process termination event triggers **Recording** via the invocation link. The dashed arc between the two events represents a "logical-xor" relation between them.

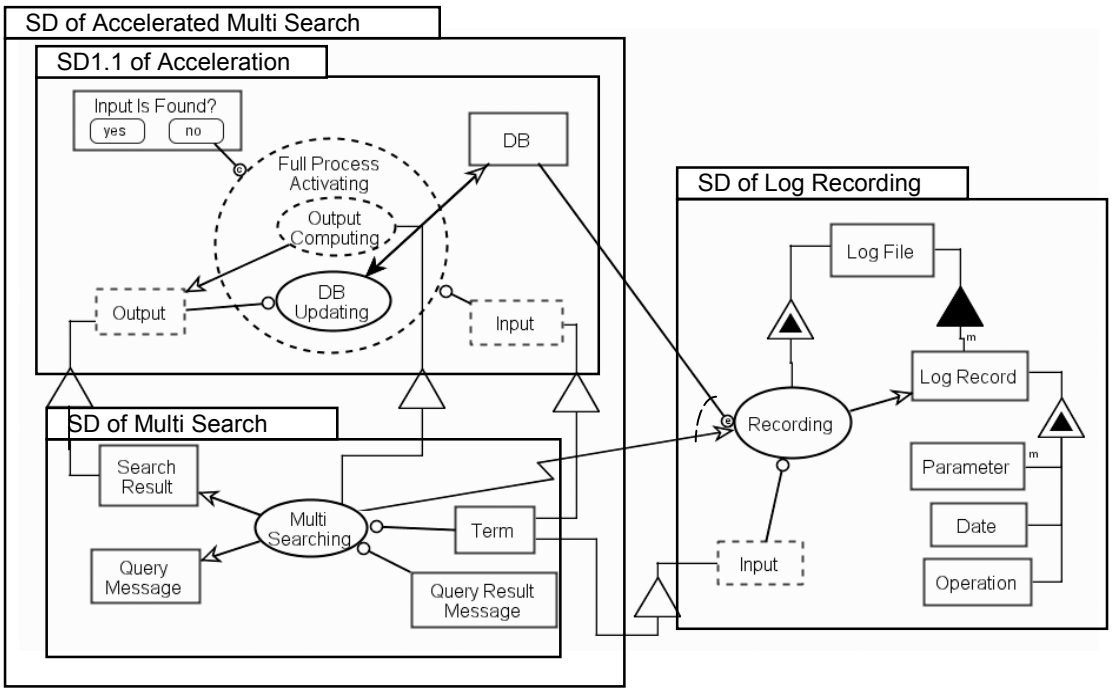


Figure 33. Reusing the **Log Recording** component in the **Accelerated Multi Search** component

Part 3. OPM/Web Evaluation

8. OPM/Web vs. UML – An Experiment[7]

In order to establish the level of comprehension of a given OPM/Web model and the quality of the models constructed using it, OPM/Web was experimentally compared to Conallen's extension of UML to the Web application domain. Third year undergraduate information systems engineering students had to respond to comprehension and construction questions about two representative Web application models. This chapter describes the experiment hypothesis, design, results, and conclusions.

8.1 Comparing Modeling Techniques – Related Work

The emergence of a large number of software system modeling methods over the years has raised the need to evaluate them either theoretically or empirically. Theoretical approaches use techniques, such as metamodeling or defining objective criteria for language comparisons, to examine various attributes of the modeling methods, such as expressiveness, complexity, and accuracy. In one theoretical approach, using metamodeling, Hillegersberg et al. [47] studied the fitness of the Booch method [9], an object-oriented analysis and design method, to the object-oriented programming languages Smalltalk, Eiffel and C++. Halpin and Bloesch [43] examined the relative strengths and weaknesses of the Object Role Model (ORM) and UML for data modeling according to criteria of expressiveness, clarity, semantic stability, semantic relevance, validation mechanisms, abstraction mechanisms, and formal foundation. Siau and Cao [91] compared the complexity metric values of UML with other object-oriented techniques. Their comparison related to seventeen complexity metrics,

[7] An extended version of this section is submitted to the Empirical Software Engineering journal.

including independent metrics (such as the number of object types per technique) and aggregate metrics (e.g., the division of work in a specific technique).

Modeling methods can also be compared empirically. Empirical studies are based on experiments, in which the results are examined quantitatively according to certain criteria, and the subjects may be untrained people, students, or professional systems analysis and design experts. Kim and March [51] compared Extended Entity Relationship (EER) and Nijssen Information Analysis Method (NIAM) [96] for user comprehension, which was measured by counting the number of correct answers to questions about the various modeling constructs. Shoval and Shiran [90] compared EER and object-oriented data models from the point of view of design quality. They measured quality in terms of correctness of the achieved model, time to complete the design task, and designer's preferences for the models. Otero and Dolado [76] compared the semantic comprehension of three different notations for representing behavior in UML: sequence diagrams, collaboration diagrams, and state diagrams. The comparison relied on the total time and score. They found that the comprehension of dynamic modeling in object-oriented designs depends on the complexity of the system. Peleg and Dori [78] compared OPM/T, an extension of OPM to real-time systems, with a variation of Object Modeling Technique (OMT) [86] to the same domain (T/OMT). They examined the specification quality and its comprehension on both the structural and behavioral aspects of a system. The subjects were asked to fill in a questionnaire consisting of statements on a system modeled in one of the methods and to specify another system using the second method. The conclusion was that the single view of OPM/T is more effective than the multiple-view of T/OMT in generating a better system specification. Most of the errors in T/OMT resulted from the lack of integration among the method's different views and the need to maintain consistency and gather information that is scattered across these views. According to the comprehension part of the experiment, a

significant difference was found in favor of T/OPM in three issues: (1) identifying triggering events of processes; (2) identifying processes that are triggered by a given event; and (3) identifying objects that participate in a process. T/OMT, on other hand, was found better in two categories: (1) identifying events that affect objects by changing their state, and (2) determining the order of process execution.

8.2 Experiment Goal, Hypotheses, and Design

Following the empirical approach for evaluating modeling techniques, OPM/Web was compared to Conallen's extension of UML to Web applications [18]. This extension, called Conallen's UML for short, is based on a set of UML stereotypes, tagged values, and constraints, which are commonly used with Web applications, as well as a set of well-formedness rules. The reason for choosing this extension over other UML extensions for the Web application domain was its adoption by the UML standard user community.

The goal of the experiment was to compare OPM/Web to Conallen's UML with respect to two aspects: (1) comprehension: the level of comprehending a given model expressed in each language, and (2) construction: the quality and ease of modeling a given system in each language.

8.2.1 Experiment Hypotheses

The experiment conjecture (the null hypothesis) regarding comprehension was that questions that can be answered by inspecting a single UML view would be more correctly answered using UML than OPM/Web. More specifically, since UML is object-oriented and is comprised of multiple views (diagram types), the UML class diagram would better serve subjects who are looking for answers to questions related to structural parts of a given system. Examples of questions of this type are "What is the database structure of the given application?" and "What are the navigational paths of the pages within the application?"

Conversely, OPM/Web will be preferable for understanding the dynamic aspects of a system and the complex relations among various (structural and dynamic) system components.

With respect to construction, OPM/Web, which uses a single model with three scaling (abstraction-refinement) mechanisms, was expected to be correctly applied more easily than UML for modeling complex, dynamic applications. The reason for this conjecture, in addition to the single- vs. multiple view difference between the two approaches, is that OPM/Web keeps a compact set of graphic alphabet elements, as opposed to the additional element types which Conallen introduced to UML. Conallen's UML defines a relatively large set of no less than 18 domain-specific stereotypes and tagged values, whose syntax and semantics may seem quite obscure. This set includes, among other things, implementation-dependent concepts, such as RMI, IIOP, and Java Script, which make the reuse of the same model in different technology frameworks a difficult task. OPM/Web, on the other hand, extends OPM's expressive power by providing new options for bindings and relations between existing elements, implying new semantics. For example, in OPM/Web a link can exhibit (i.e., be characterized by) features (attributes and operations), while in OPM only things (objects and processes) can exhibit features. Many of these extended capabilities are currently being incorporated into the core OPM.

8.2.2 Population Background and Training

As noted, the subjects of the experiment were third year students at the Technion, Israel Institute of Technology, who took the course "Specification and Analysis of Information Systems" in the spring semester of the 2002 academic year. Most of them were students of the Information Systems Engineering program, which is managed jointly by the Faculty of Industrial Engineering and Management and the Faculty of Computer Science. They had no previous knowledge or experience in system modeling and specification.

Table 5. The syllabus of the course "Specification and Analysis of Information Systems"

Week no.	***Lecture (3 hours per week)***	***Recitation (2 hours per week)***	***Assignment***
1	Introduction – system development lifecycle	Relational databases	
2	DFD	Relational databases	
3	DFD and OO approach	DFD	
4	UML Use Case and Class Diagrams	DFD	Modeling in DFD
5	UML structural model	-----	
6	UML structural model	UML Use Case and Class Diagrams	
7	UML Interaction Diagrams	UML structural model	
8	UML Statecharts	UML Interaction Diagrams	Modeling in UML
9	OPM	UML Statecharts	Modeling in Statechart
10	OPM	OPM	
11	Conallen's UML	OPM	Modeling in OPM
12	OPM/Web	Conallen's UML and OPM/Web	
13	Discussion about analysis approaches	Rehearsal tutorial	

During the 13-week course, the students studied three representative methods: DFD for two weeks, UML for five weeks, and OPM for two weeks. They also studied Conallen's UML and OPM/Web for one additional week each. The students were required to submit four modeling assignments in order to practice the use of DFD, UML – Use Case Diagrams, Class Diagrams, and Sequence Diagrams, UML – Statecharts, and OPM. The syllabus of the course is summarized in Table 5.

The course staff included one adjunct lecturer and two graduate student teaching assistants. They were all familiar with these methods prior to teaching them, but had no particular preference or knowledge of any one specific method.

8.2.3 Experiment Design

The experiment took place during the final three-hour examination of the course. The examination contained three tasks. The two main tasks, which related to Web or distributed

applications and counted for 80% of the examination's grade, constituted the experiment. The third task, which related to DFD, appeared last in the examination, so its effect on the experiment results, if any, was uniform.

The two experimental tasks referred to two case studies: a project management system and a book ordering application. The project management system is a distributed, data-intensive system that handles projects, their tasks, and their intermediate products. The electronic commerce book ordering application is a Web-based system that enables searching for books and ordering them. The students were divided arbitrarily into two groups. The size and tasks of each group are summarized in Table 6.

Table 6. Experiment Design

Task	***Weight***	***Group A*** (40 subjects)	***Group B*** (41 subjects)
Project management system	40%	Conallen's UML	OPM/Web
Book ordering application	40%	OPM/Web	Conallen's UML
DFD technical question	20%	DFD	DFD

To verify that the populations of these two groups were identical, a preliminary t-test was carried out on the grades the students had received in the "Design and Implementation of Information Systems" course, a prerequisite mandatory course in the Information Systems Engineering program. No significant difference was found between the groups ($t = -0.19$, $p < 0.85$).

8.2.4 The OPM/Web and UML Models and Questions

The OPM/Web and UML models of the project management system and the electronic commerce book ordering application and the corresponding questions are presented in Appendix C. Each model consists of five diagrams followed by eight comprehension questions and one modeling problem, which called for extending the system model. The questions on both models for the same case study were identical.

8.3 Results and Discussion

Table 7 summarizes the results (average score) of each question in the examination. Each comprehension question could score a maximum of 3 points (24 points in total for each system), while the modeling problem could score as much as 16 points, totaling 40 points for each system. Incomplete answers, or answers with missing elements, scored less. All the questions about the project management system (in both Conallen's UML and OPM/Web) were graded by one of the two teaching assistants, while the questions about the book ordering application were graded by the other. This grader assignment scheme (together with a strict grading policy) was designed to eliminate any potential bias towards one of the modeling languages in grading the examinations.

Table 7. Experiment Results

Question No.	***Max. Score***	***Project Management System***		***Book Ordering Application***	
		Conallen's UML	**OPM/Web**	**Conallen's UML**	**OPM/Web**
1	3	2.03	**2.82**	**2.65**	1.99
2	3	1.78	**1.88**	1.79	**2.01**
3	3	1.53	**2.70**	1.55	**2.58**
4	3	2.78	**2.85**	1.67	**2.03**
5	3	1.98	**2.49**	1.85	**2.33**
6	3	**2.24**	2.11	2.20	**2.23**
7	3	1.90	**2.16**	1.83	**2.14**
8	3	1.94	**2.40**	2.78	2.78
9	16	9.18	**10.73**	8.32	**8.95**
Total	40	25.33	**30.13**	24.63	**27.01**

As Table 7 shows, most of the questions scored higher when the system was modeled using OPM/Web than when the system was modeled using Conallen's UML. In particular, the construction problems for both systems scored higher when students were required to use OPM/Web.

In the project management system, the only question in which Conallen's UML scored slightly (but not significantly) higher related to representing the navigation order of the

project management pages (question 6: 2.24 for Conallen's UML vs. 2.11 for OPM/Web). However, in the same question about the book ordering system (question 6) the result was reversed in favor of OPM/Web (2.23 for OPM/Web vs. 2.20 for Conallen's UML).

The largest gap in favor of OPM/Web in the project management system was in question 3, which was phrased "*What is the trigger of the project order handling process? From which diagram did you conclude it?*" For this question, students who got the OPM/Web model scored 2.70 (out of 3), almost twice as much as the students who got the Conallen's UML model, who scored 1.53. The reason for this large difference is attributed to the fact that "*project order handling"* is a name of an OPM process which explicitly appears in some of the diagrams, while in the Conallen's UML model the behavior and triggers of this process need to be searched in the sequence diagram.

The next largest gap between the two languages (1.98 vs. 2.49) was in question 5, which was: "*What database classes are affected by the project order handling process? How? (i.e., are they created, destroyed, or changed?)*" Here, the facts that OPM/Web combines the system's structure and behavior in a single view and that the transformation (creation, destruction, or change) of the process on each object is explicitly shown, has helped getting a more correct answer.

In the book ordering application, the only question in which Conallen's UML scored higher (2.65 vs. 1.99) was question 1, which related to the types of pages that the user can view and the information presented at each page. This result in favor of UML is probably due to the fact that Conallen's UML introduces a special element to represent pages or screens, the *boundary entity*, whose symbol and meaning is different than a standard class. OPM/Web, on the other hand, represents Web pages or screens as standard objects with no special symbol. Assigning a special symbol to Web pages helps to quickly and more accurately identify objects of this type. Although the addition of the boundary entity symbol was found

to be helpful in answering this particular question, adding special symbols to denote such elements as components, executables, and libraries, as UML indeed does, comes at the price of unjustifiably complicating the language. Indeed, Siau and Cao [91] found that the UML vocabulary is overall 2-11 times more complex than other object-oriented single view technique.

Table 8. Results of the overall and construction grades – the mixed model

	Factor Name	***Values of Factor***	***Mean Grade (%)***	***F***	***p-level***
Overall Score	Method	Conallen's UML	62.45	54.17	0.0001
		OPM/Web	**71.48**		
	Case	Project management	**69.40**	15.24	0.0002
		Book ordering	64.53		
	Method*Case			1.17	0.2828
Construction Score	Method	Conallen's UML	54.65	7.41	0.0080
		OPM/Web	**61.57**		
	Case	Project management	**62.28**	10.77	0.0015
		Book ordering	53.94		
	Method*Case			0.46	0.4987

Table 8 presents a summary of all the possible effects between the two experiment factors, Method and Case (case study), with respect to (1) the overall grade and (2) the construction question score. The results, normalized to a percentage scale, show significant differences in the main effects (Method and Case) and insignificant differences in their interaction (Method*Case). Both the overall and the construction grades were higher in OPM/Web. The significance of the case study implies that the project management system was less difficult to handle than the book ordering application. Hence, the methods were compared separately for each case study. As Table 9 shows, the overall grades were significantly higher in OPM/Web in both case studies, while the difference of the construction grades was not

significant in the book ordering system and only borderline significant ($p \sim 0.06$) in the project management system.

Table 9. Results of the overall and construction grades according to the case studies

	Case Study	*Method*	*Mean Grade (%)*	*F*	*p-level*
Overall Grade	Project Management	Conallen's UML	63.32	13.57	0.0004
		OPM/Web	**75.33**		
	Book Ordering	Conallen's UML	61.58	4.35	0.0402
		OPM/Web	**67.53**		
Construction Grade	Project Management	Conallen's UML	57.38	3.60	0.0613
		OPM/Web	**67.06**		
	Book Ordering	Conallen's UML	52.00	0.70	0.4062
		OPM/Web	**55.94**		

To gain more insight into the comprehension questions, they were divided into three categories:

1. Structure: questions 1-2 in both case studies, which related to the system structure;
2. Behavior: questions 3-5 in both case studies and question 8 in the book ordering system, which related to the dynamics of the system; and
3. Distribution: questions 6-7 in both case studies and question 8 in the project management system, which related to aspects of the system's distributed nature.

The expectations were that since UML has a separate structure view (the class diagram), it would be better in the structure comprehension category, while OPM/Web will be favorable in the behavior category due to its structure-behavior integration in a single view. It could not be argued in advance in favor of either method with respect to distribution comprehension.

Table 10 summarizes the comprehension results according to the three categories. The students' grades in each category were converted to a binary type, labeled as success when a

student scored more than an average of 2 points (out of 3) in the category, and failure otherwise.

Table 10. Results of the comprehension grades – the GENMOD model

Category	*Case Study*	*Method*	*% Success*	*p-level*
Structure	Project Management	Conallen's UML	35.00	0.09
		OPM/Web	**53.66**	
	Book Ordering	Conallen's UML	**65.85**	0.09
		OPM/Web	47.50	
	Total	Conallen's UML	50.62	0.99
		OPM/Web	50.62	
Dynamics	Project Management	Conallen's UML	70.00	0.02
		OPM/Web	**90.24**	
	Book Ordering	Conallen's UML	56.10	0.001
		OPM/Web	**87.50**	
	Total	Conallen's UML	62.96	<0.0001
		OPM/Web	**88.89**	
Distribution	Project Management	Conallen's UML	30.00	0.005
		OPM/Web	**60.98**	
	Book Ordering	Conallen's UML	34.15	0.09
		OPM/Web	**52.50**	
	Total	Conallen's UML	32.10	0.0009
		OPM/Web	**56.79**	

As noted, one would expect that since UML is object-oriented, and hence focused primarily on structure, it would be significantly better than OPM/Web in the structure category in both case studies. However, Conallen's UML was better only in the book ordering system, while OPM/Web was better in the project management case study. One explanation for this difference might be the higher complexity of the book ordering system compared with the project management application. The students captured the complex structural part of the book ordering system more easily in the UML class diagram, a separate view, which deals with structure only. This does not explain though why OPM/Web was better for the project

management system. Combining the results of the two case studies, the differences between the methods are not significant ($p = 0.99$) with respect to structure. Conversely, with a confidence level of 0.01, one can argue that they are equivalent.

As expected, in the behavior comprehension category, OPM/Web was significantly better ($p < 0.0001$) in both case studies. This outcome might be due to the fact that finding answers to the behavior questions involved consulting several UML views, while in OPM/Web the answers are found in the single diagram type, and the only thing one has to do is traverse across OPDs with different granularity levels. This type of navigation is easier than moving from one type of diagram to another, with each diagram type using its own set of symbols and distinct semantics.

As for the distribution comprehension aspect, in both case studies OPM/Web was significantly better ($p = 0.0009$) than Conallen's UML. This difference can be explained by the fact that the questions in this category involved structural and behavioral aspects of the system distribution that span across different UML views. In other words, in the Conallen's UML model, the students had to integrate information gathered from various diagram types to fully answer these questions, while in OPM/Web diagrams the same information could be achieved by moving from a less detailed diagram to a more detailed (in-zoomed) one.

Although the comparison used a specific extension of UML for Web applications, the findings are quite general. UML's segregation of the system model into multiple views is a major source of difficulty in capturing the system as a whole, understanding its parts, and being able to coherently follow the functionality it performs, as it spans across different diagram types. Moreover, the use of various sets of extensions (stereotypes, tagged values, and constraints), whose syntax and semantics are not universal, weakens UML power as a standard.

Part 4. OPM/Web Metamodel[8]

9. The Metamodeling Technique

A system modeling and development methodology is a combination of a *language* for expressing the universal or domain ontology and an *approach* for developing systems that makes use of this language. Development activities can be divided into three types with increasing abstraction levels: real world, model, and metamodel [46, 95]. The real world is what system analysts perceive as reality or what system architects wish to create as reality. A model is an abstraction of this perceived or contemplated reality that enables its expression using some approach, language, or methodology. A metamodel is a model of a model, or more accurately, a model of the modeling methodology [100]. Metamodels help understand the deep semantics of a methodology as well as relationships among concepts in different languages or methods. They can therefore serve as devices for method development (also referred to as method engineering) and as conceptual schemas for repositories of software engineering and CASE tools. The level of abstraction at which metamodeling is carried out is higher than the level at which modeling is normally done for the purpose of generating a model of a system [46].

The proliferation of object-oriented methods has given rise to a special type of metamodeling, **reflective metamodeling**, in which a methodology is modeled using the means and tools that the methodology itself provides. While metamodeling is a formal definition of the methodology, reflective metamodeling can serve as a common way to check and demonstrate the methodology's expressive power. A reflective methodology, i.e., a methodology that can model itself, is especially powerful since it is self-contained and does not require auxiliary means or external tools to specify itself.

---

[8] The metamodel of an OPM-based development process was accepted to ER'2003 [26].

Existing object-oriented languages, especially UML, have (partial) reflective metamodels. The reflective UML metamodel in [68], for example, includes class diagrams, Object Constraint Language (OCL) [99] constraints, and natural language explanations for describing the main elements in UML and the static relations among them. This metamodel is incomplete in more than one way. First, UML is only a language, not a methodology, so only the language elements are metamodeled, but not any (object-oriented or other) development process. Second, class diagrams are used to model all UML views and the metamodel does not constrain any consistency requirements among the various views of a UML system model. Third, the OCL constraints describes only static invariants.

The Meta Object Facility (MOF) [70] is a standard metadata architecture whose main theme is extensibility and support of metadata. MOF defines four layers of metadata: information (i.e., real world concepts, labeled M0), model (M1), metamodel (M2), and meta-metamodel (M3). The meta-metamodel layer describes the structure and semantics of meta-metadata. In other words, it is an "abstract language" for defining different kinds of metadata (e.g., meta-classes and meta-attributes).

The Meta Modeling Facility (MMF) [17] provides a modular and extensible method for defining and using modeling languages. It comprises a static, object-oriented language (MML), used to write language definitions; a tool (MMT) used to interpret those definitions; and a method (MMM) which provides guidelines and patterns encoded as packages that can be specialized to particular language definitions.

MOF and MMF have been applied to metamodel UML. Since both are object-oriented, they emphasize UML elements and do not deal with expressing the procedural aspects of the methodology, such as its development processes. Several "software process models" have been associated with UML to create complete UML-based methods. One such familiar development process is the Rational Unified Process (RUP) [80]. Although RUP is not a

metamodel-based process, it is specified using class and package diagrams, which specify its structure and its main modules.

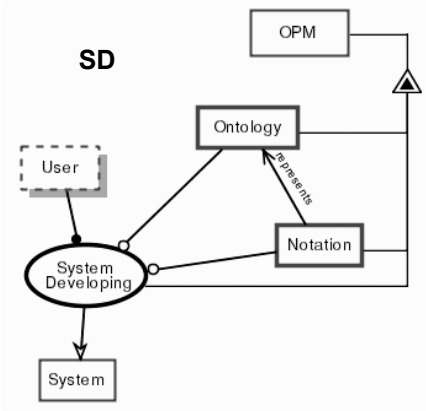
The Software Process Engineering Metamodel (SPEM) [72] uses UML to describe a concrete software development process or a family of related software development processes. Nevertheless, this metamodel does not relate to the process enactment due to the limitations of the UML vocabulary.

The Object-oriented Process, Environment, and Notation (OPEN) [40, 75], as a methodology, offers a set of principles for modeling all aspects of software development across the entire system lifecycle. The development process is described by a contract-driven lifecycle model, which is complemented by a set of techniques and a formal representation using the OPEN Modeling Language (OML) [45]. The lifecycle process, including its techniques, tasks, and tools, is described in terms of classes and their structural relations.

The above metamodels, as well as other metamodels that use structural- or object-oriented methodologies, emphasize the objects and their relations within the metamodel, while the procedural aspects are suppressed and revealed only through operations of objects and the messages passed among them [22]. It is therefore difficult and inconvenient to model a process-oriented component, such as a system development process of a methodology, using an object-oriented approach. OPM overcomes this shortcoming by recognizing processes as stand-alone entities. Moreover, since OPM is reflective, there is absolutely no need for a separate language such as in MOF for specifying the “meta-metamodel” (M4) level. The OPM framework requires only the first three levels: The application level, expressed as an OPM model instance, the model level, expressed as an OPM model, and the (reflective) metamodel level, which refers to OPM as a complex dynamic system and expresses it in OPM terms. This part of the work presents a reflective metamodel of OPM, which includes OPM/Web extensions.

10. OPM Reflective Metamodel – The Top Level Specification

The System Diagram (**SD**), which is the top-level, most abstract specification of the OPM metamodel, is presented in Figure 1. **SD** contains **OPM** and its features, which are the attributes **Ontology** and **Notation**, and the operation **System Developing**. **OPM**, **Ontology**, and **Notation** are objects, symbolized by rectangles, while **System Developing** is a process, as its enclosing ellipse denotes. An exhibition-characterization relation (symbolized by a black triangle within a white one) connects **Ontology**, **Notation**, and **System Developing** to **OPM**, denoting that these objects and process characterize **OPM**. The OPL sentence that corresponds to this graphic ensemble is the first one in the OPL paragraph shown in Figure 1: "**OPM** exhibits **Ontology** and **Notation,** as well as **System Developing.**" Following OPM's graphics-text equivalence principle, the rest of the statements expressed graphically in the OPD are also repeated as natural language statements in the OPL paragraph.



OPM exhibits **Ontology** and **Notation**, as well as **System Developing.**
Notation represents **Ontology.**
System Developing requires **Notation** and **Ontology.**
System Developing yields **System.**
User is environmental and physical.
User handles **System Developing.**

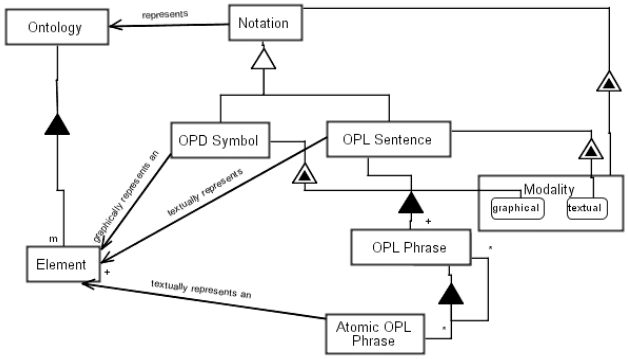
Figure 34. **SD**, the top level specification, of the OPM reflective metamodel

System Developing, which represents the entire OPM-based set of processes, is handled by the **User**, who is the *agent* of **System Developing**. This **User** can be the system architect, developer, or any other stakeholder who uses OPM to architect, develop and evolve a **System**, as well as a team consisting of these stakeholders. The **System Developing** process requires OPM's **Ontology** and **Notation** as instruments (unchangeable inputs) to create a new **System**.

Ontology encompasses OPM elements (entities and links), their features, and the structural and procedural relations among them, but it does not specify anything about the symbols used to denote them. The **Notation** represents the **Ontology** both visually, through interconnected OPD symbols, and textually, through OPL paragraphs and sentences.

Unfolding **Notation**, **SD1** (shown in Figure 35) exposes the detailed relationships between **Ontology** and **Notation**. **Notation** is characterized by **Modality**, which has two possible states (symbolized in the OPD by rounded-corner rectangles): **graphical** and **textual**. An **OPD Symbol** is a **Notation** the **Modality** of which is **graphical**, while an **OPL Sentence** is a **Notation** the **Modality** of which is **textual**.

SD1



Ontology consists of **Elements.**

Notation exhibits **Modality.**

Modality can be **graphical** or **textual.**

Notation represents Ontology.

OPD Symbol is a **Notation,** the **Modality** of which is **graphical.**

OPD Symbol graphically represents an Element.

OPL Sentence is a **Notation,** the **Modality** of which is **textual.**

OPL Sentence consists of at least one **OPL Phrase.**

OPL Phrase consists of optional **OPL Phrases** and optional **Atomic OPL Phrases.**

Atomic OPL Phrase textually represents an Element.

OPL Sentence textually represents at least one **Element.**

Figure 35. **SD1,** in which **OPM Notation** is unfolded

Ontology consists of **Elements**. **Ontology represents Notation** either in a **graphical Modality** via **OPD Symbols** or in a **textual Modality** via **OPL Sentences**. An **OPD Symbol graphically represents** an OPM **Element**, while an **OPL Sentence textually represents** several **Elements**. An **OPL Sentence** may consist of several **OPL Phrases**, each of which can be an **Atomic OPL Phrase** or a complex **OPL Phrase**, i.e., one that consists of other **OPL Phrases**. An **Atomic OPL Phrase textually represents** a single OPM **Element**.

The rest of this part is organized as follows. Chapter 11 presents the reflective metamodel of OPM structure (entities and possible structural and procedural links between them), while chapter 12 presents OPM behavior (complexity management mechanisms and system development processes). Each section includes informal definitions of OPM's concepts, which are exemplified through an ordering system model, and a reflective metamodel of the relevant ontology concepts using OPM notations (both OPDs and OPL sentences).

11. Metamodel of OPM Structure

11.1 Elements

11.1.1 Informal Element Definitions

The OPM ontology consists of two types of elements: entities and links. *Entities* are classified into things and states. A *thing* is a generalization of an object and a process. *Objects* are entities that exist, while *processes* are entities that transform things by generating, consuming, or affecting them. A *state* is a situation at which an object exists. Therefore, a state is not a stand-alone thing, but rather an entity that is "owned" by an object. At any given point in time, the state-owning object is at one of its states. The status of an object, i.e., the current state of the object, is changed solely through an occurrence of a process. Objects and processes are respectively denoted in an OPD by rectangles (as in class diagrams in UML and earlier notations) and ellipses (as in data-flow diagrams). Following Statecharts [44] notation, the OPD symbol of a state is a rounded corner rectangle within the rectangle of its owning object.

A *link* is an element that connects two entities to represent some semantic relation between them. Links can be structural or procedural. A *structural link* is a binary relation between two entities, which specifies a structural aspect of the modeled system, such as an aggregation-participation (whole-part) relation or a generalization-specialization. A *procedural link* connects an entity with a process to denote a dynamic, behavioral flow of data, material, energy, or control. An *event link* is a specialization of a procedural link which models a significant happening in the system that takes place during a particular moment and might trigger a process. Links are denoted in an OPD by lines with different types of arrowheads or triangles, as summarized in Appendix A and explained in Section 11.4.

To exemplify specific points in the OPM ontology and notation, we will use a simple OPM model of an ordering system, shown in Figure 36, in which **Order** is an object, while **Ordering** is a process. **Order** can be at one of two states, **ordered** or **supplied**. **Ordering** changes **Order** from **ordered** to **supplied**. The equivalent two-sentence OPL paragraph in Figure 36 describes the structural relations and procedural links between the entities. The first sentence is a *state enumeration sentence*, which specifies that **ordered** and **supplied** are the two states of the object **Order**. The second sentence is a *change sentence*. It corresponds to the pair of procedural links, the input link (from **ordered** to **Ordering**) and the output link (from **Ordering** to **supplied**). Together, this pair of links change **Order** from its **ordered** state to its **supplied** state.

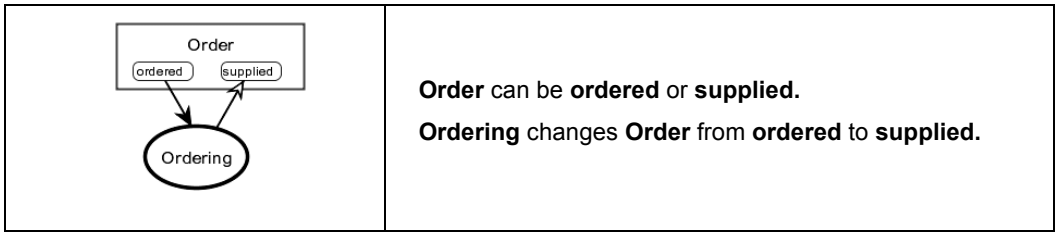


Figure 36. A simple OPM model of objects, processes, states, and links

Any OPM element can be either systemic or environmental. A *systemic element* is internal to the system and has to be completely specified, while an *environmental element* is external to the system and may therefore be specified only partially. The OPD symbol of an environmental element differs from its systemic counterpart in that its borderline is dashed. The **Product Catalog** in Figure 37, for example, is an environmental object, which is external to the system but should be used as an unchangeable input for the **Ordering** process.

In an orthogonal fashion, an OPM element can also be either physical or informatical. A *physical element* is tangible in the broad sense, while an *informatical element* relates to information. A physical entity is symbolized in an OPD as a shadowed closed shape – rectangle, ellipse, or rounded corner rectangle for a physical object, a physical process, or a

physical state, respectively. The **Receipt** in Figure 37 is a physical object resulting from the **Ordering** process.

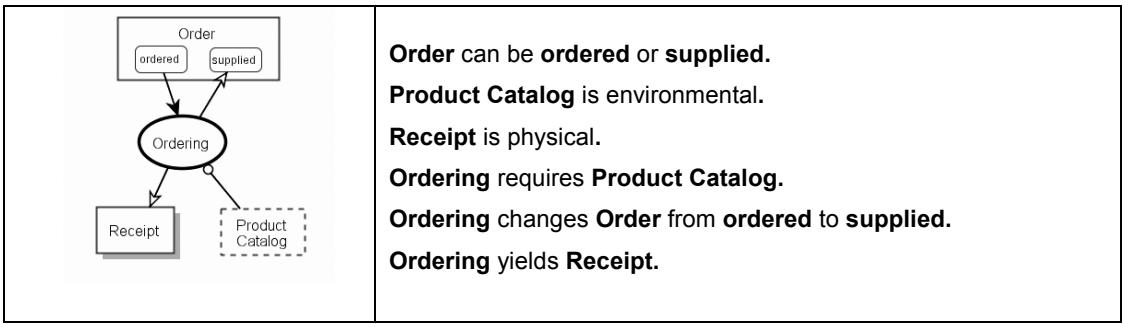


Figure 37. The OPM model of Figure 36 extended with the environmental object **Product Catalog** and the physical object **Receipt**

11.1.2 Element Metamodel

Figure 38 shows the third OPD of the OPM metamodel, labeled **SD2**, in which **Ontology** is unfolded. It specifies that **Ontology** consists of **Entities** and **Links**, each of which is an **Element**. An **Entity**, which exhibits (i.e., is characterized by) a **Name**, specializes into a **Thing** and a **State**. A **Thing** further specializes into an **Object** and a **Process**. The structural relation between an **Object** and a **State** represents that an **Object owns** some **States**, while a **State specifies the status of an Object**.

A **Link** exhibits **Homogeneity**, which is **homogeneous** for a **Structural Link** (that usually connects two **Objects** or two **Processes**) and **non-homogeneous** for a **Procedural Link** (that usually connects an **Entity** and a **Process**).

As noted, each **Element** is characterized by three orthogonal attributes:

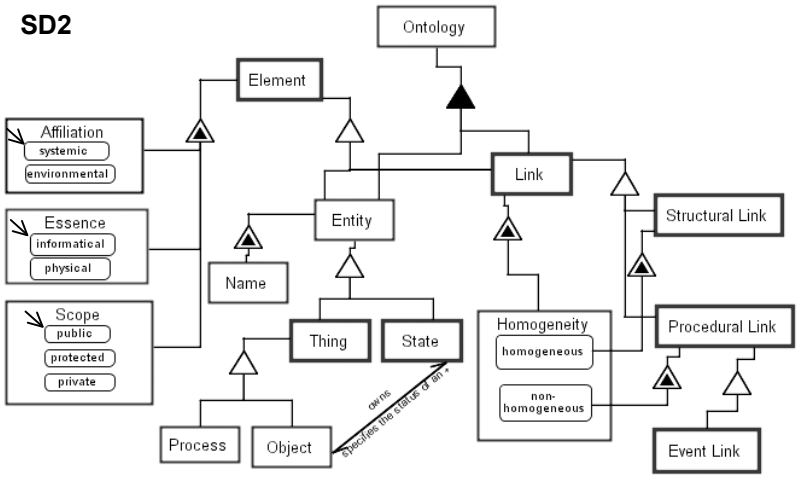
(1) **Affiliation**, which can be **systemic** (the default) or **environmental**;

(2) **Essence**, which can be **informatical** (the default) or **physical**; and

(3) **Scope**, which can be **public** (the default), **protected**, or **private** .

An **environmental Element** is an **Element**, the **Affiliation** of which is **environmental**. An **environmental Element** is external to the system or only partially specified, while a **systemic Element** is internal to the system and completely specified.

As noted, a **physical Element** consists of matter and/or energy. It can be a **physical Object** (e.g., a *Machine*), a **physical Process** (e.g., *Manufacturing*), a **physical State** (e.g., *tested*), or a **physical Link** (e.g., a communication line between two remote computers). An **informatical Element** relates to information. An example of an **informatical Element** is the informatical object *Customer* that stores information about a real-life, physical customer. Other examples include the informatical processes *Database Updating* and *Computation*, and a link between the informatical *Customer* and its *Orders*.



Element exhibits **Affiliation**, **Essence**, and **Scope.**
Affiliation can be **systemic**, which is the default, or **environmental.**
Essence can be **informatical**, which is the default, or **physical.**
Scope can be **public**, which is the default, **protected**, or **private**
Ontology consists of **Entity** and **Link.**
Entity is an **Element.**
Entity exhibits **Name.**
Thing is an **Entity.**
Object is a **Thing.**
Object owns optional **States.**
Process is a **Thing.**
State is an **Entity.**
State specifies the status of an Object.
Link is an **Element.**
Link exhibits **Homogeneity.**
Homogeneity can be **homogeneous** or **non-homogeneous.**
Structural Link is a **Link,** the **Homogeneity** of which is **homogeneous.**
Procedural Link is a **Link,** the **Homogeneity** of which is **non-homogeneous.**
Event Link is a **Procedural Link.**

Figure 38. **SD2,** in which **Ontology** of **OPM** is unfolded

As in programming languages, the **Scope** of an **Element** can be **private** (i.e., it can be accessed only by itself), **protected** (accessible only by itself and its sub-elements), or **public** (accessible by any element in the system). Unlike the object-oriented paradigm, where a method can affect or access only the attributes of the same class, the default **Scope** in OPM is **public**, which implies that any OPM process can use or change all the objects in the model. While seemingly violating the object-oriented encapsulation principle, this provision increases the flexibility of modeling patterns of behavior as OPM processes that involve and cut across several object classes.

11.2 Things

11.2.1 Informal Thing Definitions

A thing is a generalization of an object and a process. An *object* is a thing that exists, at least potentially, and represents a class of instances that have the same structure and can exhibit the same behavior. The **Order** in Figure 39, for example, is a complex object which exhibits three simple attributes (each of which is an object in its own right): **Order Number**, which is of type integer; **Order Date**, which is of type date; and **Order Price**, which is of type float.

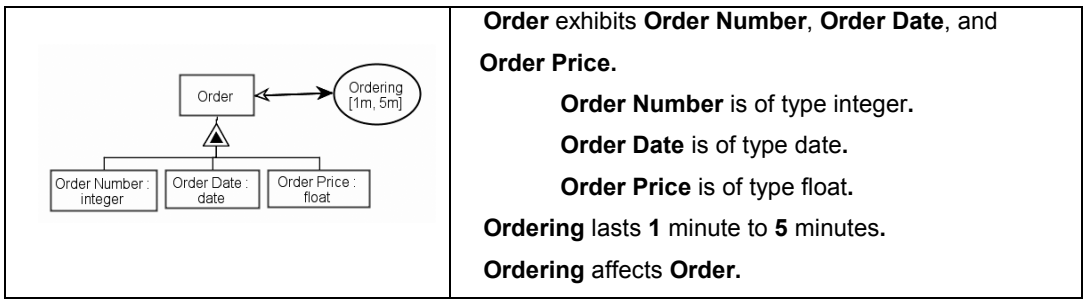


Figure 39. An OPM model of simple and complex objects and time constrained processes

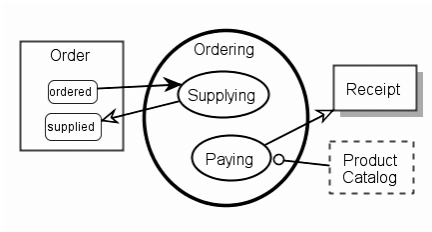
A *process* is a class of occurrences (or instances) of a behavior pattern, which transforms at least one object. Transformation can be change (effect), creation, or consumption of a thing. To carry out the transformation, the process may need to be enabled by one or more objects of different types of classes, which are considered instruments (enablers) for that process and

are not transformed by the process they enable. A process instance is an occurrence (execution) of the specific process and it is analogous to an object instance. The execution time of a process can be constrained by minimal and maximal limits, implying that any process execution can only take a time interval that falls within these time limits. The time limits appear in the OPD as [minimal time constraint, maximal time constraint] within the ellipse representing the process. For example, the specification of the minimal and maximal time limits of the **Ordering** process in Figure 39 implies that it must take at least 1 minute and at most 5 minutes. The corresponding OPL in Figure 39 is "**Ordering** lasts **1** minute to **5** minutes."

A process can be atomic, sequential, or parallel. An atomic process is a lowest-level, elementary action which is not divided into sub-processes, while sequential and parallel processes are refined (through in-zooming or unfolding) into several sequential or parallel sub-processes. The time line in an OPD flows from the top of the diagram downwards, and, hence, the vertical axis within an in-zoomed process defines the execution order: The sub-processes of a sequential process are depicted in the in-zoomed frame of the process stacked on top of each other with the earlier process on top of a later one. Analogously, sub-processes of a parallel process appear in the OPD side by side, at the same height.

In Figure 40, for example, **Ordering** is in-zoomed to show its two sub-processes, which are **Supplying** and **Paying**. In Figure 40(a), **Supplying** and **Paying** are executed in a serial order: **Supplying** is executed first, followed by **Paying**. In Figure 40(b), on the other hand, **Supplying** and **Paying** are executed independently and may occur in parallel. The default execution order is the sequential one, so only the parallel execution order is specified in the fourth OPL sentence in Figure 40(b).

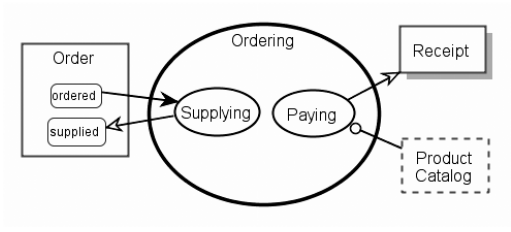
(a)



Order can be **ordered** or **supplied.**
Product Catalog is environmental**.**
Receipt is physical**.**
Ordering zooms into **Supplying** and **Paying.**

Supplying changes **Order** from **ordered** to **supplied.**
Paying requires **Product Catalog.**
Paying yields **Receipt.**

(b)



Order can be **ordered** or **supplied.**
Product Catalog is environmental**.**
Receipt is physical**.**
Ordering zooms into **Supplying** and **Paying**, which are executed in parallel**.**
Supplying changes **Order** from **ordered** to **supplied.**
Paying requires **Product Catalog.**
Paying yields **Receipt.**

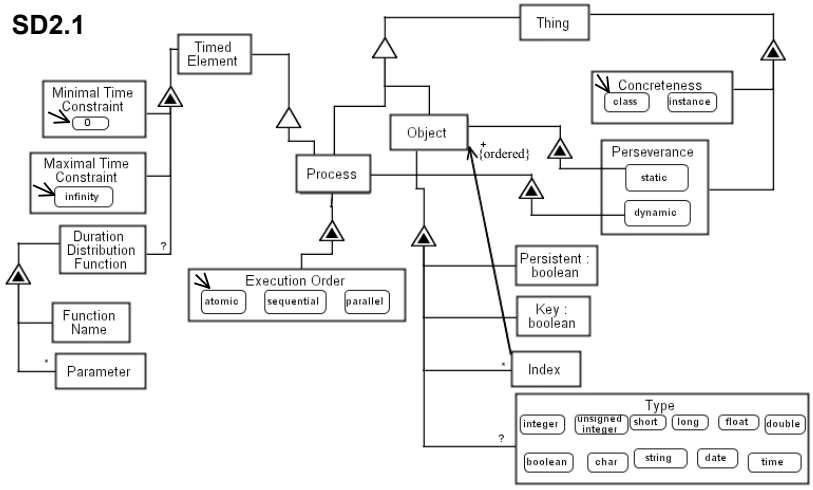
Figure 40. An OPM model of sequential and parallel processes.
(a) **Supplying** and **Paying** are executed serially.
(b) **Paying** and **Supplying** are executed in parallel.

11.2.2 Thing Metamodel

Unfolding **Thing** of the OPM metamodel, **SD2.1** (Figure 41) shows its **Perseverance** attribute, which can be **static** or **dynamic**. An **Object** is a **Thing** with **static Perseverance**, while a **Process** is a **Thing** with **dynamic Perseverance**. In addition to **Perseverance**, a **Thing** also exhibits the **Concreteness** attribute, which determines whether the thing is a **class** (the default) or an **instance**. The difference between an **Object class** and an **Object instance** is similar to the difference between these concepts in the object-oriented approach. A **Process instance** is an occurrence of the process class, which, as noted, is a behavior pattern that the process instances follow. In programming terms, it can be thought of as an executable version of code, which can be executed a specified finite number of times, while a **Process class** is the complete code that can be (re)compiled and executed unboundedly. Following the UML notation of classes and objects, a thing instance is denoted in OPM by a rectangle

or an ellipse, within which the class name is written as **:ClassName**, where the identifier of the instance can optionally precede the colon.

An **Object** can optionally exhibit **Type** (e.g., **integer**, **float**, or **string**), whether it is **Persistent** (i.e., stored in a database), whether it is **Key**, and optional **Indices**. Each **Index** is an ordered tuple of **Objects** (each of which is an attribute).



Timed Element exhibits **Minimal Time Constraint**, **Maximal Time Constraint**, and an optional **Duration Distribution Function.**

Minimal Time Constraint is **0** by default.

Maximal Time Constraint is **infinity** by default.

Duration Distribution Function exhibits **Function Name** and optional **Parameters.**

Thing exhibits **Perseverance** and **Concreteness.**

Perseverance can be **static** or **dynamic.**

Concreteness can be **class**, which is the default, or **instance.**

Object is a **Thing**, the **Perseverance** of which is **static.**

Object exhibits **Persistent**, **Key**, optional **Indices**, and an optional **Type.**

Persistent is of type Boolean.

Key is of type Boolean.

Index relates to an ordered set of at least one **Object.**

Type can be **integer**, **unsigned integer**, **short**, **long**, **float**, **double**, **boolean, char**, **string**, **date**, or **time.**

Process is a **Thing**, the **Perseverance** of which is **dynamic.**

Process is a **Timed Element.**

Process exhibits **Execution Order.**

Execution Order can be **atomic**, which is the default, **sequential,** or **parallel.**

Figure 41. SD2.1, in which Thing of OPM Ontology is unfolded

Process, which is a **Thing** with a **dynamic Perseverance**, is also a **Timed Element** and as such it inherits **Minimal Time Constraint** (0 by default) and **Maximal Time Constraint** (infinity

by default). As noted, these constraints limit the **Process** execution time within the specific bounds. **Process** also inherits from **Timed Element** a **Duration Distribution Function**, which is characterized by **Function Name** and **Parameters**. This attribute specifies the distribution of the process duration that determines how long a process execution lasts and it is most useful for simulation purposes.

In addition, **Process** exhibits **Execution Order**, which can be **atomic**, **sequential**, or **parallel**. Since a process can be either sequential or parallel (but not both), an in-zoomed process will have sub-processes that are all depicted either stacked or in a row, but not as a mixture of these two modes.

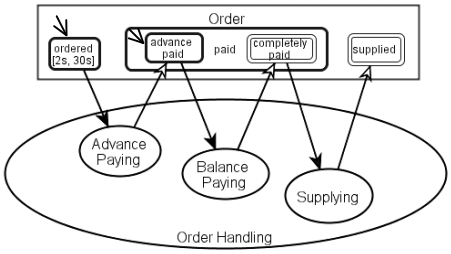
11.3 States

11.3.1 Informal State Definitions

A *state* is a situation in which an object can be for a period of time. It can represent a consecutive value range or a discrete (enumerated) set of values. **Order** in Figure 42, for example, has three top-level, possible states: **ordered**, **paid**, and **supplied**. Both **ordered** and **paid** are initial states, as denoted by the thick borderline rounded corner rectangle. This implies that **Order** can be created in either of its **ordered** or **paid** states. If not otherwise specified, **Order** will be created in its **ordered** state as denoted by the default mark (the small downward diagonal arrow that points towards the **ordered** state). The **supplied** state is the final state of **Order**, as denoted by the double line rounded corner rectangle. When entering this final state, **Order** can be consumed (i.e., destroyed or deleted).

Like process durations, state durations can also be limited on both sides. For example, the **ordered** state of **Order** in Figure 42 has a minimal time limit of 2 seconds and a maximal time limit of 30 seconds, implying that between 2 to 30 seconds must pass from the moment **Order** enters its **ordered** state until it exists this state.

Another similarity of states to processes is that like processes, object states can also be in-zoomed to expose sub-states. In Figure 42, for example, in its **paid** state, **Order** can be at two sub-states: **advance paid**, which is the default of a **paid Order**, or **completely paid**. The in-zoomed processes of **Order Handling** show that **Advance Paying** first changes **Order** from **ordered** to **advance paid**, then **Balance Paying** changes **Order** from **advance paid** to **completely paid**, and finally **Supplying** changes **Order** from **completely paid** to **supplied**.



Order can be **ordered**, which is the default, **paid**, or **supplied.**

Ordered is initial and lasts **2** seconds to **30** seconds**.**

Paid is initial**.**

Paid zooms into **advance paid**, which is the default, and **completely paid.**

Advanced paid is initial**.**

Completely paid is final**.**

Supplied is final**.**

Order Handling zooms into **Advance Paying**, **Balance Paying**, and **Supplying.**

Advance Paying changes **Order** from **ordered** to **advance paid.**

Balance Paying changes **Order** from **advance paid** to **completely paid.**

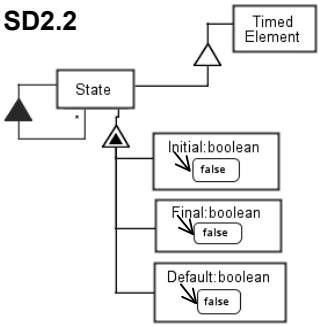
Supplying changes **Order** from **completely paid** to **supplied.**

Figure 42. An OPM model of initial, default, final, and timed states

11.3.2 State Metamodel

As noted, a **State** (or a value) describes a situation at which an **Object** can be (or a value it can assume). Therefore, a **State** cannot stand alone, but is rather "owned" by the (stateful) object. At any given point in time, an **Object** can be at exactly one of the **States** it owns, or in transition between two states. Like a **Process**, a **State** is a **Timed Element**, and as such it exhibits **Minimal Time Constraint** and **Maximal Time Constraint**, i.e., the minimal and maximal bounds for a continuous stay of the owning **Object** in that **State**. As a **Timed Element**, **State** also exhibits **Duration Distribution Function** for simulation purposes.

The OPD labeled **SD2.2** (Figure 43) specifies that a **State** has three additional Boolean attributes: **Initial**, **Final**, and **Default**. **Initial** determines whether the object can be initially (i.e., upon its creation) at this state. **Final** determines whether the object can be consumed (destroyed) when it is at that state. **Default** determines whether this state is the default state (or value) of the owning object, i.e., the state into which the object enters when there is more than one initial state. The aggregation loop attached to **State** indicates that it may recursively consist of lower-level **States**, which are nested sub-states.



State is a **Timed Element.**
State exhibits **Initial**, **Final**, and **Default.**
Initial is of type Boolean and is **false** by default.
Final is of type Boolean and is **false** by default.
Default is of type Boolean and is **false** by default.
State consists of optional **States.**

Figure 43. SD2.2, in which State of OPM Ontology is unfolded

11.4 Links

Links are the "glue" that holds entities (processes and objects with their states) together and enables the construction of system modules of ever growing complexity. OPM links are classified into two types: structural links and procedural links, where the latter are specialized into event links.

As **SD2.3** (Figure 44) shows, a **Link** exhibits two link ends: **Source End** and **Destination End**. Both are specializations of **Link End**, which is characterized by **Participation Constraint** (also known as multiplicity). **Participation Constraint** defines the **Minimal Cardinality** (with 1

as its default value) and the **Maximal Cardinality** (also 1 by default). These specify the minimal and maximal number of instances that can be connected by the link at the corresponding (source or destination) **Link End**. In addition a **Link** exhibits the **Homogeneity** attribute, which has two states: **homogeneous** and **non-homogeneous**. A **Link** is **homogeneous** if both its **Link Ends**, i.e., its **Source End** and **Destination End**, are linked to **Things** whose **Perseverance** value are the same. In other words, a **homogeneous Link** connects either two **Objects** or two **Processes**, while a **non-homogeneous Link** connects an **Entity** to a **Process**. **Structural Links**, which denote static, non-temporal relations between the linked **Entities**, are usually **homogeneous Links**. **Procedural Links**, which model the behavior of the system along time and represent flows of data, material, energy, or control between the linked entities, are **non-homogeneous Links**.

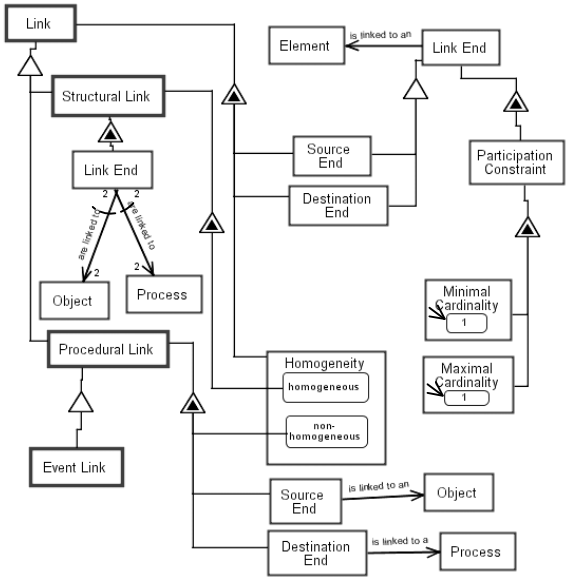
The **Essence**, **Affiliation**, and **Scope** values of a link, inherited from **Element**, are determined by the following three rules, which are formulated in more detail in Appendix D.

Link Essence: A **physical Link** can connect only two **physical Elements**.

Link Affiliation: An **environmental Link** can connect only two **environmental Elements**.

Link Scope: The **Scope** value of a **Link** is the widest of the **Scope** values of the two connected **Elements**, where **public**, **protected**, and **private**, are the widest, intermediate, and most narrow **Scope** values, respectively.

SD2.3



Link End exhibits **Participation Constraint.**
- **Participation Constraint** exhibits **Minimal Cardinality** and **Maximal Cardinality.**
  - **Minimal Cardinality** is **1** by default**.**
  - **Maximal Cardinality** is **1** by default**.**

Link End is linked to an Element.

Link exhibits **Source End, Destination End,** and **Homogeneity.**
- **Source End** is a **Link End.**
- **Destination End** is a **Link End.**
- **Homogeneity** can be **homogeneous** or **non-homogeneous.**

Structural Link is a **Link,** the **Homogeneity** of which is **homogeneous.**

2 Link Ends of **Structural Link are** either **linked to 2 Objects** or **2 Processes.**

Procedural Link is a **Link,** the **Homogeneity** of which is **non-homogeneous.**

Source End of **Procedural Link is linked to an Entity.**

Destination End of **Procedural Link is linked to a Process.**

Event Link is a **Procedural Link.**

Figure 44. **SD2.3,** in which **Link** of **OPM Ontology** is unfolded

11.4.1 Informal Structural Link Definitions

A *structural link*, which is one of the two types of links, denotes a static, time-independent relation between two elements. It usually connects two objects or two processes. Structural links further specialize into tagged structural links, which are general structural links, and four fundamental structural links. A *tagged structural link* can be unidirectional, graphically symbolized by $\rightarrow$, or bi-directional, graphically symbolized by $\leftrightarrows$. It denotes a general, static relation between two objects or two processes, which is labeled by a meaningful forward tag (for the unidirectional link) or a pair of forward and backward tags (for the bi-

directional link). These tags are set by the system architect to convey a meaningful relation between the two linked entities. In Figure 45, for example, **Order** and **Customer** are two objects that are linked with a general structural link tagged "**is placed by**". This link connects an **Order** and its **Customer**. Similarly, **Order** and **Cooperation** are linked with a tagged structural link that is also labeled "**is placed by**".

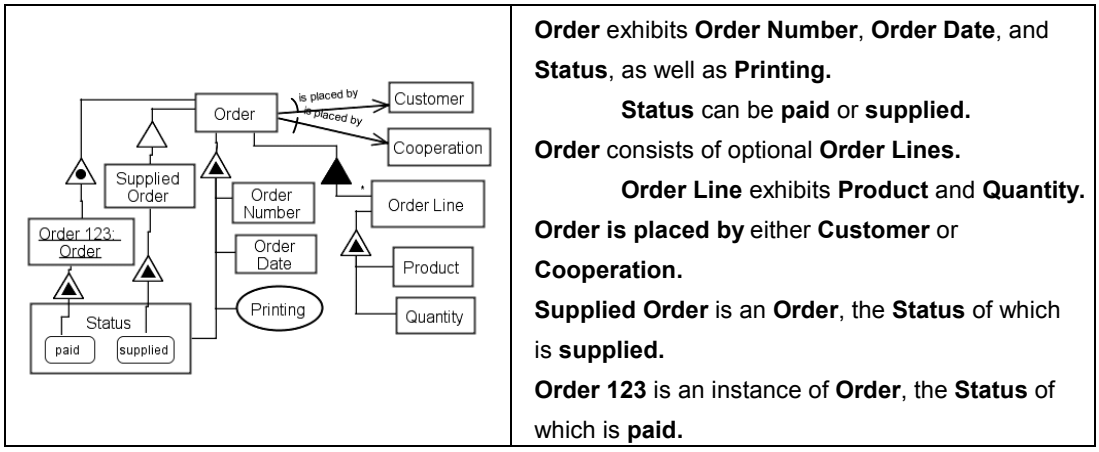


Figure 45. An OPM model with various structural links

The four fundamental structural links are the most prevalent and useful relations and, hence, are assigned various triangular symbols, which are graphically more appealing to the eye than the tag text and save the need to type the nature of the relation each time it is used. The fundamental structural links are:

1. **Aggregation-Participation** denotes the fact that a thing aggregates (i.e., consists of, or comprises) one or more (lower-level) things, each of which is a part of the whole. It is denoted by ▲ whose tip is linked to the whole and whose base is linked to the parts. To achieve the same semantics, we could use "**consists of**" and "**is part of**" as the forward and backward tags of a tagged structural link, respectively, but, as noted, using the black triangle symbol helps distinguish this relation from any other tagged structural relation (and the other three fundamental structural relations). In Figure 45, **Order** consists of 0 or more **Order Lines** (as the * denotes).

2. **Exhibition-Characterization** denotes the fact that a link or a thing exhibits, or is characterized by, another lower-level thing. The exhibition-characterization symbol is ◬. The features (which can be attributes or operations) are connected to the base of the triangle. In Figure 45, **Order** exhibits (is characterized by) the attributes **Order Number**, **Order Date**, and **Status** and the operation **Printing**, while **Order Line** exhibits **Product** and **Quantity**.

3. **Generalization-Specialization (Gen-Spec)** is a fundamental structural relation between two entities: the specialized entities share common structural and procedural links with the generalized entity. The symbol of the gen-spec relation is △ whose tip is linked to the generalizing entity and its base – to the specialized entities. In Figure 45, **Supplied Order** defines a sub-class of **Orders** whose **Status** is **supplied**. Similar to an **Order**, a **Supplied Order** has its **Order Number**, **Order Date**, **Status**, **Order Lines**, and owning **Customer** or **Cooperation**, and it can execute the operation **Printing**.

4. **Classification-Instantiation** represents a fundamental structural relation between a class of things and an instance of that class. This type of link is denoted by ◬. The tip of the shape is linked to the class, while its base – to the instances. **Order 123** in Figure 45 is an instance of an **Order** whose **Status** is **paid**.

Structural links of the same type can be connected by "or" and "xor" relations to specify alternative structures. An "or" relation is symbolized by a double line, dashed arc connecting the relevant structural links, while a "xor" relation is denoted by a single line, dashed arc. In Figure 45, for example, an **Order is placed by** either a **Customer** or **Cooperation**, but not by both. If there were no arc in the specification, a specific **Order** would have an owning **Customer** *and* an owning **Cooperation**.

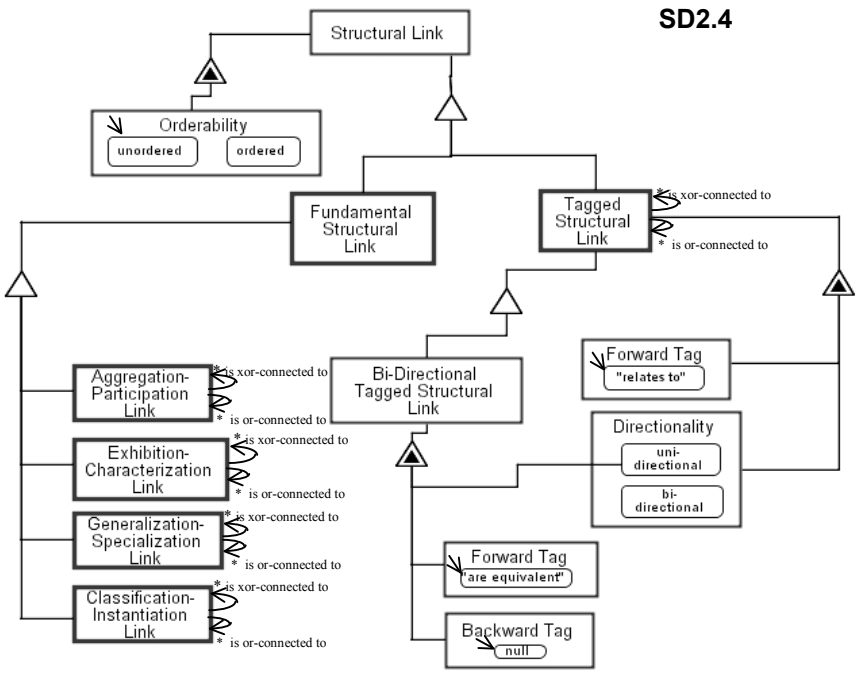
11.4.2 Structural Link Metamodel

SD2.4 (Figure 46) unfolds OPM **Structural Links**. A **Structural Link** is characterized by **Orderability**, which can be **ordered** (e.g., an array) or **unordered** (e.g., a set) by default. An **ordered Structural Link** adds the reserved label {ordered} next to the **Object** classes whose instances are required to be ordered. In Figure 41, for example, **Object** is characterized by optional **Indices**, each of which is an ordered set of **Objects**.

SD2.4 also unfolds the two types of **Structural Links**: **Tagged Structural Links** and **Fundamental Structural Links**. A **Tagged Structural Link** exhibits **Forward Tag**, whose default value is the string "**relates to**", and **Directionality**. A **Bi-Directional Tagged Structural Link**, which is a **Tagged Structural Link** whose **Directionality** is **bi-directional**, exhibits in addition **Backward Tag**, whose default value is **null**, and the default value of its **Forward Tag** is "**are equivalent**".

Fundamental Structural Links are specialized into **Aggregation-Participation Links**, **Exhibition-Characterization Links**, **Generalization-Specialization Links**, and **Classification-Instantiation Links**.

As noted, **Structural Links** of the same type can be connected by "or" and "xor" relations. This is specified by the self tagged structural links labeled "**is or-connected to**" and "**is xor-connected to**", respectively.



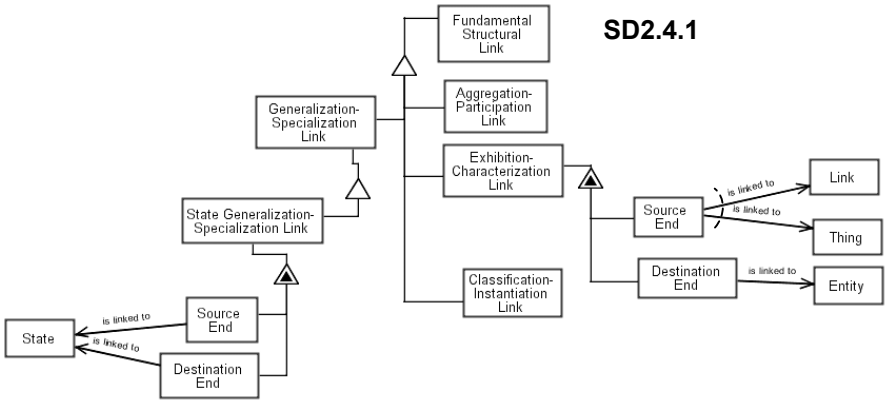
Structural Link exhibits **Orderability.**
Orderability can be **unordered**, which is the default, or **ordered.**
Tagged Structural Link is a **Structural Link.**
Tagged Structural Link exhibits **Forward Tag** and **Directionality.**
Forward Tag is **"relates to"** by default.
Directionality can be **uni-directional** or **bi-directional.**
Tagged Structural Link is xor-connected to optional **Tagged Structural Links.**
Tagged Structural Link is or-connected to optional **Tagged Structural Links.**
Bi-Directional Tagged Structural Link is a **Tagged Structural Link**, the **Directionality** of which is **bi-directional.**
Bi-Directional Tagged Structural Link exhibits **Forward Tag** and **Backward Tag.**
Forward Tag is **"are equivalent"** by default.
Backward Tag is **null** by default.
Fundamental Structural Link is a **Structural Link.**
Aggregation-Participation Link is a **Fundamental Structural Link.**
Aggregation-Participation Link is xor-connected to optional **Aggregation-Participation Links.**
Aggregation-Participation Link is or-connected to optional **Aggregation-Participation Links.**
Exhibition-Characterization Link is a **Fundamental Structural Link.**
Exhibition-Characterization Link is xor-connected to optional **Exhibition-Characterization Links.**
Exhibition-Characterization Link is or-connected to optional **Exhibition-Characterization Links.**
Generalization-Specialization Link is a **Fundamental Structural Link.**
Generalization-Specialization Link is xor-connected to optional **Generalization-Specialization Links.**
Generalization-Specialization Link is or-connected to optional **Generalization-Specialization Links.**
Classification-Initialization Link is a **Fundamental Structural Link.**
Classification-Initialization Link is xor-connected to optional **Classification-Initialization Links.**
Classification-Initialization Link is or-connected to optional **Classification-Initialization Links.**

Figure 46. **SD2.4** in which **Structural Link** of **OPM Ontology** is unfolded

SD2.4.1 (Figure 47), which unfolds the **Fundamental Structural Links**, specifies constraints on the **Elements** that can be connected by this type of links. Being **Structural Links**,

Fundamental Structural Links connects two **Objects** or two **Processes**. There are two exceptions to this simple rule specified in **SD2.4.1**:

1. An **Exhibition-Characterization Link** connects a **Thing** or a **Link** (as its **Source End**) and an **Entity** (as its **Destination End**). For example, the communication link between remote computers, which is modeled as a **Tagged Structural Link**, can be characterized by the object *Transfer Rate* and/or the process *Encrypting*. A **Paid Order** is characterized by a **Status** attribute whose value (state) is **paid**.
2. A **Generalization-Specialization Link** can connect two **States** of different **Objects** to represent state inheritance. In this type of link, which is called **State Generalization-Specialization Link**, the inherited state has at least the same structural and behavioral links as the inheriting state.



Aggregation-Participation Link is a **Fundamental Structural Link.**
Exhibition-Characterization Link is a **Fundamental Structural Link.**
Source End of **Exhibition-Characterization Link is linked to** either **Link** or **Thing.**
Destination End of **Exhibition-Characterization Link is linked to Entity.**
Generalization-Specialization Link is a **Fundamental Structural Link.**
State Generalization-Specialization Link is a **Generalization-Specialization Link.**
Source End of **State Generalization-Specialization Link is linked to State.**
Destination End of **State Generalization-Specialization Link is linked to State.**
Classification-Instantiation Link is a **Fundamental Structural Link.**

Figure 47. **SD2.4.1**, in which **Fundamental Structural Link** of **OPM Ontology** is unfolded

Table 1 summarizes the possible structural relations between OPM elements in a tabular way.

Table 1. Possible structural relations between OPM elements. S and D denote the link source and destination, respectively. + denotes a legal link.

Tagged Structural Link / Aggregation-Participation Link					**Exhibition-Characterization Link**				
S / **D**	Object	Process	State	Link	**S** / **D**	Object	Process	State	Link
Object	+	-	-	-	Object	+	+	-	+
Process	-	+	-	-	Process	+	+	-	+
State	-	-	-	-	State	+	+	-	+
Link	-	-	-	-	Link	-	-	-	-
Generalization-Specialization Link					**Classification-Instantiation Link**				
S / **D**	Object	Process	State	Link	**S** / **D**	Object	Process	State	Link
Object	+	-	-	-	Object	+	-	-	-
Process	-	+	-	-	Process	-	+	-	-
State	-	-	+	-	State	-	-	-	-
Link	-	-	-	-	Link	-	-	-	-

11.4.3 Informal Procedural Link Definitions

A procedural link represents a dynamic relation between a process and a thing. It has one of the following meanings: (1) the thing enables the process, (2) the process transforms the thing, or (3) the thing triggers the process. Accordingly, procedural links are divided into enabling links, transformation links, and event links. An enabling link, called an *instrument link*, is a procedural link that connects a process with an enabler of that process. The enabler is a thing that must be present in order for that process to occur, but it is not transformed as a result of the process occurrence. The instrument link can originate from an object, a process, or a state, denoting that the object existence, the process existence, or the object in the specific state is the enabler, respectively. Graphically, an instrument link is symbolized by —○, while textually it is represented by the reserved word "requires". In Figure 48, for example, **Product Catalog**, which is a systemic object in this model, is required for the **Ordering** process. Nevertheless, the occurrence of **Ordering** does not affect **Product Catalog** in any way. Therefore, **Product Catalog** is an instrument of the process **Ordering**. However, for another process, such as **Catalog Updating**, **Product Catalog** would be an affectee, an object

affected by **Catalog Updating**. Hence, being an instrument can be though of as a "role" of an object class with respect to a particular process class.

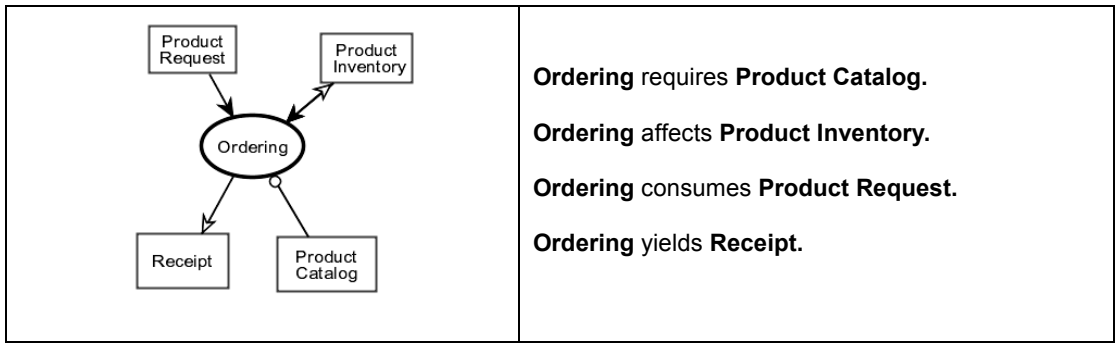


Figure 48. An OPM model with various procedural links

A *transformation link* denotes that a thing is transformed by the occurrence of a specific process. Transformation is a generalization of consumption, result, and effect. A *consumption link* is a transformation link that connects an entity to a process which consumes it. A consumption link is denoted by ⟶ from the consumed entity to the process, while the reserved word "consumes" represents it in OPL. In Figure 48, for example, **Product Request** is consumed by the process **Ordering**. In other words, **Product Request** had existed before an occurrence of **Ordering**, disappeared (was destroyed) during this execution, and does not exist after **Ordering** is finished. A consumption link originating from a state of an object means that the process consumes that object only when the object is in that specific state.

A *result link* is a transformation link that denotes a creation of a thing or an object at a specific state. It is symbolized in an OPD by ⟶▷ from the process to the resultant entity, while the reserved word "yields" denotes it in OPL. In Figure 48, **Ordering** creates a **Receipt**: the **Receipt** had not existed before the beginning of **Ordering**, it was created during this execution, and it exists after **Ordering** is finished.

An *effect link* connects a process with a thing that is affected, i.e., undergoes a state (or value) change, during that process. The thing had existed before the process occurred and it keeps existing after the process was finished, but at least one of its states or attribute values

changed. The effect link is denoted in an OPD by ◁⟶ where the white-headed arrow pointing towards the affected thing and the black-headed arrow – towards the process. This reduces a potential ambiguity when an effect link connects two processes. OPL uses the reserved word "affects" to represent effect links. In Figure 48, for example, **Ordering** affects **Product Inventory**, which is a systemic object that represents the quantity of a product. This object had existed before an occurrence of **Ordering**, and it exists thereafter, but its value before the process occurrence is different from the value afterwards. Assuming that **Ordering** reduces **Product Inventory** by 1, Figure 49 is a refinement of this OPM model in which **Product Inventory** is state expressed to expose two states: **value** and **value-1**. The effect link is also refined through its split into an input link from **value** to **Ordering**, and an output link from **Ordering** to **value-1**. Overall, the meaning of this phrase is that **Ordering** changes the state of **Product Inventory** from **value** to **value-1**. Input and output links can be though of as specializations of consumption and result links respectively: the process "consumes" the input state and "yields" the output state. However, the object as a whole is neither consumed nor generated – it merely changes its state (or its value).

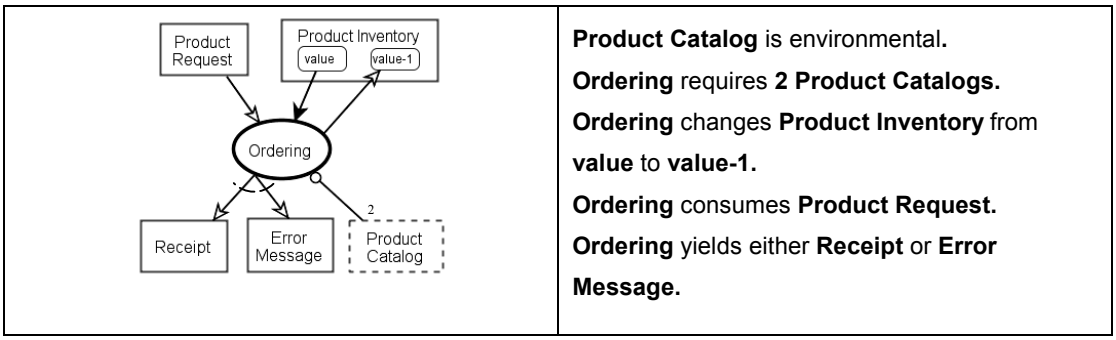


Figure 49. A refined OPM model of effect links as consumption and result links

Like structural links, procedural links can have multiplicity constraints, they can be connected by "or" and "xor" relations, and they can exhibit attributes, such as a set of conditions governing or guarding the link. For example, in Figure 49, **Ordering** requires 2 **Product Catalogs**, while consuming one (the default, when no multiplicity constraint is

indicated) **Product Request** and affecting one **Product Inventory**. **Ordering** yields either one **Receipt** or one **Error Message**.

A procedural link may have one or more path labels. A *path label* is a character string label on a procedural link that removes the ambiguity arising from multiple procedural links outgoing from the same entity. When procedural links that originate from an entity are labeled, the one that must be followed is the one whose label is identical with the label of the procedural link that arrives at the entity. The path labels in Figure 50, for example, specify two possible scenarios of **Order Handling**: Symbolized by the path label **paying**, this process occurs when **Order** is at its **ordered** state, and it changes the state of **Order** to **paid**. Symbolized by the path label **supplying**, the process occurs when **Order** is at its **paid** state, and it changes the state of **Order** to **supplied**. A path label can be put also on an enabling link, indicating that the thing attached to the link bearing the label is required for the process. For example, the path label **supplying** along the instrument link from **Warehouse** to **Order Handling** indicates that **Warehouse** is required when the state of **Order** changes from **paid** to **supplied**, but not when it changes from **ordered** to **paid**.

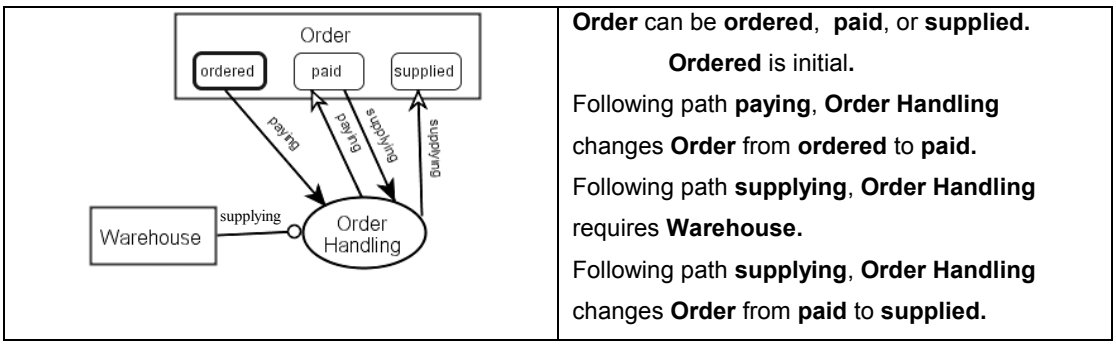
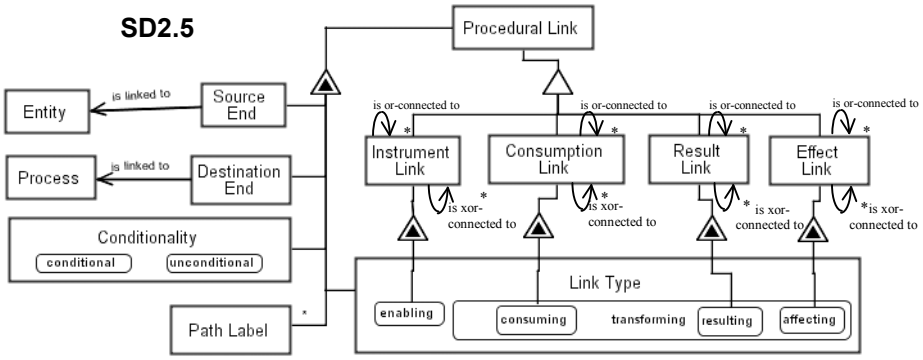


Figure 50. An OPM model with path labels on procedural links

11.4.4 Procedural Link Metamodel

Any **Procedural Link** has a **Process** as its **Destination End**, while its **Source End** is connected to an **Entity**. As shown in **SD2.5** (Figure 51), a **Procedural Link** exhibits three attributes: **Link Type**, **Conditionality**, and optional **Path Labels**. The **Link Type** of a **Procedural Link**

distinguishes primarily between **enabling** and **transforming Procedural Links**. **Transforming Procedural Links** are further divided into **affecting**, **consuming**, and **resulting Procedural Links**.



Procedural Link exhibits **Link Type**, **Conditionality**, and optional **Path Labels.**
Link Type can be **enabling** or **transforming.**
Transforming zooms into **affecting**, **consuming**, or **resulting.**
Conditionality can be **conditional** or **unconditional.**
Source End of **Procedural Link is linked to Entity.**
Destination End of **Procedural Link is linked to Process.**
Instrument Link is a **Procedural Link**, the **Link Type** of which is **enabling.**
Instrument Link is xor-connected to optional **Instrument Links.**
Instrument Link is or-connected to optional **Instrument Links.**
Consumption Link is a **Procedural Link**, the **Link Type** of which is **consuming.**
Consumption Link is xor-connected to optional **Consumption Links.**
Consumption Link is or-connected to optional **Consumption Links.**
Result Link is a **Procedural Link**, the **Link Type** of which is **resulting.**
Result Link is xor-connected to optional **Result Links.**
Result Link is or-connected to optional **Result Links.**
Effect Link is a **Procedural Link**, the **Link Type** of which is **affecting.**
Effect Link is xor-connected to optional **Effect Links.**
Effect Link is or-connected to optional **Effect Links.**

Figure 51. SD2.5, in which **Procedural Link** of **OPM Ontology** is unfolded

A **conditional Procedural Link**, i.e., a **Procedural Link** whose **Conditionality** is **conditional**, enables the **Process** execution only if the condition it symbolizes holds, else the **Process** is skipped and the next process in turn is examined for possible execution. With the exception of **Result Link**, each type of procedural link can be a **conditional Procedural Link** or **unconditional Procedural Link**. A **Result Link** cannot be a **conditional Procedural Link** because

the **Entity** which the **Process** generated upon its completion cannot be a condition for the **Process** that generated it. For simplicity, this assertion is defined in Appendix E by the OCL constraint (1).

The difference between an enabling link and a conditional enabling link is demonstrated in Figure 52. The **Process** in Figure 52(a) waits until the **Object** is at the **state**. In Figure 52(b), on the other hand, the **Process** is executed only if the **Object** is at the **state**. Otherwise, the control is passed to the next process that should be executed. The OPD symbol of a **conditional Procedural Link** is similar to its non-conditional counterpart, except that it has the letter "c" inside or next to the link symbol (see Appendix A). In OPL, an occurrence sentence is added for a **conditional Procedural Link**, as demonstrated in Figure 52(b).

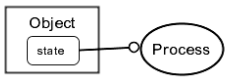
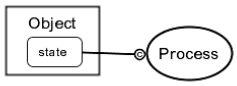
(a) Instrument link Semantics: **Process** waits until **Object** is at **State.**	 **Process** requires **state Object.**
(b) Conditional Instrument link Semantics: **Process** executes only if **Object** is at **State**, otherwise it is skipped and control moves to the next process.	 **Process** occurs if **Object** is **state.**

Figure 52. An OPM model of an instrument link (a) and a conditional instrument link (b)

Like a **Structural Link**, a **Procedural Link** can be connected by "xor" relations to other **Procedural Links** of the same type, as shown by the self tagged structural links labeled "**is xor-connected to**" and "**is or-connected to**" in **SD2.4**.

11.4.5 Informal Event Link Definitions

An event is a significant happening in the system that takes place during a particular moment and triggers some process in the system. An event is represented in OPM by an *event link*,

which is a procedural link that connects a source entity with a destination process. The semantics of this type of links is that the source entity attempts to trigger the destination process. The process does not start unless the event link is enabled, i.e., the event occurs and allthe process pre-conditions, represented by the incoming procedural links, are satisfied.

There are five types of event links in OPM: agent, state change, general event, invocation, and timeout. These are defined and exemplified next.

Agent Link

An *agent* is an intelligent object, a human or an organization consisting of humans, that controls a process by supplying input. An *agent link* is an event link which connects an agent with the process it triggers. For example, the process **Ordering** in Figure 53 starts only when its agent, the physical and environmental (external) **User**, enables its occurrence. The OPD symbol of an agent link is —● from the agent to the triggered process. In the OPL paragraph, this link is represented by the reserve word "handles".

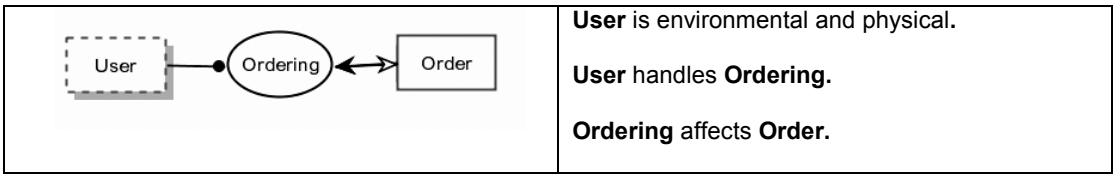


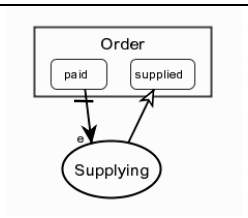
Figure 53. An OPM model of an agent link

State Change Event Link

An object state can cause two different types of events: a state entrance event and a state exit event. A *state entrance event* occurs when the object enters the particular state, while a *state exit event* occurs when the object exits that state. A *state change event* is a generalization of a state entrance event and a state exit event. A state change event link connects an object state with the process it triggers when entering or exiting the state. After the destination process occurs, the source entity of a state change event can either remain unchanged or be consumed or affected. A state entrance event link in which the source entity remain unchanged is symbolized by an enabling state entrance event link, +—©. The symbol +—►ᵉ, on the other

hand, is a consumption state entrance event link, which denotes the fact that the event source entity is consumed or affected. Similarly, a state exit event link can be an enabling link, symbolized by —⊣©, or a consumption link, symbolized by —⊣►ᵉ. If at some level of the OPM model there is no specification if the state change event link represents state entrance or state exit event, it will be simply symbolized by —© or —►ᵉ from the state to the triggered process.

In OPL, a triggering sentence is added to the OPL sentence representing the procedural link. Figure 54 specifies an OPM model of a **Supplying** process, which is triggered when the **Order** object enters its **paid** state. Two OPL sentences describe this link: a triggering sentence, which expresses the event aspect of the link, and a change sentence, which represents the procedural nature of this link.



Order can be **paid** or **supplied.**

Order triggers **Supplying** when it enters **paid.**

Supplying changes **Order** from **paid** to **supplied.**

Figure 54. An OPM model of a state entrance event link

For a state exit event link, the second sentence in Figure 54 would be "**Order** triggers **Supplying** when it exits **paid,**" while for a state change event link that sentence would be "**Order** triggers **Supplying** when it is **paid.**"

General Event Link

A general event can be an external stimulus, a change in an object state or value, etc. The source of a general event link is a thing (object or process). Figure 55, for example, specifying that **Reporting** is triggered any time **Order** changes its state. This single link could be replaced by two state entrance event links from each one of the states of **Order**, but the notation in Figure 55 is graphically more compact and conceptually more intuitive. **Order** itself, which triggers the **Reporting** process, does not change during **Reporting**, as denoted by

the circle in the symbol —©. A general event link can also be of type consumption, symbolized by —►ℓ, or effect, symbolized by ◄—►ℓ, denoting respectively that the source thing (object or process) is consumed or affected by the triggered process.

If the source entity of this link is an object with states, then the corresponding OPL sentence is "**Order** triggers **Reporting** when its state changes." Otherwise, the OPL sentence is simply "**Order** triggers **Reporting**."

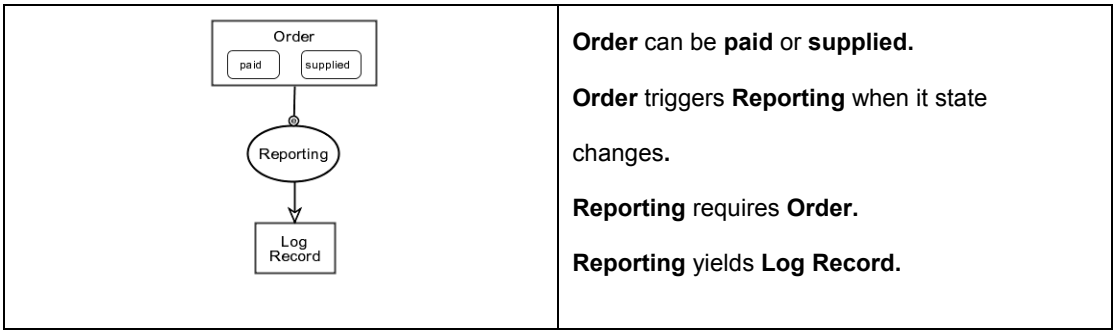
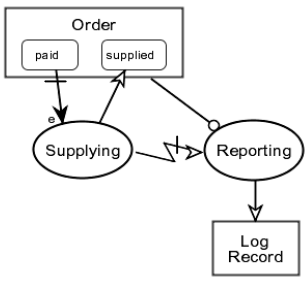


Figure 55. An OPM model of a general event link

Invocation Link

An *invocation link* is an event link between an invoking process and an invoked one. As noted, the vertical axis in an OPD denotes the time line within an in-zoomed process. The invocation link enables overriding this timeline default which is needed for example in cases such as loops or when the involved processes do not appear in the same context. An invocation link can trigger the invoked process when the invoking process starts, represented by +Z▷, when it ends, denoted by Z+▷, when it starts or ends, represented by +Z+▷, or at any time during its execution, represented by Z▷. Figure 56 specifies that **Reporting**, which uses **Order** as an instrument to create a **Log Record**, is triggered any time **Supplying** terminates.



Order can be **paid** or **supplied.**

Order triggers **Supplying** when it enters **paid.**

Supplying changes **Order** from **paid** to **supplied.**

Supplying triggers **Reporting** when it ends.

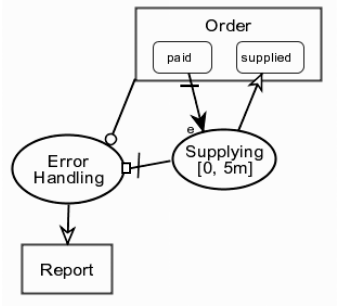
Reporting requires **Order.**

Reporting yields **Log Record.**

Figure 56. An OPM model of an invocation link

Timeout Event Link

A *timeout event link* connects a timed element, which can be a process, a state, or an event link, with a process which is triggered when the element violates its time constraints. When the element violates its minimal time constraint, the minimal timeout event link, denoted by ⊣□, is followed. When the element violates its maximal time constraint, the maximal timeout event link, denoted by —⊣□, is followed. The ⊣⊣□ symbol represents a timeout event link which is followed whenever the minimal or maximal time constraints are violated, while —□ represents an unspecified timeout violation event. In all cases, the box head of the link points towards the triggered process.



Order can be **paid** or **supplied.**

Order triggers **Supplying** when it enters **paid.**

Supplying lasts between **0** to **5** minutes.

Supplying changes **Order** from **paid** to **supplied.**

Supplying triggers **Error Handling** when it lasts more than **5** minutes.

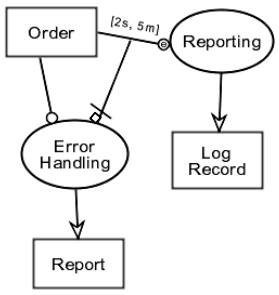
Error Handling requires **Order.**

Error Handling yields **Report.**

Figure 57. An OPM model of a timeout event link

The **Supplying** process in Figure 57, for example, is specified to last between 0 to 5 minutes. If it lasts more than 5 minutes, it triggers the **Error Handling** process, reporting the existing error.

The minimal and maximal time constraints of an event link define the minimal and maximal reaction timeout constraints: if the triggered process does not start within the interval [minimal time constraint, maximal time constraint] after a stimulus occurred, a timeout event occurs. In Figure 58, for example, **Reporting** should be triggered within 2 seconds to 5 minutes after a change in the **Order** state. If **Reporting** is not triggered within 5 minutes from the **Order** state change, **Error Handling** is triggered, creating a **Report** (using the **Order** information).



Order triggers **Reporting** when its state changes**,** with a reaction time of **2** seconds to **5** minutes**.** This link triggers **Error Handling** when its reaction time lasts more than **5** minutes**.**

Reporting requires **Order.**

Reporting yields **Log Record.**

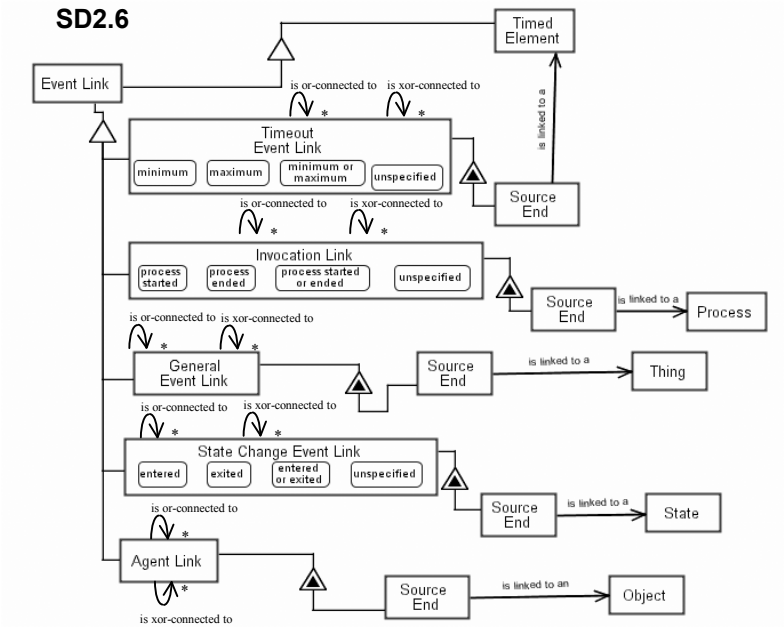
Error Handling requires **Order.**

Error Handling yields **Report.**

Figure 58. An OPM model of a reaction timeout event

11.4.6 Event Link Metamodel

As noted, an **Event Link**, which is unfolded in **SD2.6** (Figure 59), is a **Timed Element**, and as such, it inherits **Minimal Time Constraint**, **Maximal Time Constraint**, and **Duration Distribution Function** as its attributes. The **Duration Distribution Function** of an **Event** can be used for system simulation to define the distribution of the time that passes from the event occurrence to the start of the corresponding triggered process.



Event Link is a **Timed Element.**
Timeout Event Link is an **Event Link.**
Timeout Event Link can be **minimum**, **maximum**, **minimum or maximum**, or **unspecified.**
Source End of **Timeout Event Link is linked to a Timed Element.**
Timeout Event Link is xor-connected to optional **Timeout Event Links.**
Timeout Event Link is or-connected to optional **Timeout Event Links.**
Invocation Link is an **Event Link.**
Invocation Link can be **process started**, **process ended**, **process started or ended**, or **unspecified.**
Source End of **Invocation Link is linked to a Process.**
Invocation Link is xor-connected to optional **Invocation Links.**
Invocation Link is or-connected to optional **Invocation Links.**
General Event Link is an **Event Link.**
Source End of **General Event Link is linked to a Thing.**
General Event Link is xor-connected to optional **General Event Links.**
General Event Link is or-connected to optional **General Event Links.**
State Change Event Link is an **Event Link.**
State Change Event Link can be **entered**, **exited**, **entered or exited**, or **unspecified.**
Source End of **State Change Event Link is linked to a State.**
State Change Event Link is xor-connected to optional **State Event Links.**
State Change Event Link is or-connected to optional **State Event Links.**
Agent Link is an **Event Link.**
Source End of **Agent Link is linked to an Object.**
Agent Link is xor-connected to optional **Agent Links.**
Agent Link is or-connected to optional **Agent Links.**

Figure 59. **SD2.6**, in which **Event Link** of **OPM Ontology** is unfolded

SD2.5.1 also specifies the five types of **Event Links**:

1. **Agent Link**.
2. **State Change Event Link**, which can be **entered State Change Event Link**, **exited State Change Event Link**, **entered or exited State Change Event Link**, or **unspecified State Change Event Link**;

3. **General Event Link**;
4. **Invocation Link**, which can be **process started Invocation Link**, **process ended Invocation Link**, **process started or ended Invocation Link**, or **unspecified Invocation Link**; and
5. **Timeout Event Link**, which can be **minimum Timeout Event Link**, **maximum Timeout Event Link**, **minimum or maximum Timeout Event Link**, or **unspecified Timeout Event Link**.

An **Event Link** can be any **Procedural Link**, except for the **Result Link**, since the **Entity** of a **Result Link** is created during the **Process** and, hence, it cannot be the source that triggers the process. An **Event Link** does not represent a **Condition** either, but an attempt to trigger a process which succeeds if and only if the conditions, represented by the links that are coming into the process, hold. These constraints are also defined in OCL by constraint (2) in Appendix E.

12. Metamodel of OPM Behavior

12.1 Complexity Management

Complexity management aims at balancing the tradeoff between two conflicting requirements: completeness and clarity. Completeness requires that the system details be stipulated to the fullest extent possible, while the need for clarity imposes an upper limit on the level of complexity and does not allow for an OPD that is too cluttered or overloaded with entities and links among them. The seamless, recursive, and selective OPM scaling, i.e., refinement-abstraction, enables presenting the system at various detail levels without losing the "big picture" and the comprehension of the system as a whole.

12.1.1 Informal Refinement and Abstraction Mechanism Definitions

OPM features three built-in refinement-abstraction mechanisms, which are in-zooming and out-zooming, unfolding and folding, and state-expressing and state-suppressing.

In-Zooming and Out-Zooming

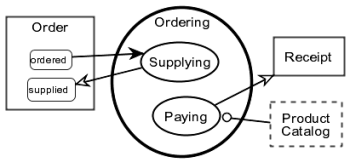
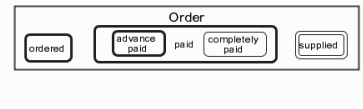
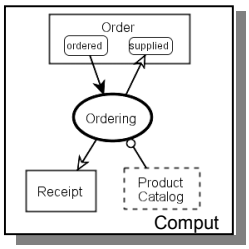
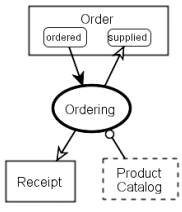
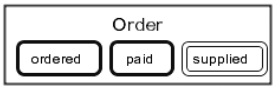
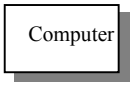
In-zooming and out-zooming are a pair of refinement and abstraction mechanisms, respectively, which can be applied to all the three entity types: objects, processes, and states. *In-Zooming*, i.e., zooming into an entity, decreases the distance of viewing it, such that lower-level elements enclosed within the entity become visible. Conversely, *out-zooming*, i.e., zooming out of a refined entity increases the distance of viewing it, such that all the lower-level elements that are enclosed within it become invisible.

Table 11 demonstrates zooming into a process, a state, and an object in both OPD and OPL. In Table 11(a), the **Ordering** process of Table 11(b) is in-zoomed to expose its two sub-processes: **Supplying** and **Paying**. This in-zoomed view also specifies that the **Order** state is changed during the first phase of **Ordering**, called **Supplying,** while **Receipt** is created during the second phase of **Ordering**, which is **Paying**. In Table 11(c), the state **paid** of **Order** in Table

11(d) is in-zoomed to expose its two sub-states: **advance paid** and **completely paid**. Finally, the in-zoomed OPD in Table 11(e) exposes the resource and computational components of the physical **Computer** object shown in Table 11(f). These components include the objects **Order**, **Product Catalog**, and **Receipt**, and the process **Ordering**.

An OPL paragraph that is equivalent to an in-zoomed OPD starts with an in-zooming sentence, as demonstrated at the bottom of Table 11. These sentences are for the cases when the in-zooming operation creates a new diagram. If the in-zooming operation is carried out in the same diagram, the diagram names do not appear in the OPL sentences. For example, "**Ordering** zooms into **Supplying** and **Paying**."

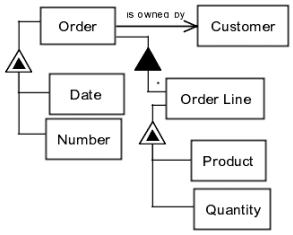
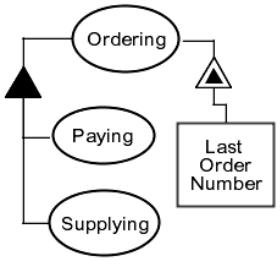
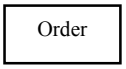
Table 11. Examples of the three entity types in their in-zoomed and out-zoomed versions.
(a) The process **Ordering** in-zoomed (b) **Ordering** out-zoomed (c) The state **paid** in-zoomed (d) **paid** out-zoomed (e) The object **Computer** in-zoomed (f) **Computer** out-zoomed.

	Process	**State**	**Object**
In-Zooming	(a) SD1 - Ordering in-zoomed 	(c) SD1 - paid in-zoomed 	(e) SD1 - Computer in-zoomed 
Out-Zooming	(b) SD 	(d) SD 	(f) SD 
OPL	**Ordering** from **SD** zooms in **SD1** into **Supplying** and **Paying.**	**Paid** from **SD** zooms in **SD1** into **advance paid** and **completely paid.** **Advance paid** is initial.	**Computer**from **SD** zooms in **SD1** into **Order**, **Receipt**, and **Product Catalog,** as well as **Ordering.**

Unfolding and Folding

Unfolding and folding are a pair of refinement and abstraction mechanisms, respectively, which can be applied to things – objects or processes. *Unfolding* reveals a set of lower-level entities that are hierarchically below a relatively higher-level thing. The hierarchy is with respect to one or more structural links. The result of unfolding is a graph the root of which is the thing being unfolded. Linked to the root are the things that are exposed as a result of the unfolding. Conversely, *folding* is applied to the tree from which the set of unfolded entities is removed, leaving just the root.

Table 12. Examples of the two thing types in their unfolded and folded versions. (a) The object **Order** unfolded. (b) **Order** folded. (c) The process **Ordering** unfolded. (d) **Ordering** folded.

	Object	**Process**
Unfolding	(a) SD1 – Order unfolded 	(c) SD1 – Ordering unfolded 
Foldin	(b) SD 	(d) SD 
OPL	**Order** from **SD** unfolds in **SD1** to consist of optional **Order Line**, to exhibit **Date** and **Number**, and to be owned by **Customer.**	**Ordering** from **SD** unfolds in **SD1** to consist of **Paying** and **Supplying** and to exhibit **Last Order Number.**

The object **Order** of Table 12(b) is unfolded in Table 12(a) three times. The first is *aggregation unfolding*, which exposes the parts of **Order**, one or more **Order Lines.** The second unfolding is *exhibition unfolding*, which exposes the features (attributes only in this case) of **Order**, **Date** and **Number**. The third unfolding of **Order** is *tagged unfolding*, which

lists the things of **Order** connected to it via a tagged structural link, **Customer**. **Order Line**, in turn, is unfolded to show its **Product** and **Quantity** attributes.

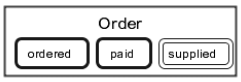
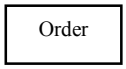
Processes can be unfolded just like objects. In Table 12(c), for example, the process **Ordering** from Table 12(d) is unfolded to expose its parts, **Paying** and **Supplying**, and its feature (in this case an attribute which is an internal variable), **Last Order Number**.

Table 12 also specifies the OPL unfolding sentences for the cases where the unfolding operation creates a new diagram.

State Expressing and State Suppressing

State expressing is a refinement mechanism applied to objects which reveals a set of states inside an object. *State Suppressing* is the abstraction mechanism which conceals a set of states inside an object. Table 13(a) expresses the three states of **Order**: **ordered**, **paid**, and **supplied**, while Table 13(b) suppresses them. The equivalent OPL sentence is "**Order** can be **ordered**, **paid**, and **supplied.**"

Table 13. Examples of state expressing and state suppressing. (a) The object **Order** is state expressed. (b) **Order** is state suppressed.

State Expressing	State Suppressing	OPL
(a) 	(b) 	**Order** can be**ordered** , **paid**, or **supplied.** **Ordered** is initial. **Paid** is initial.

12.1.2 Refinement and Abstraction Mechanism Metamodel

In **SD3** (Figure 60) **System** from **SD** is tag unfolded to expose the fact that it is specified by one or more **OPM Components**. Each **OPM Component** is a stand-alone model of a system that can be reused as a subsystem in another, more complex system. An **OPM Component is graphically represented by an OPD-set** or, equivalently, by an **OPL Script**. When a system is composed of more than one **OPM Component**, each component is denoted as being distinct

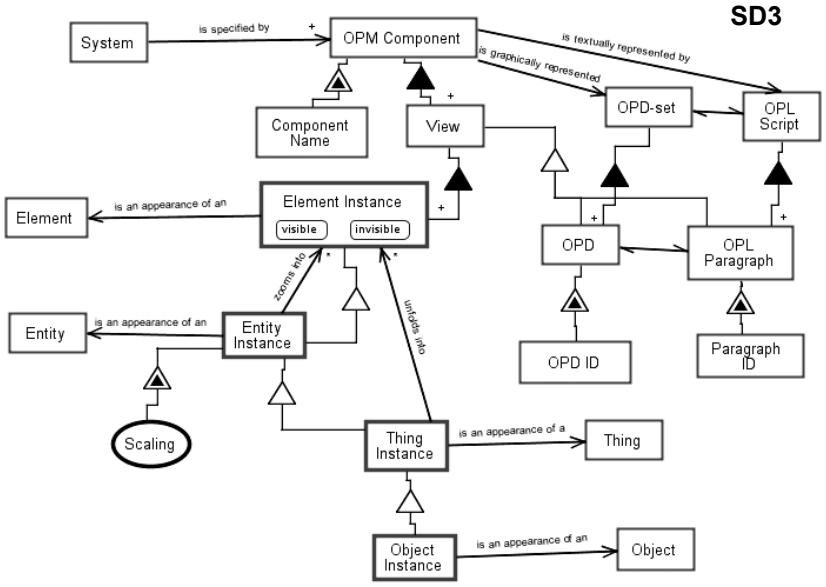
both graphically and textually. Graphically, each component is enclosed within the component symbol,  (which is the same as the UML package symbol). In the textual counterpart, the prefix "The component" or "the component" precedes the **Component Name**. For example, if a Web portal system includes an Auction component, the OPL sentence will read "**Web Portal** consists of the component **Auction.**" Like any complex object, a component can be in-zoomed, and this is expressed in a component in-zooming sentence, such as, "The component **Auction** of **SD** zooms in **SD1** into **Bidder**, **Start Price,** and **Caller**, as well as **Bidding** and **Winning.**"

Each **OPM Component** is composed of at least one **View**, which is an **OPD** (the graphic expression) or an **OPL Paragraph** (the textual expression), respectively. An **OPD** is characterized by an **OPD ID** (e.g., **SD**, **SD1.3**, etc.), while an **OPL Paragraph** has a corresponding **Paragraph ID**, which is the **OPD ID** to which the letter **P** (for **Paragraph**) is appended (e.g., **SDP**, **SD1.3P**, etc.).

A **View** in an **OPM Component** consists of **Element Instances**, each of which is an appearance of an **Element**. An **Element Instance** can be visible or invisible, which determines whether the **Element Instance** appears (is visible) in the **OPD** along with the equivalent **OPL Sentences** in which it participates. Like the **Element** hierarchy, an **Element Instance** specializes into an **Entity Instance**, which in turn specializes into **Thing Instance** that further specializes into **Object Instance**. An **Entity Instance**, a **Thing Instance**, and an **Object Instance** are appearances of an **Entity**, a **Thing**, and an **Object**, respectively.

An **Entity Instance** exhibits **Scaling**, which is an operation that changes the level of detail at which the system, or parts of it, is specified. **Scaling**, which is unfolded in **SD3.1** (Figure 61), has three attributes: **Purpose**, **Mode**, and **Diagram**. **Purpose**, which can be **elaboration** or **simplification**, indicates whether the specification is refined or abstracted, respectively. **Refining** is **elaboration Scaling**, i.e., **Scaling** the **Purpose** of which is **elaboration**. **Refining**

means exposing more details of the system by showing more entities and how they interconnect. **Abstracting**, the inverse of **Refining**, which means hiding details, is **simplification Scaling**, i.e., **Scaling** the **Purpose** of which is **simplification**.

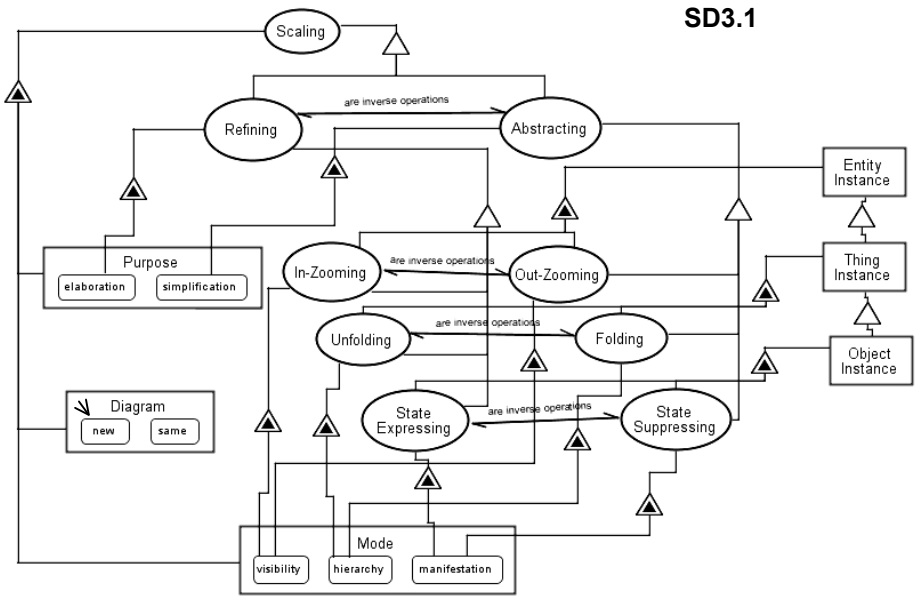


OPD-set consists of at least one **OPD.**
OPD is a **View.**
OPD exhibits **OPD ID.**
OPL Script consists of at least one **OPL Paragraph.**
OPL Paragraph is a **View.**
OPL Paragraph exhibits **Paragraph ID.**
OPL Paragraph and **OPD** are equivalent**.**
OPD-Set and **OPL Paragraph** are equivalent**.**
System is specifiedby at least one **OPM Component.**
OPM Component exhibits **Component Name.**
OPM Component consists of at least one **View.**
View consists of at least one **Element Instance.**
Element Instance can be **visible** or **invisible.**
Element Instance is an appearance of an Element.
Entity Instance is an **Element Instance.**
Entity Instance exhibits **Scaling.**
Entity Instance is an appearance of an Entity.
Entity Instance zooms into optional **Element Instances**.
Thing Instance is an **Entity Instance.**
Thing Instance is an appearance of a Thing.
Thing Instance unfolds into optional **Element Instances**.
Object Instance is an **Thing Instance.**
Object Instance is an appearance of an Object.
OPM Component is graphically represented by an OPD-set.
OPM Component is textually represented by an OPL Script.

Figure 60. **SD3**, in which **System** is unfolded

The **Scaling Mode** attribute, which can be **visibility**, **hierarchy**, or **manifestation**, expresses the graphic way by which the **Scaling** is done. **Visibility Mode Scaling** means showing or

hiding the inner content of an **Entity** (**Object, Process**, or **State**). **Hierarchy Mode Scaling** means showing or hiding the hierarchical tree structure of a **Thing**, which becomes the root of that tree. **Manifestation Mode Scaling** expresses (shows) or suppresses (hides) the **State** contents of an **Object**.



Scaling exhibits **Purpose**, **Mode**, and **Diagram.**
Purpose can be **elaboration** or **simplification.**
Model can be **visibility**, **hierarchy**, or **manifestation.**
Diagram can be **new**, which is the default, or **same**
Refining is **Scaling**, the **Purpose** of which is **elaboration.**
Abstracting is **Scaling**, the **Purpose** of which is **simplification.**
Abstracting and **Refining are inverse operations.**
Entity Instance exhibits **In-Zooming** and **Out-Zooming.**
In-Zooming is **Refining**, the **Mode** of which is **visibility.**
Out-Zooming is **Abstracting**, the **Mode** of which is **visibility.**
In-Zooming and **Out-Zooming are inverse operations.**
Thing Instance is an **Entity Instance.**
Thing Instance exhibits **Unfolding** and **Folding.**
Unfolding is **Refining**, the **Mode** of which is **hierarchy.**
Folding is **Abstracting**, the **Mode** of which is **hierarchy.**
Unfolding and **Folding are inverse operations.**
Object Instance is a **Thing Instance.**
Object Instance exhibits **State Expressing** and **State Suppressing.**
State Expressing is **Refining**, the **Mode** of which is **manifestation.**
State Suppressing is **Abstracting**, the **Mode** of which is **manifestation.**
State Expressing and **State Suppressing are inverse operations.**

Figure 61. **SD3.1**, in which **Scaling** is unfolded

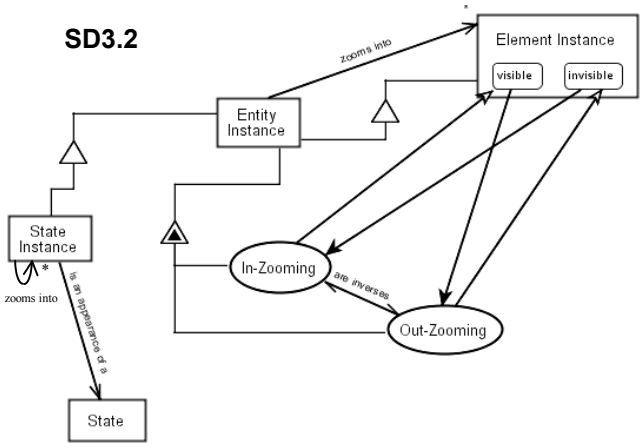
Refining whose **Mode** is **visibility** is called **In-Zooming,** while **Abstracting** whose **Mode** is **visibility** is called **Out-Zooming**. **Hierarchy Mode Scaling** is called **Unfolding** when its **Purpose**

is **elaboration** and **Folding** when its **Purpose** is **simplification**. Similarly, **Manifestation Mode Scaling** is called **State Expressing** when its **Purpose** is **elaboration** and **State Suppressing** when its **Purpose** is **simplification**.

As noted, **In-zooming** and **Out-zooming** are operations of an **Entity** (**Process**, **State**, and **Object**), while **Unfolding** and **Folding** are operations of a **Thing** (**Object** and **Process**). **State Expressing** and **State Suppressing** can be applied only to **Objects**.

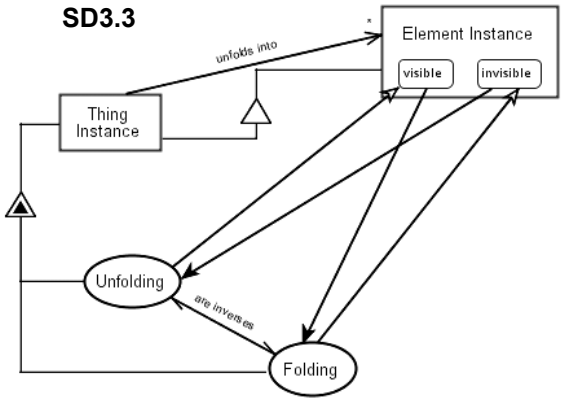
The third attribute of **Scaling** is **Diagram**. It relates to the target OPD in which the **Scaling** (**Refining** or **Abstracting**) operation is done. The **Diagram** attribute has two values: **new** (which is the default) and **same**. **New Diagram Scaling** generates a new OPD, in which the entity of interest is scaled (refined or abstracted), while **same Diagram Scaling** uses the exiting OPD to scale the entity of interest. A newly created diagram is automatically given a name, which consists of an identifier and a description. The identifier of the top-level (level zero) OPD is SD, for System Diagram. The identifier of any OPD generated by refining an entity in SD is SDi, where i is 1,2, etc. The identifier of any OPD generated by refining an entity in SDi is SDi.j, where j is 1,2, and so on.

SD3.2 (Figure 62) unfolds **Entity Instance**, showing its scaling mechanisms, **In-Zooming** and **Out-Zooming**. As noted, **In-Zooming** is a refining mechanism which exposes the inner details of an **Entity Instance** within its frame, while **Out-Zooming** is its inverse abstracting mechanism. **In-Zooming** into an **Entity** makes the in-zoomed **Element Instances** inside the **Entity** visible. Conversely, **Out-Zooming** of an **Entity** makes the visible (in-zoomed) **Element Instances** inside the **Entity** invisible. This functionality is also defined by the OCL constraint (3) in Appendix E. Moreover, this constraint and **SD3.2** also specify that the in-zoomed **Element Instances** of a **State Instance** can be only **State Instances**. In other words, only states can be nested within a state.



Element Instance can be **visible** or **invisible.**
Entity Instance is an **Element Instance.**
Entity Instance exhibits **In-Zooming** and **Out-Zooming.**
In-Zooming changes **Element Instance** from **invisible** to **visible.**
Out-Zooming changes **Element Instance** from **visible** to **invisible.**
In-Zooming and **Out-Zooming are inverses.**
Entity Instance zooms into optional **Element Instances.**
State Instance is an **Entity Instance.**
State Instance is an appearance of a State.
State Instance zooms into optional **State Instances.**

Figure 62. **SD3.2**, in which **Entity Instance** is unfolded



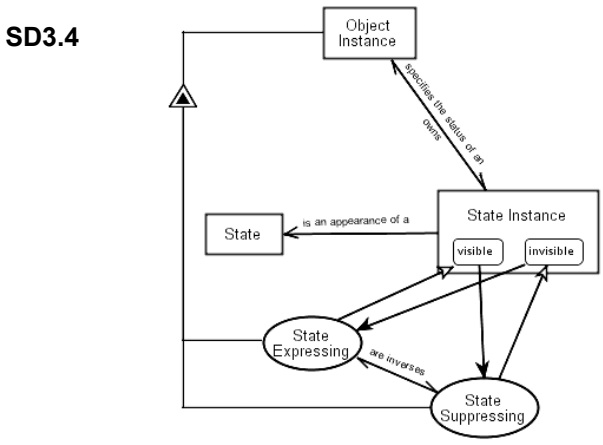
Element Instance can be **visible** or **invisible.**
Thing Instance is an **Element Instance.**
Thing Instance exhibits **Unfolding** and **Folding.**
Unfolding changes **Element Instance** from **invisible** to **visible.**
Folding changes **Visibility** of **Element Instance** from **visible** to **invisible.**
Unfolding and **Folding are inverses.**
Thing Instance unfolds into optional **Element Instances.**

Figure 63. **SD3.3**, in which **Thing Instance** is unfolded

SD3.3 (Figure 63) unfolds **Thing Instance** which exhibits an additional scaling mechanism, **Unfolding** and **Folding**. In **Unfolding**, structural relations are used for refining and detailing the structural parts of a **Thing**, while **Folding** is the inverse abstracting mechanism of **Unfolding**.

The functionality of the folding and unfolding mechanisms is textually stated by the OCL constraint (4) in Appendix E.

SD3.4 (Figure 64) unfolds **Object Instance**, showing its **State Instances**. These **State Instances** can be expressed or suppressed by the **State Expressing** and **State Suppressing** operations of an **Object Instance**. **State Expressing** changes the **State Instances** of the current object to **visible** (i.e., the object states are shown or expressed). Conversely, **State Suppressing** changes the **State Instances** of the current object to **invisibile** (i.e., the object state instances are hidden or suppressed). The OCL constraint (5) in Appendix E enforces the change (from **visible** to **invisible** or vice versa) on all the **State Instances** of the current **Object Instance**.



State Instance can be **visible** or **invisible.**
State Instance is an appearance of a State.
State Instance specifies the status of an Object Instance.
Object Instance exhibits **State Expressing** and **State Suppressing.**
State Expressing changes **State Instance** from **invisible** to **visible.**
State Suppressing changes **State Instance** from **visible** to **invisible.**
State Expressing and **State Suppressing are inverses.**
Object Instance is a appearance of an Object.
Object Instance owns optional **State Instances.**

Figure 64. **SD3.4**, in which **Object Instance** is unfolded

Sometimes, especially when the number of states is large, it is desirable to show only a subset of the states in an object in a particular OPD, especially those that are input to or output of processes shown in that OPD. To this end, OPM also allows selective **State**

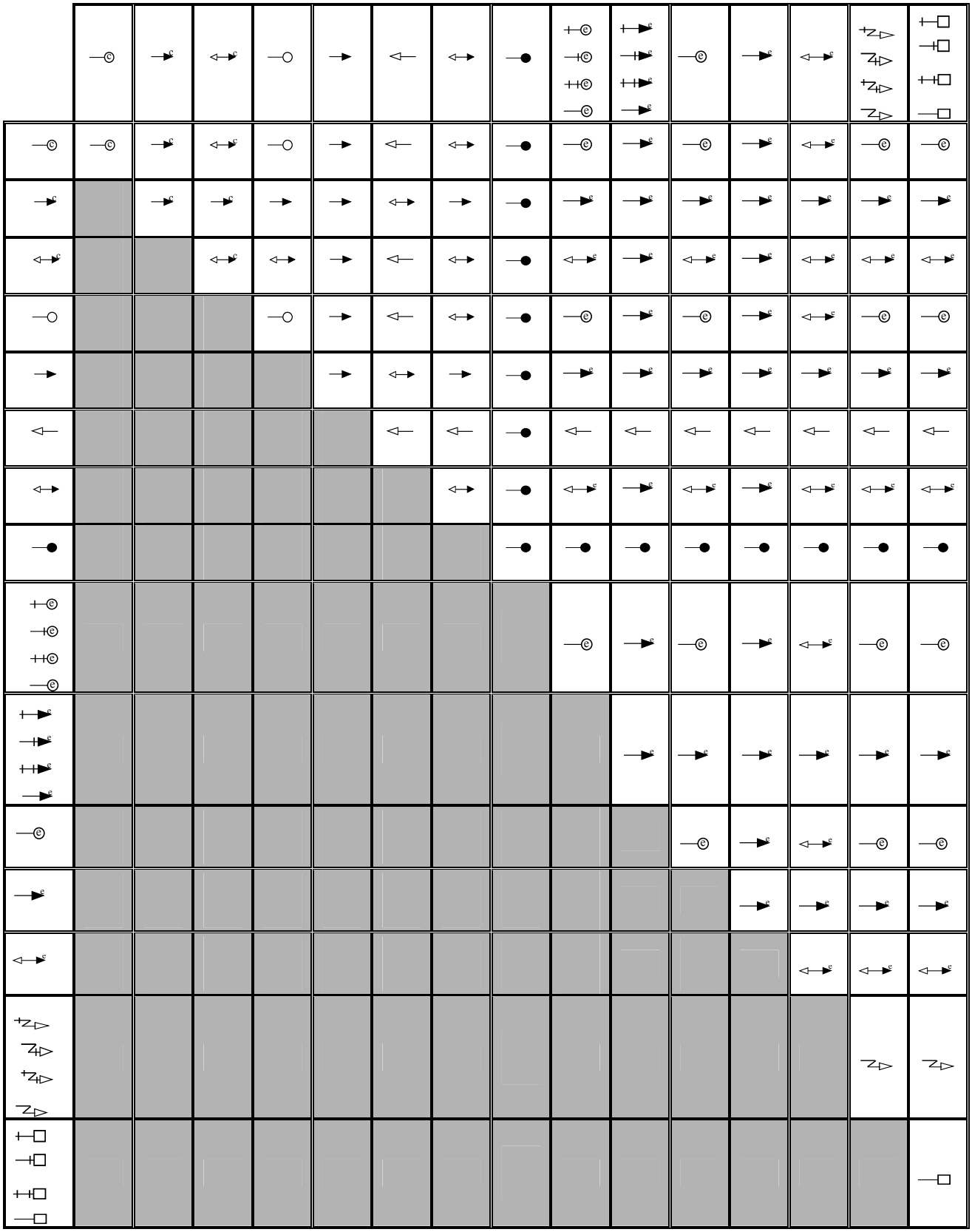
Expressing and selective **State Suppressing**, in which only a selected subset of states is expressed or suppressed, respectively.

12.1.3 Informal Consistency Rule Definitions

While abstracting (folding, out-zooming, or state-suppressing) an **Entity**, all the procedural links connecting an **Entity** outside the abstracted **Entity** with an **Entity** inside the abstracted **Entity** migrate to the circumference of the abstracted **Entity**, because all the internal entities disappear. However, OPM mandates that two entities are not linked by more than one procedural link. Hence, if more than one link type results in from abstracting, only one of them must be selected as the representative of all the resulting links. To maintain consistence, the selected link is the one that is most abstract of all the possible link candidates.

Table 14 determines the abstraction order of procedural links by defining for each pair of "competing" procedural links a third procedural link (which may be one of the two) that has to be selected while abstracting an **Entity**. In Figure 65(a), for example, **Order** is implicitly connected to **Ordering** through **Date** with an enabling instrument link and through **Quantity** of **Order Line** via an effect link. This implies that **Ordering** changes **Quantity** of an **Order Line** and uses **Date** of **Order** without changing it. Since effect can be viewed as a combination of use and change, an effect link is more abstract than an instrument link, as denoted in Table 14. Hence, when folding **Order**, **Order** and **Ordering** are linked by an effect link. In Figure 65(b), a result link connects **Creating** of **Ordering** to **Order**, an effect link links **Order** to **Updating** of **Ordering**, and an instrument link connects **Order** to **Printing** of **Ordering**. Since creation can be viewed as an effect in which the existence of the created thing is changed from non-exist to exist and consumption can be viewed as an effect in which the existence of the consumed thing is changed from exist to non-exist, an effect link is more abstract than both result and consumption links.

Table 14. Abstraction order of procedural links



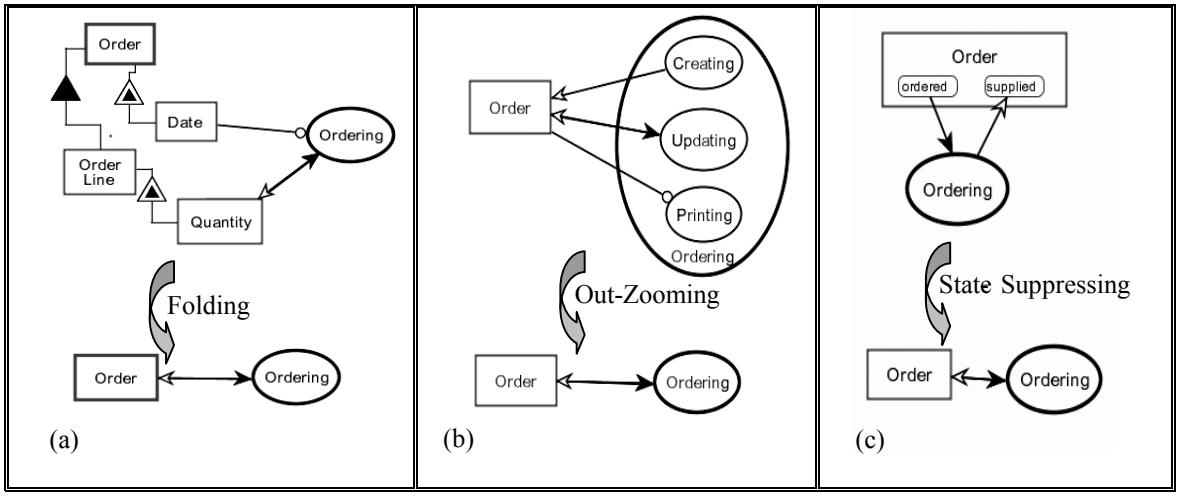


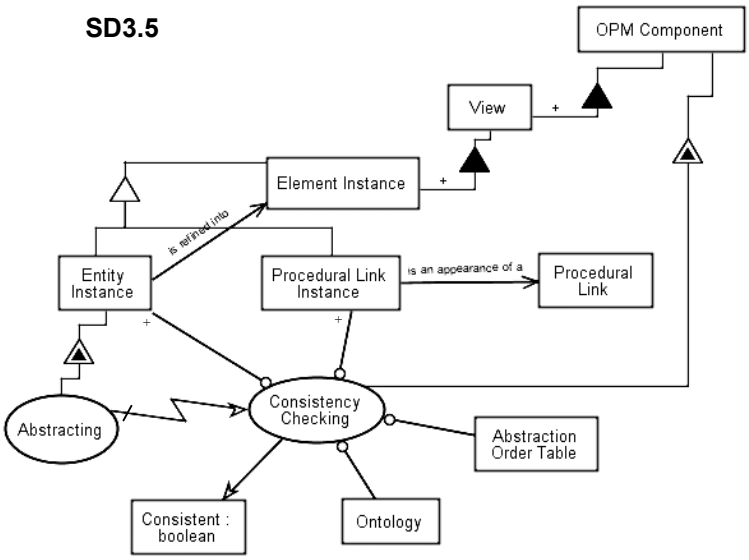
Figure 65. Example for scaling consistency rules. (a) Folding of the object **Order**. (b) Out-zooming of the process **Ordering**. (c) State suppressing of the object **Order**.

Hence, when **Ordering** is out-zoomed, **Ordering** and **Order** are linked via an effect link. In Figure 65(c), **Order** is connected to **Ordering** through its **ordered** state with a consumption (input) link and through its **supplied** state via a result (output) link. When state-suppressing **Order**, **Order** and **Ordering** are linked by an effect link, since an effect link is the abstraction of result and consumption links, which means that "**Ordering** changes **Order** from **ordered** to **supplied.**"

12.1.4 Metamodel of the Abstraction Procedural Link Consistency Rule

The abstraction procedural link consistency rule is stated as follows:

While abstracting an entity E that is linked with more than one procedural link to another entity E', the selected procedural link between E and E' is the most abstract link among the links that had existed between E' and the refineables of E. If this link cannot connect the relevant entities according to the metamodel constraints, no link will be shown at the more abstract level.



Consistent is of type Boolean**.**
OPM Component exhibits **Consistency Checking.**
Consistency Checking requires optional **Entity Instances**, optional **Procedural Link Instances**, **Abstraction Order Table**, and **Ontology.**
Consistency Checking yields **Consistent.**
OPM Component consists of at least one **View.**
View consists of at least one **Element Entity.**
Entity Instance is an **Element Instance.**
Entity Instance exhibits **Abstracting.**
Abstracting triggers **Consistency Checking** when it ends**.**
Entity Instance is refined into optional **Element Instances.**
Procedural Link Instance is an **Element instance.**
Procedural Link Instance is an appearance of a Procedural Link.

Figure 66. **SD3.5**, in which the consistency rule is specified

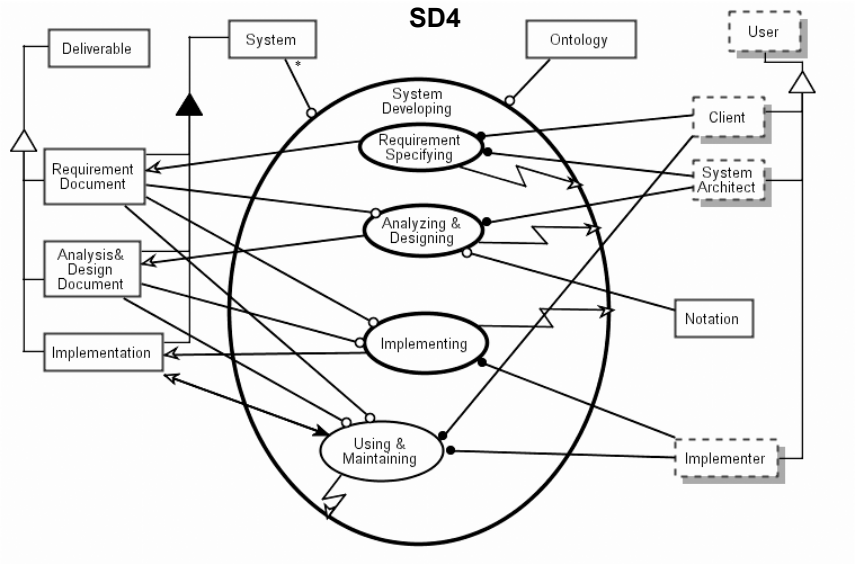
SD3.5 (Figure 66) defines the inputs and outputs required for the **Consistency Checking** process, which is triggered whenever an **Abstracting** operation finishes executing. **Consistency Checking** requires the **Abstraction Order Table** (defined in Table 14), **OPM Ontology**, **Entity Instances**, and **Procedural Link Instances**. It defines whether the **OPM Component** is **Consistent** or not. **Consistency Checking** functionality, defined by the OCL constraints (6) and (7) in Appendix E, verifies that the **Procedural Link Instance** that directly connects two **Entity Instances** is actually the possible most abstract link that can connect between the two **Entities** according to the OPM **Ontology** and the **Abstraction Order Table**.

12.2 Metamodel of an OPM-based Development Process

As noted, a system development process is part of any methodology and, hence, should be part of its metamodel. The system development model, presented in this section, follows generic concepts of systems evolution and lifecycle, namely requirement specification, analysis and design, implementation, usage and maintenance. As such, it is not specific to OPM-based system development, neither to Web application development. The elaborate backtracking options of this model, which are built-in at all levels, make it flexible enough to represent a variety of information system development approaches, ranging form the classical waterfall model through incremental development to prototyping.

12.2.1 Main Development Stages

Zooming into **System Developing** of **SD** (Figure 34), **SD4** (Figure 67) shows the common sequential stages of system developing processes: **Requirement Specifying**, **Analyzing & Designing**, **Implementing**, and **Using & Maintaining**. All of these processes use the same OPM **Ontology**, a fact that helps narrow the gaps between the different stages of the development process. Figure 67 shows that the **Client** and the **System Architect**, who, along with the **Implementer**, specialize **User**, handle the **Requirement Specifying** sub-process. **Requirement Specifying** takes OPM **Ontology** as input and creates a new **System**, which, at this point, consists only of a **Requirement Document**. The termination of **Requirement Specifying** starts **Analyzing & Designing**, the next sub-process of **System Developing**.



User is environmental and physical**.**

Client, which is environmental and physical, is a **User.**

Client handles **Requirement Specifying** and **Using & Maintaining.**

System Architect, which is environmental and physical, is a **User.**

System Architect handles **Requirement Specifying** and **Analyzing & Designing.**

Implementer, which is environmental and physical, is a **User.**

Implementer handles **Implementing** and **Using & Maintaining.**

System consists of **Requirement Document**, **Analysis & Design Document**, and **Implementation.**

Requirement Document is a **Deliverable.**

Analysis & Design Document is a **Deliverable.**

Implementation is a **Deliverable.**

System Developing zooms into **Requirement Specifying**, **Analyzing & Designing**, I**mplementing**, and **Using & Maintaining.**

System Developing requires **Ontology** and optional **Systems.**

Requirement Specifying yields **Requirement Document.**

Requirement Specifying invokes **System Developing.**

Analyzing & Designing requires **Notation** and **Requirement Document.**

Analyzing & Designing yields **Analysis & Design Document.**

Analyzing & Designing invokes **System Developing.**

Implementing requires **Requirement Document** and **Analysis & Design Document.**

Implementing yields **Implementation.**

Implementing invokes **System Developing.**

Using & Maintaining requires **Requirement Document** and **Analysis & Design Document.**

Using & Maintaining affects **Implementation.**

Using & Maintaining invokes **System Developing**.

Figure 67. **SD4**, in which **System Developing** is in-zoomed

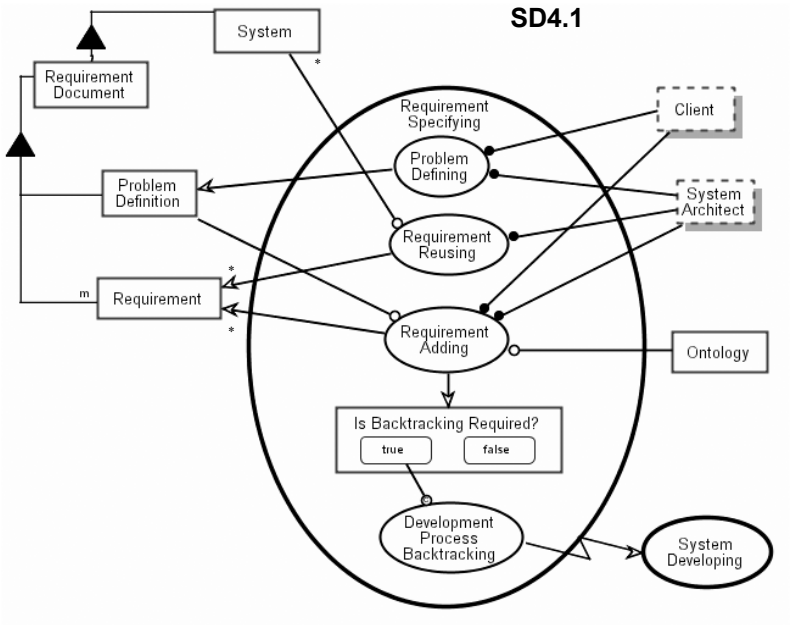
The agent of the **Analyzing & Designing** stage is the **System Architect**, who uses the **Requirement Document** and OPM **Notation** to create a new part of the system, the **Analysis &**

Design Document. When the **Analyzing & Designing** process terminates, the **Implementer** (programmer, DBA, etc.) starts the **Implementing** phase, which uses the **Requirement Document** and the **Analysis & Design Document** in order to create the **Implementation**. Finally, the **Implementer** changes the system **Implementation** during the **Using & Maintaining** stage, while the **Client** uses the **System**.

Each **System Developing** sub-process can invoke restarting of the entire development process, which potentially enables the introduction of changes to the requirements, analysis, design, and implementation of the **System**. These invocations give rise to an iterative development process, in which an attempt to carry out a sub-process reveals faults in the deliverable of a previous sub-process, mandating a corrective action.

12.2.2 The Requirement Specifying stage

In **SD4.1** (Figure 68), **Requirement Specifying** is in-zoomed, showing its four sub-processes. First, the **System Architect** and the **Client** define the problem to be solved by the system (or project). This **Problem Defining** step creates the **Problem Definition** part of the current system **Requirement Document**. Next, through the **Requirement Reusing** sub-process, the **System Architect** may reuse requirements that fit the problem at hand and are adapted from any existing **System** (developed by the organization). Reuse helps achieve high quality systems and reduce their development and debugging time. Hence, when developing large systems, such as some Web applications or real-time systems, it is important to try first to reuse existing artifacts adapted from previous generations, analogous systems, or commercial off-the-shelf (COTS) products that fits the current system development project. Existing, well-phrased requirements are often not trivial to obtain, so existing relevant requirements should be treated as a potential resource no less than code. Indeed, as **SD4.1** shows, reusable artifacts include not only components (which traditionally have been the primary target for reuse), but also requirements.



System consists of **Requirement Document.**

Requirement Document consists of **Problem Definition** and **Requirements.**

Client, which is environmental and physical, handles **Problem Defining** and **Requirement Adding.**

System Architect, which is environmental and physical, handles **Problem Defining**, **Requirement Reusing,** and **Requirement Adding.**

Requirement Specifying zooms into **Problem Defining**, **Requirement Reusing**, **Requirement Adding**, and **Development Process Backtracking**, as well as **Is Backtracking Required?.**

Is Backtracking Required? is of type Boolean**.**

Problem Defining yields **Problem Definition.**

Requirement Reusing requires optional **Systems.**

Requirement Reusing yields optional **Requirements.**

Requirement Adding requires **Problem Definition** and **Ontology.**

Requirement Adding yields **Is Backtracking Required?** and optional **Requirements.**

Development Process Backtracking occurs if **Is Backtracking Required?** is true**.**

Development Process Backtracking invokes **System Developing.**

Figure 68. **SD4.1**, in which **Requirement Specifying** is in-zoomed

After optional reuse of requirements from existing systems (or projects), the **System Architect** and the **Client**, working as a team, add new **Requirements** or update existing ones. This step uses OPM **Ontology** in order to make the **Requirement Document** amenable to be processed by other potential OPM tools, and in particular to an OPL compiler. Since the **System Architect** and the **Client** use OPM **Ontology** in defining the new requirements, the resulting **Requirement Document** is indeed expressed, at least partially, in OPL in addition to explanations in free natural English. Such structured OPM-oriented specification enables

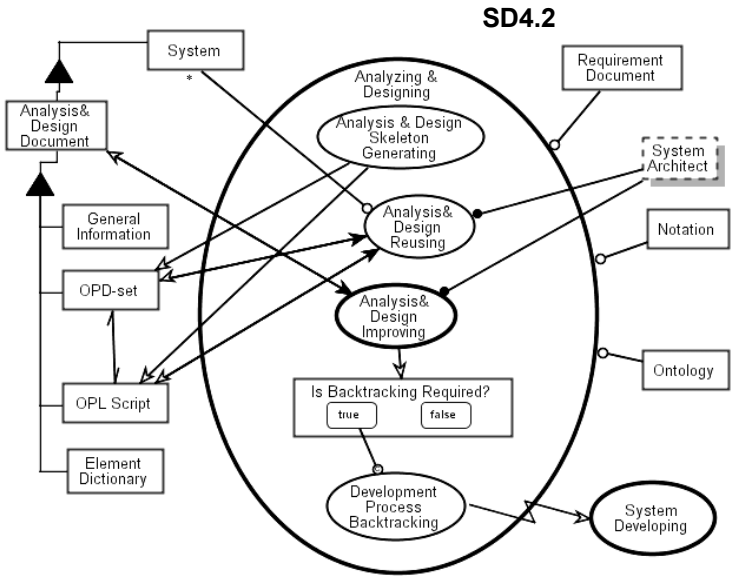
automatic translation of the **Requirement Document** to an OPM analysis and design skeleton (i.e., a skeleton of an OPD-set and its corresponding OPL script). Naturally, at this stage the use of free natural language beside OPM seems mandatory to document motivation, alternatives, considerations, etc.

Finally, the **Requirement Adding** process results in the Boolean object "**Is Backtracking Required?**", which determines whether **System Developing** should be restarted. If so, **Development Process Backtracking** invokes the entire **System Developing**. Otherwise, **Requirement Specifying** terminates, enabling the **Analyzing & Designing** process to begin.

12.2.3 The Analyzing and Designing stage

During the **Analyzing & Designing** stage, shown in **SD4.2** (Figure 69), a skeleton of an **OPL Script** is created from the **Requirement Document** for the current system. As noted, in order to make this stage as effective and as automatic as possible, the **Requirement Document** should be written using OPM, such that the resulting OPL script can be compiled. The **System Architect** can then optionally reuse analysis and design artifacts from previous systems (projects), creating a basis for the current system analysis and design. Finally, in an iterative process of **Analysis & Design Improving** (which is in-zoomed in **SD4.2.1**, Figure 70), the **System Architect** can engage in **OPL Updating**, **OPD Updating**, **System Animating, General Information Updating**, or **Analysis & Design Terminating**.

Any change a user makes to one of the modalities representing the model triggers an automatic response of the development environment software to reflect the change in the complementary modality. Thus, as **SD4.2.1** shows, **OPD Updating** (by the **System Architect**) affects the **OPD-set** and immediately invokes **OPL Generating**, which changes **OPL Script** according to the new **OPD-set**. Conversely, **OPL Updating** (also activated by the **System Architect**) affects the **OPL Script**, which invokes **OPD Generating**, reflecting the OPL changes in the **OPD-set**.



System Architect, which is environmental and physical, handles **Analysis & Design Reusing** and **Analysis & Design Improving.**

System consists of **Analysis & Design Document.**

Analysis& Design Document consists of **General Information**, **OPD-set**, **OPL Script**, and **Element Dictionary.**

OPD-set and **OPL Script** are equivalent.

Analyzing & Designing zooms into **Analysis & Design Skeleton Generating**, **Analysis & Design Reusing**, **Analysis & Design Improving**, and **Development Process Backtracking**, as well as **Is Backtracking Required?.**

Analyzing & Designing requires **Ontology**, **Notation**, and **Requirement Document.**

Is Backtracking Required? is of type Boolean.

Analysis & Design Skeleton Generating yields **OPD-set** and **OPL Script.**

Analysis & Design Reusing requires optional **Systems.**

Analysis & Design Reusing affects **OPD-set** and **OPL Script.**

Analysis & Design Improving affects **Analysis & Design Document.**

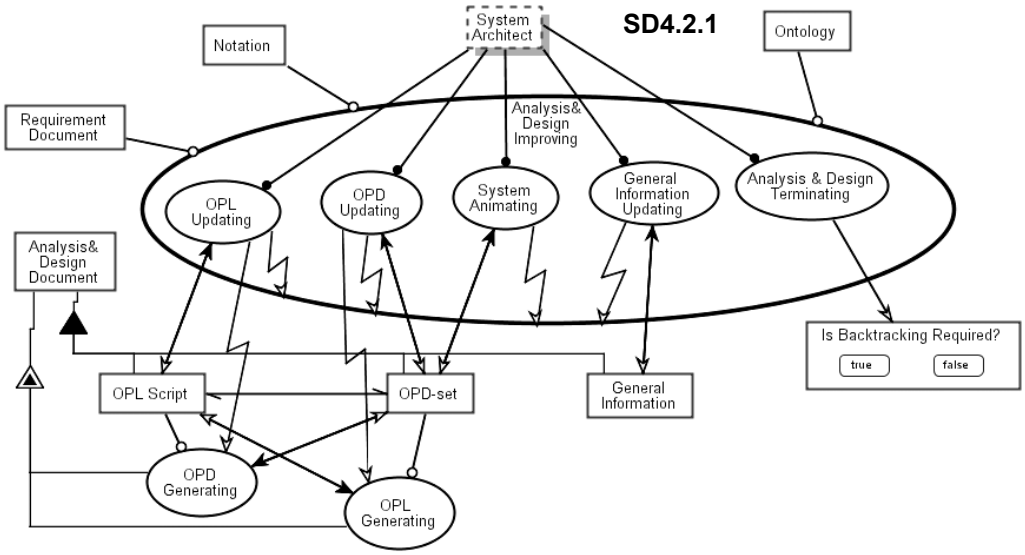
Analysis & Design Improving yields **Is Backtracking Required?.**

Development Process Backtracking occurs if **Is Backtracking Required?** is true.

Development Process Backtracking invokes **System Developing.**

Figure 69. **SD4.2**, in which **Analyzing & Designing** is in-zoomed

Since OPM enables modeling system dynamics and control structures, such as events, conditions, branching, and loops, **System Animating** simulates an **OPD-set**, enabling **System Architects** to dynamically examine the system at any stage of its development. Presenting live animated demonstrations of system behavior reduces the number of design errors percolated to the implementation phase. Both static and dynamic testing help in detecting discrepancies, inconsistencies, and deviations from the intended goal of the system.



System Architect, which is environmental and physical, handles **OPL Updating**, **OPD Updating**, **System Animating**, **General Information Updating,** and **Analysis & Design Terminating.**

Analysis & Design Document consists of **General Information**, **OPD-set**, and **OPL Script.**

OPD-set and **OPL Script** are equivalent.

Analysis & Design Document exhibits **OPD Generating** and **OPL Generating.**

OPD Generating requires **OPL Script.**

OPD Generating affects **OPD-set.**

OPL Generating requires **OPD-set.**

OPL Generating affects **OPL Script.**

Is Backtracking Required? is of type Boolean.

Analysis & Design Improving zooms into **OPL Updating**, **OPD Updating**, **System Animating**, **General Information Updating,** and **Analysis & Design Terminating.**

Analysis & Design Improving requires **Ontology**, **Notation,** and **Requirement Document.**

OPL Updating affects **OPL Script.**

OPL Updating invokes **OPD Generating** and **Analysis & Design Improving.**

OPD Updating affects **OPD-set.**

OPD Updating invokes **OPL Generating** and **Analysis & Design Improving.**

System Animating affects **OPD-set.**

System Animating invokes **Analysis & Design Improving.**

General Information Updating affects **General Information.**

General Information Updating invokes **Analysis & Design Improving.**

Analysis & Design Terminating yields **Is Backtracking Required?.**

Figure 70. **SD4.2.1**, in which **Analysis & Design Improving** is in-zoomed

As part of the dynamic testing, the simulation enables designers to track each of the system scenarios before writing a single line of code. Any detected mistake or omission is corrected at the model level, saving costly time and effort required if the error were only treated at the implementation level. Avoiding and eliminating design errors as early as possible in the

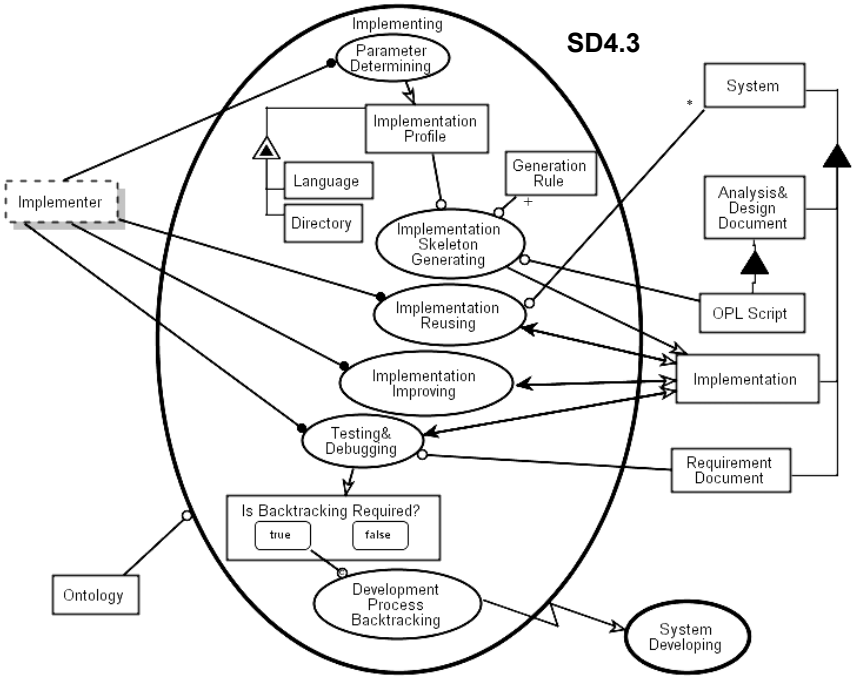
system development process and keeping the documentation up-to-date contribute also to shortening the system's delivery time ("time-to-market").

Upon termination of the **Analysis & Design Improving** stage, if needed, the entire **System Developing** process can restart or the **Implementing** stage begins.

12.2.4 The Implementing stage

The **Implementing** stage, in-zoomed in **SD4.3** (Figure 71), begins by defining the **Implementation Profile**, which includes the target **Language** (e.g., Java, C++, or SQL) and a default **Directory** for the artifacts. Then, the **Implementation Skeleton Generating** process uses the **OPL Script** of the current system and inner **Generation Rules** in order to create a skeleton of the **Implementation**. A **Generation Rule** saves pairs of OPL sentence types (templates) and their associated code templates in various target **Languages**. Chapter 14 details about the automatic OPM implementation generator.

The initial skeleton of the **Implementation**, which includes both the structural and behavioral aspects of the system, is then modified by the **Implementer** during the **Implementation Reusing** and **Implementation Improving** steps. In the **Testing & Debugging** stage, the resulting **Implementation** is checked against the **Requirement Document** in order to verify that it meets the system requirements defined jointly by the **Client** and the **System Architect**. If any discrepancy or error is detected, the **System Developing** process is restarted, else the system is finally delivered, assimilated and used. These sub-processes are embedded in the **Using & Maintaining** process at the bottom of **SD4** (Figure 67). While **Using & Maintaining** takes place, the **Client** collects new requirements that are eventually used when the next generation of the system is initiated. A built-in mechanism for recording new requirements in OPM format while using the system would greatly facilitate the evolution of the next system generation [23].



Implementer, which is environmental and physical, handles **Parameter Determining**, **Implementation Reusing**, **Implementation Improving**, and **Testing & Debugging.**

System consists of **Requirement Document** and **Analysis & Design Document.**

Analysis & Design Document consists of **OPL Script.**

Implementing requires **Ontology.**

Implementing zooms into **Parameter Determining**, **Implementation Skeleton Generating**, **Implementation Reusing**, **Implementation Improving**, **Testing & Debugging**, and **Development Process Backtracking**, as well as **Implementation Profile**, **Generation Rule**, and **Is Backtracking Required?.**

Implementation Profile exhibits **Language** and **Directory.**

Is Backtracking Required? Is of type Boolean**.**

Parameter Determining yields **Implementation Profile.**

Implementation Skeleton Generating requires **OPL Script**, **Generation Rules**, and **Implementation Profile.**

Implementation Skeleton Generating yields **Implementation.**

Implementation Reusing requires optional **Systems.**

Implementation Reusing affects **Implementation.**

Implementation Improving affects **Implementation.**

Testing & Debugging requires **Requirement Document.**

Testing & Debugging affects **Implementation.**

Testing & Debugging yields **Is Backtracking Required?.**

Development Process Backtracking occurs if **Is Backtracking Required?** is true**.**

Development Process Backtracking invokes **System Developing.**

Figure 71. **SD4.3**, in which **Implementing** is in-zoomed

Part 5. Summary and Implementation Issues

13. Summary and Contribution

Existing hypermedia authoring techniques and system development methods are not up to the task of complete and accurate modeling of all the complex structural and dynamic aspects of Web applications. Moreover, the aspects that are supported by these techniques are usually spread across various views, which make the comprehension and integrity of the system as a whole difficult. To meet the challenges posed by the Web, OPM/Web augments OPM to enable specification of Web applications in a single, coherent view without adding to OPM vocabulary new concepts, but rather enabling additional ways of using the existing elements. This increases the expressiveness of OPM, such that OPM models remain validalso in OPM/Web.

OPM/Web improves the accuracy and expressiveness of existing Web application modeling languages and methods in the following ways.

- OPM/Web combines the physical, static, behavioral, and functional views of a system within a single framework, enabling coherent, clear, and precise modeling of code mobility concepts, design paradigms, and dynamic architectures. Mental integration of the structure and behavior of Web applications in order to comprehend them in their entirety can be achieved with current methods only with great difficulty due to the multiplicity of views that need to be consulted. The segregation of a UML model, for example, into multiple views, which span across different diagram types, was found as a source of difficulty in capturing and understanding the system as a whole [78]. Indeed, comparing the complexity metric values of UML with other object-oriented techniques, Siao and Cao [91] found that each diagram in UML is not distinctly more complex than techniques in other object-oriented methods, but as a whole, UML is 2-11 times more complex than

other object-oriented methods. Moreover, the single view of OPM/Web enables specifying mutual effects between the various aspects of Web applications that cannot be done in the standard UML. For example, an elementary operation of transferring a computational component from one physical location to another and executing it there, which can be modeled by OPM/Web in a straightforward manner, is difficult to model with UML.

- OPM/Web applies a variety of scaling mechanisms for seamlessly and flexibly changing the level of detail of the system being designed. Complexity is inherent in real-life systems, so a set of tools for controlling and managing this complexity should be an integral part of a system development methodology. Complexity management entails a tradeoff between two conflicting requirements: completeness and clarity. OPM introduces three refinement-abstraction mechanisms, which are extended and formalized in this work.
- OPM/Web enables modeling stand-alone processes, which are at the heart of Web applications and are best expressed without the need to break them apart and distribute the parts as operations (or methods) of the governing objects, as the object-oriented paradigm and its UML design standard require. Modeling the system behavior in UML is spread across up to five different views: use case diagrams, in which the system functionality is defined; class diagrams, in which the operations are specified as being owned by objects; interaction diagrams, in which the messages that pass among the different objects is specified; Statecharts, which represent the change in object states over time; and activity diagrams, which specify the performance and flow of the system actions or sub-activities. Each of these diagrams has its own set of symbols, syntax, and semantics, and the human modeler must mentally traverse back and forth among these views. In spite of this model

multiplicity, none of the models capture the whole functionality of a process as a pattern along with its effects on different objects it involves. Indeed, as the experiment presented in this work has shown, OPM/Web is better than UML in capturing and understanding the behavior of Web applications.

- OPM/Web provides for a technology-independent method (known in the MDA nomenclature as Platform Independent Model, or PIM), in which process triggers, pre-conditions, and post-conditions are specified generically. Once the application is modeled, a solid and detailed skeleton of the technology-dependent implementation (known in MDA as Platform Specific Model, or PSM) can be automatically generated and simulated. This skeleton includes not only the structure of the application, but also its behavior, enabling design verification and leaving to the implementer only the coding at the bottom level. In contrast, UML requires that a set of stereotypes (denoted by different graphical symbols), tagged values, and constraints be defined for each domain of discourse. Such extension mechanisms undermine UML standardization efforts, since each researcher or company working in a particular domain can develop a different set of extensions. Lack of a universal set of such extension entities inhibits the efforts to develop reusable components. As noted, OPM/Web makes it a point not to enlarge the OPM vocabulary, but rather to enable new modes of using and combining existing elements. Indeed, a significant number of OPM/Web extensions have been incorporated into the core OPM, enabling system architects to use a single, relatively simple development methodology that is suitable for modeling systems in a variety of domains.
- OPM/Web advocates maintaining reusable components at a high level of abstraction and refining them in specific contexts. The OPM combination of object- and process-oriented paradigms enables modeling partially specified structures and

behaviors and adapting them to specific target components in a clean and clear way. The ability to weave objects and processes in the same OPM/Web model significantly improves upon existing Component-Based Development (CBD) approaches and methods, which primarily refer to black box reuse of structures.

- OPM/Web has an elaborate underlying ontology and a bi-modal notation, which are expressed in a reflective way using Object-Process Diagrams (OPDs) and corresponding Object-Process Language (OPL) paragraphs. Although object-oriented methods have reached the conclusion that a system model should also describe its behavioral aspects (e.g., UML interaction diagrams), for the most part, metamodels of these methods depict their structural parts. Being an object-process approach, OPM enables reflective metamodeling of a complete methodology, including its ontology, language (notation), and development process. The reflective OPM/Web metamodel provides a definition of OPM/Web elements, relations, and consistency constraints. The bimodal, dual representation of OPM increases the processing capability of humans according to the cognitive theory [60] and engages the power of "both sides of the brain" – the visual interpreter and the lingual one.

Table 15 maps OPM/Web features to the requirements that a complete Web application modeling method must satisfy. These requirements were discussed and listed in the introduction.

Table 15. Mapping of OPM/Web concepts to the requirements of a Web application modeling method

Requirement	OPM/Web Concepts
Complex dynamic and distributed architecture	• Physical and informatical things • In-zooming mechanism • Link characterization
Catering to an unlimited number of heterogeneously skilled users	• Refinement-abstraction scaling mechanisms • Agent link characterization • Component reuse
Security and privacy support	• Link characterization by pre-defined processes • Authorization by agent links
Up-to-date, heterogeneous information sources	• Customized multiplicity of features and parts • Partially specified structures and behaviors
Dynamic behavior modeling	• Link characterization • Identical path name • Migration processes of complete code (process classes) and executable code (process instances)

The evaluation of OPM/Web in comparison to a Web application extension of UML [18], presented in this work, has shown that OPM/Web is easier to understand and apply for untrained users. The comparison included comprehension and modeling (constructing) questions on two representative Web applications. Two major factors contributed to the better results obtained by OPM users in the comprehension questions concerning the system dynamics and distribution. One is the single OPM diagram type, which supports the various structural and dynamic aspects throughout the system lifecycle. The other is the ability of OPM to model stand-alone processes, which specify the system's behavior in its entirety. Another experiment which was conducted on 20 advanced students who studied UML and OPM for a year showed similar results. In that experiment, the students had rehearsal tutorials about the core UML and OPM. None of them had studied the extensions of the two modeling languages to the Web application domain. The experiment includes two models and nine comprehension questions. The two models were a webSales system, which handles sales

through the internet, in Conallen's UML and the GLAP system (see chapter 5.5.1) in OPM/Web. The comprehension questions which refer to system dynamics and distribution were better answered using OPM/Web models. The students noted that the single view and the technology-independent model in OPM/Web helped them understand the system purpose, functionality, and architecture.

14. Implementation Issues

Formal visual analysis and design methods have been evolving at a high pace in part due to their claim to be implementation-independent. Developers using these methods communicate with each other on the basis of a common ontology rather than on a specific programming language or technology. Moreover, complex systems often involve various kinds of programming languages. For example, objects in a Web application can be implemented as HTML documents, while processes of the same application can be written in Java. Formal development methods help system architects design systems conceptually and then generate different portions of their models in the most appropriate programming languages.

The translation of a system design to an implementation is increasingly done by automatic code generators. The benefits of using such tools include higher productivity and quality of the developed systems; enabling mechanical and repetitive operations to be done quickly, reliably and uniformly; relieving designers from mundane tasks so they can focus on essence; and enforcing the generation of legible code by human programmers. This tendency to automate code generation is in line with the widespread observation that the most complex task in the implementation phase is generating the design and implementation model at the semantic level and not in the detailed code writing.

Existing code generators define rules for translating visual constructs to corresponding code blocks in the target programming languages. These rules are strict and reflect the insight and the style of the code generator implementers. Changing the translation rules or the visual

constructs usually requires rewriting the code generator. The eXtendible Markup Language (XML) [71], which has become popular as the prime language for the Internet, provides a universal means for communicating between methods in general and for translating models among various programming languages in particular. OPM's code generator uses XML as an infrastructure for translating OPL sentences to various programming languages.

Figure 72 describes the architecture and functionality of the generic OPM code generator in which the transformation rules are external to the code generator and hence can be modified and adapted to different writing styles and implementation frameworks.

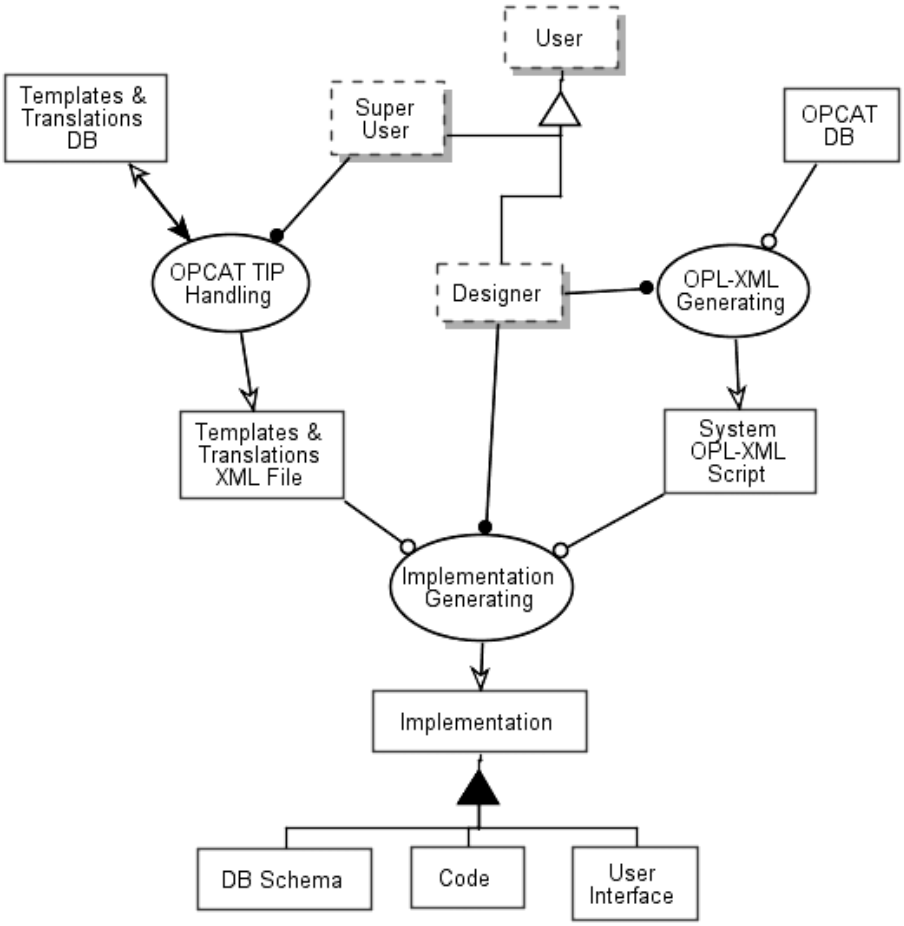


Figure 72. The architecture and functionality of the generic OPM code generator

OPCAT TIP (Templates for Implementation Generation) **Handling** is a component with which **Super Users** can communicate to insert and update conversion rules into the **Templates & Translations DB**. Figure 73 is a snapshot of the OPL tab of the main OPCAT TIP screen. In

this tab the OPL templates are handled. For example, the exhibition sentence, shown in this figure, has three constituents: *ObjectName*, which is mandatory, *ExhibitedObject*, which is a template that can appear 0 or more times, and *Operation*, which is a string that can appear 0 or more times. The XML schema which is automatically generated for this template and appears in the XML tab is:

```
<xs:element name="ObjectExhibitionSentence">
   <xs:complexType>
      <xs:sequence>
         <xs:element name = "ObjectName" type="xs:string"/>
         <xs:element ref="ExhibitedObject" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name = "Operation" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

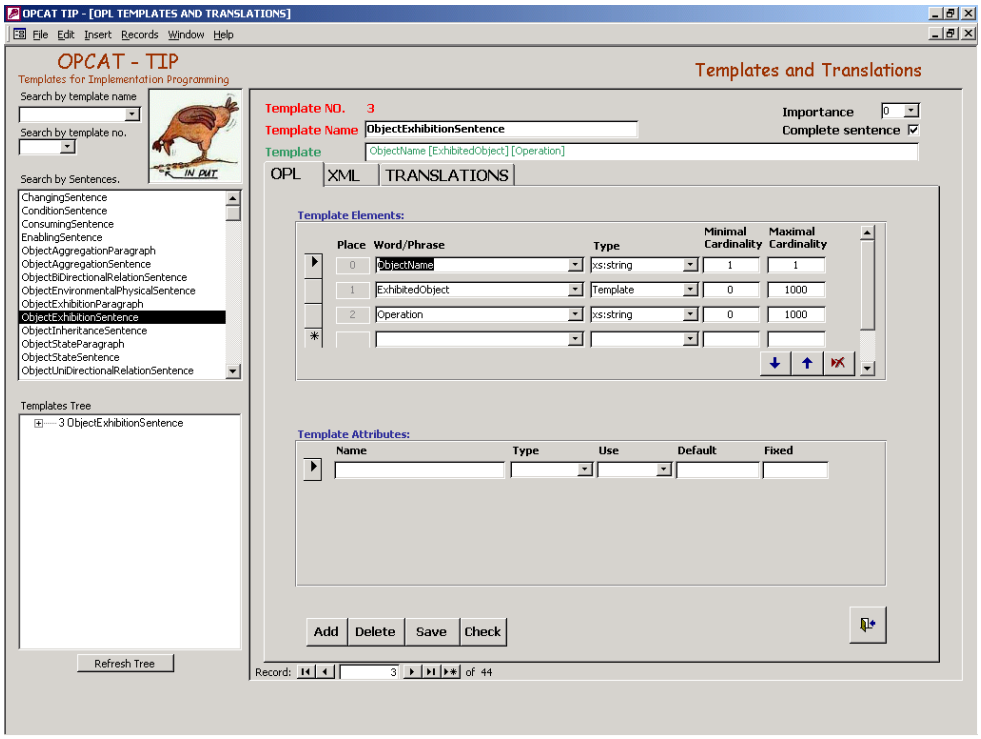


Figure 73. OPCAT TIP screen snapshot – the OPL tab

An example of an OPL sentence of this type is "**A** exhibits **B** and **C**, as well as **Ping.**" The XML presentation of this sentence is:

```
<ObjectExhibitionSentence>
        <ObjectName> A </ObjectName>
        <ExhibitedObject[10]> <ObjectName> B </ObjectName> </ExhibitedObject>
        <ExhibitedObject> <ObjectName> C </ObjectName> </ExhibitedObject>
        <Operation> Ping </Operation>
</ObjectExhibitionSentence>
```

Figure 74 is a snapshot of the translations part of the OPCAT TIP screen. In this tab the subject template is translated into various programming languages, for example, HTML and Java, using pre-defined functions, listed in Table 16. Figure 75 presents the "operation details" screen, in which the functions (with the given parameters) are called when some conditions are satisfied.

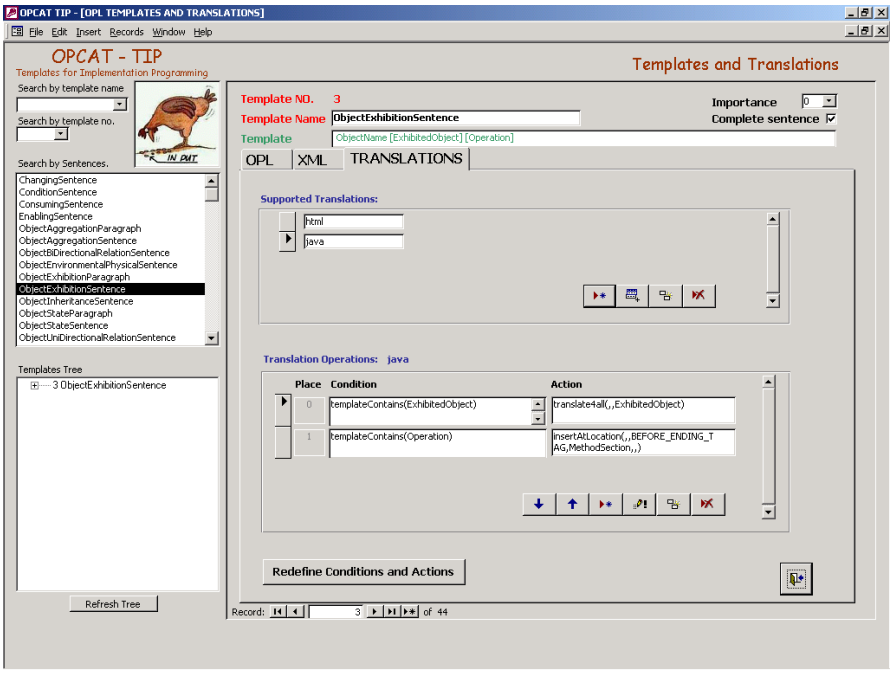


Figure 74. OPCAT TIP screen snapshot – the Translations tab

[10] *ExhibitedObject* is a template which contains a multiplicity constraint and *ObjectName*. The multiplicity is 1 by default and, hence, this sub-element does not appear here.

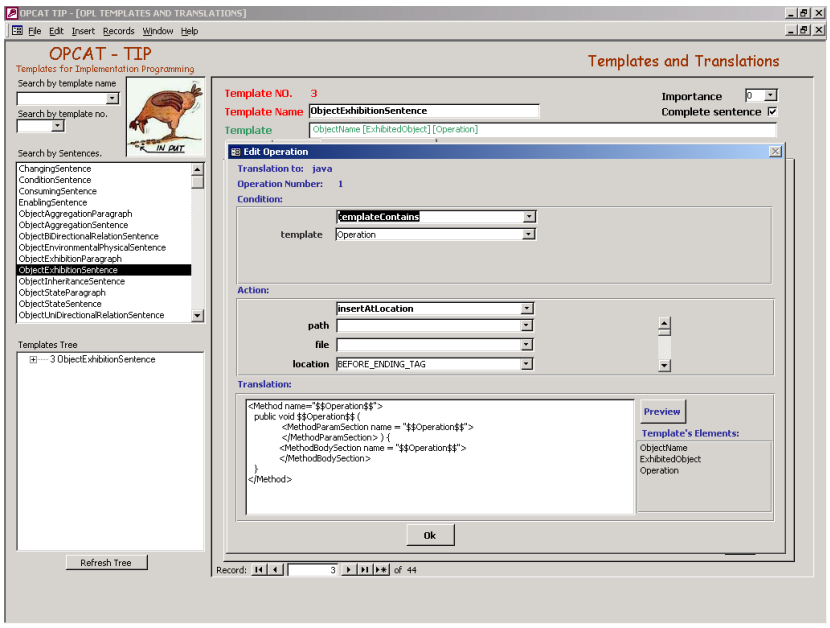


Figure 75. OPCAT TIP screen snapshot – operation details

OPCAT TIP generates a **Templates and Translations XML File**, which includes an XML schema for each OPL template (see Appendix B) and its translations (conditions + actions) to the various supported programming languages.

OPL-XML Generating in Figure 72 uses **OPCAT DB** in order to generate a **System OPL-XML Script**. This script is the basis for OPL paragraphs of OPM specifications on one hand and an input for the **Implementation Generating** process on the other hand. **Implementation Generating** gets the **Templates and Translations XML File** from **OPCAT TIP Handling** and the **System OPL-XML Script** from **OPL-XML Generating**. It yields the system **Implementation**, which is composed of **DB Schema**, **Code**, and **User Interface**. **Implementation Generating** finds matches between the XML file representing a specific system and the XML schema in the **Templates and Translations XML File**. When such a match is found, the implementation generator acts according to the translation rules. After creating the implementation files, another pass is required to transform these files, which are in XML format, to regular implementation files in the target languages. This is done by changing the tags to comments or omitting them.

Table 16. Supported functions in OPCAT TIP translations

Type	Function Signature	Function Description
Condition Functions	none()	Always true
	templateContains (template t)	The current OPL template contains t as its sub-element
	translationContains (path p, file f, tagName tn, attributeName an, attributeValue av)	The translation file f at path p contains <tn an=”av”…>
	equals (element e, value v)	The value of the sub-element or sub-attribute e of the current OPL template equals to v
	Complex condition	A combination of several atomic conditions using and, or, and not connectors
Action Functions	createDirectory (path p)	Creates a directory in path p
	createFile (path p, file f, translation t)	Creates a file named f at path p with the initial content of t
	translate4all (path p, file f, template t)	Translates all the sub-elements of type t of the current OPL template into file f at location p
	replaceContent (path p, file f, tagName tn, attributeName an, attributeValue av, translation t)	Replaces the content of <tn an=”av”…> in file f at path p with the content of t
	insertAtLocation (path p, file f, location l, tagName tn, attributeName an, attributeValue av, translation t)	Inserts the content of t at the location l in respect to <tn an=”av”…> in file f at path p. l can be one of BEFORE_STARTING_TAG, BEFORE_ENDING_TAG, AFTER_STARTING_TAG, AFTER_ENDING_TAG.

This generic and flexible implementation generator supplies a complete environment for generating Web application models to various target languages, including HTML, Java, PHP, and WSDL [98].

Appendix A. OPM/Web Concepts and Symbols

Table 17. Entities – Things and States

	Entity Type	Entity Symbol
Object	Systemic, informatical object	
	Environmental, informatical object	
	Systemic, physical object	
	Environmental, physical object	
Process	Systemic, informatical process	
	Environmental, informatical process	
	Systemic, physical process	
	Environmental, physical process	
State	Regular state	
	Initial state	
	Final state	
	Default state	
Package		

Table 18. Structural Relations, their OPD symbols, and OPL sentences

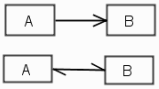
Structural Relation Name	OPD Symbol	OPL Sentence
Aggregation-Participation		**A** consists of **B.**
Exhibition-Characterization		**A** exhibits **B.**
Generalization-Specialization		**B** is an **A.**
Classification-Instantiation		**B**is an instance of **A.**
Tagged Structural Link		**A** relates to **B.** **A** and **B** are equivalent.
XOR relation		E.g., **A** relates to either **B**or **C.**
OR relation		E.g., **A** relates to **B**or **C.**

Table 19. Procedural Links, their OPD symbols, and OPL sentences

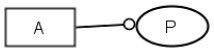
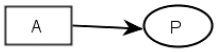
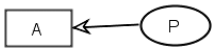
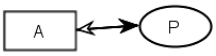
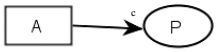
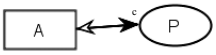
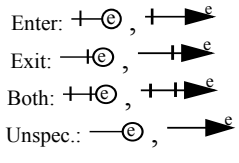
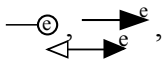
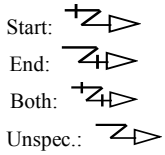
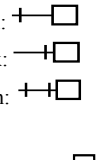
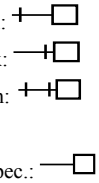
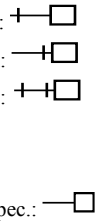
Type	Link Name	Semantics	OPD Symbol	OPL Sentence
Enabling Links	Instrument	The process requires the entity, but does not change it during execution.		**P** requires **A.**
Transforming Links	Consumption	The process consumes the entity.		**P** consumes **A.**
	Result	The process generates (creates) the entity.		**P** yields **A.**
	Effect	The process changes (affects) the thing.		**P** affects **A.**
Conditional Links	Instrument	The process occurs if the entity exists (in some state). The process requires the entity.		**P** occurs if **A** exists. **P** requires **A.**
	Consumption	The process occurs if the entity exists (in some state). The process consumes the entity.		**P** occurs if **A** exists. **P** consumes **A.**
	Effect	The process occurs if the thing exists. The process changes (affects) the thing.		**P** occurs if **A** exists. **P** affects **A.**
Logical Relations	XOR relation			E.g., **P** affects either **A** or **B.**
	OR relation			E.g., **P** affects **A** or **B.**

Table 20. Event links, their semantics and symbols

Event Type	Semantics	OPD Symbol	OPL Sentence
Agent	The process is triggered by the intelligent object.		**A** handles **P.**
State Change	The process is triggered when the object enters or exits the state. The object may be changed.		**A** triggers **P** when it enters/exists/enters or exists **st.** **St A** triggers **P.**
General Event	The process is triggered when the object or process is changed or cause external stimuli. The object may be consumed or changed.		If the object has states: **A** triggers **P** when it changes**.** Otherwise: **A** triggers **P.**
Invocation	The process is triggered when the source process starts or ends.		**P** invokes **P1** when it starts/ends/starts or ends**.** **P** invokes **P1.**
Minimal or Maximal State Timeout	The process is triggered when the object violates its minimal or maximal time constraints for staying at the state.		**A** triggers **P** when **st** lasts less than **Time**/ more than **Time**/less than **Time** or more than **Time.** Timeout of **st A** triggers **P.**
Minimal or Maximal Process Timeout	The process is triggered when the process violates its minimal or maximal execution time constraints.		**P1** triggers **P** when it lasts less than **Time**/ more than **Time**/less than **Time** or more than **Time.** Timeout of **P1** triggers **P.**
Reaction Timeout	The process is triggered when the event link violates its minimal or maximal triggering time constraints.		This link triggers **P** when its reaction time lasts less than **Time**/ more than **Time**/less than **Time** or more than **Time.** This link timeout triggers **P.**
XOR relation			E.g., **A** triggers either **P** or **Q** when it changes**.**
OR relation			E.g., **A** triggers **P** or **Q** when it changes**.**

Comments:

1. Unspec. stands for "unspecified".
2. The OPL sentences in this table are for the event aspect of the link. For each event link, an additional OPL sentence, which represents its procedural aspect, should be added.

Appendix B. The XML Schema of the Object-Process Language (OPL)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="AffectingClause">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="AffectingSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ProcessName" type="xs:string"/>
                <xs:element ref="AffectingClause" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="AggregatedObject">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="MinimalCardinality" type="xs:integer"/>
                <xs:element name="MaximalCardinality" type="xs:integer"/>
                <xs:element name="ObjectName" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ChangingClause">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element name="SourceValueName" type="xs:string"/>
                <xs:element name="DestinationValueName" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ChangingSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ProcessName" type="xs:string"/>
                <xs:element ref="ChangingClause" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ConditionClause">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element name="ValueName" type="xs:string" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ConditionSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="PathName" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="ProcessName" type="xs:string"/>
                <xs:element ref="ConditionClause" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ConsumingClause">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ConsumedObjectName" type="xs:string"/>
                <xs:element name="ValueName" type="xs:string" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
```

```
</xs:element>
<xs:element name="ConsumingSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element ref="ConsumingClause" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="DestinationWithCardinality">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="MinimalCardinality" type="xs:integer"/>
            <xs:element name="MaximalCardinality" type="xs:integer"/>
            <xs:element name="DestinationName" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="sourceName" type="xs:string" use="required"/>
        <xs:attribute name="relationName" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="EnablingClause">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ObjectName" type="xs:string"/>
            <xs:element name="StateName" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="EnablingSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element ref="EnablingClause" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ExhibitedObject">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="MinimalCardinality" type="xs:integer"/>
            <xs:element name="MaximalCardinality" type="xs:integer"/>
            <xs:element name="AttributeName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="MaxTimeValue">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Years" type="xs:integer" minOccurs="0"/>
            <xs:element name="Months" type="xs:integer" minOccurs="0"/>
            <xs:element name="Weeks" type="xs:integer" minOccurs="0"/>
            <xs:element name="Days" type="xs:integer" minOccurs="0"/>
            <xs:element name="Hours" type="xs:integer" minOccurs="0"/>
            <xs:element name="Minutes" type="xs:integer" minOccurs="0"/>
            <xs:element name="Seconds" type="xs:integer" minOccurs="0"/>
            <xs:element name="MilliSeconds" type="xs:integer" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="MinTimeValue">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Years" type="xs:integer" minOccurs="0"/>
            <xs:element name="Months" type="xs:integer" minOccurs="0"/>
            <xs:element name="Weeks" type="xs:integer" minOccurs="0"/>
            <xs:element name="Days" type="xs:integer" minOccurs="0"/>
            <xs:element name="Hours" type="xs:integer" minOccurs="0"/>
            <xs:element name="Minutes" type="xs:integer" minOccurs="0"/>
            <xs:element name="Seconds" type="xs:integer" minOccurs="0"/>
            <xs:element name="MilliSeconds" type="xs:integer" minOccurs="0"/>
```

```
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectAggregationParagraph">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ObjectAggregationSentence"/>
                <xs:element ref="ThingSentenceSet" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectAggregationSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element ref="AggregatedObject" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectBiDirectionalRelationSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="SourceWithCardinality"/>
                <xs:element ref="DestinationWithCardinality"/>
                <xs:element name="RelationName" type="xs:sting "/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectEnvironmentalPhysicalSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element name="environmental" type="xs:Boolean"/>
                <xs:element name="physical" type="xs:Boolean"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectExhibitionParagraph">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ObjectExhibitionSentence"/>
                <xs:element ref="ThingSentenceSet" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectExhibitionSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element ref="ExhibitedObject" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="Operation" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectInheritanceSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type="xs:string"/>
                <xs:element name="InheritanceFatherName" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ObjectStateParagraph">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ObjectStateSentence"/>
                <xs:element ref="StateClause" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

```
<xs:element name="ObjectStateSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ObjectName" type="xs:string"/>
            <xs:element name="DefaultState" type="xs:string" minOccurs="0"/>
            <xs:element name="StateName" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ObjectUniDirectionalRelationSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ObjectName" type="xs:string"/>
            <xs:element name="RelationName" type="xs:string"/>
            <xs:element ref="DestinationWithCardinality"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="OPLscript">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="ThingSentenceSet" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="systemName" type="xs:string" use="required"/>
        <xs:attribute name="targetDirectory" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessAggregationParagraph">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="ProcessAggregationSentence"/>
            <xs:element ref="ThingSentenceSet" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessAggregationSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="AggregatedProcess" type="xs:string" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessBiDirectionalRelationSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="DestinationProcessName" type="xs:string"/>
            <xs:element name="RelationName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessEnvironmentalPhysicalSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="environmental" type="xs:Boolean"/>
            <xs:element name="physical" type="xs:Boolean"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessExhibitionParagraph">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="ProcessExhibitionSentence"/>
            <xs:element ref="ThingSentenceSet" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessExhibitionSentence">
```

```
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="OperationName" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="ExhibitedObject" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessInheritanceSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="InheritanceFatherName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessInZoomingParagraph">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="ProcessInZoomingSentence"/>
            <xs:element ref="ThingSentenceSet" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessInZoomingSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="InZoomedProcessName" type="xs:string" minOccurs="0"
            maxOccurs="unbounded"/>
            <xs:element name="InZoomedObjectName" type="xs:string" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ProcessUniDirectionalRelationSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element name="RelationName" type="xs:string"/>
            <xs:element name="DestinationProcessName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ResultingClause">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ResultantObjectName" type="xs:string"/>
            <xs:element name="ValueName" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="ResultingSentence">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProcessName" type="xs:string"/>
            <xs:element ref="ResultingClause" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="SourceWithCardinality">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="MinimalCardinality" type="xs:integer"/>
            <xs:element name="MaximalCardinality" type="xs:integer"/>
            <xs:element name="SourceName" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="destinationName" type="xs:string" use="required"/>
        <xs:attribute name="relationName" type="xs:string" use="required"/>
    </xs:complexType>
```

```
    </xs:element>
    <xs:element name="StateClause">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="StateName" type="xs:string"/>
                <xs:element name="Initial" type="xs:Boolean" minOccurs="0"/>
                <xs:element name="Final" type="xs:Boolean" minOccurs="0"/>
                <xs:element ref="MinTimeValue" minOccurs="0"/>
                <xs:element ref="MaxTimeValue" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ThingSentenceSet">
        <xs:complexType>
            <xs:choice>
                <xs:sequence>
                    <xs:element ref="ObjectEnvironmentalPhysicalSentence" minOccurs="0"/>
                    <xs:element ref="ObjectInheritanceSentence" minOccurs="0"/>
                    <xs:element ref="ObjectStateParagraph" minOccurs="0"/>
                    <xs:element ref="ObjectExhibitionParagraph" minOccurs="0"/>
                    <xs:element ref="ObjectAggregationParagraph" minOccurs="0"/>
                    <xs:element ref="ObjectUniDirectionalRelationSentence" minOccurs="0"
                    maxOccurs="unbounded"/>
                    <xs:element ref="ObjectBiDirectionalRelationSentence" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:sequence>
                    <xs:element ref="TypeDeclarationSentence"/>
                </xs:sequence>
                <xs:sequence>
                    <xs:element ref="ProcessEnvironmentalPhysicalSentence" minOccurs="0"/>
                    <xs:element ref="ProcessInheritanceSentence" minOccurs="0"/>
                    <xs:element ref="ProcessExhibitionParagraph" minOccurs="0"/>
                    <xs:element ref="ProcessAggregationParagraph" minOccurs="0"/>
                    <xs:element ref="ProcessUniDirectionalRelationSentence" minOccurs="0"
                    maxOccurs="unbounded"/>
                    <xs:element ref="ProcessBiDirectionalRelationSentence" minOccurs="0"
                    maxOccurs="unbounded"/>
                    <xs:element ref="ConditionSentence" minOccurs="0"/>
                    <xs:element ref="EnablingSentence" minOccurs="0"/>
                    <xs:element ref="AffectingSentence" minOccurs="0"/>
                    <xs:element ref="ChangingSentence" minOccurs="0"/>
                    <xs:element ref="ConsumingSentence" minOccurs="0"/>
                    <xs:element ref="ResultingSentence" minOccurs="0"/>
                    <xs:element ref="ProcessInZoomingParagraph" minOccurs="0"/>
                </xs:sequence>
            </xs:choice>
            <xs:attribute name="subjectThingName" type="xs:string" use="required"/>
            <xs:attribute name="subjectExhibitionFatherName" type="xs:string" use="required"/>
            <xs:attribute name="subjectAggregationFatherName" type="xs:string" use="required"/>
            <xs:attribute name="systemName" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="TypeDeclarationSentence">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectName" type ="xs:string"/>
                <xs:element name="ObjectType" type="xs:string"/>
                <xs:element name="InitialValue" type="xs:string" minOccurs="0"/>
                <xs:element name="ObjectScope" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Appendix C. The OPM/Web vs. Conallen's UML experiment Forms

The Project Management System – Models and Questions

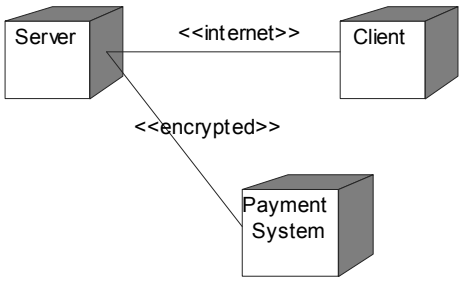


Figure 76. Conallen's UML specification of the project management system – Deployment diagram

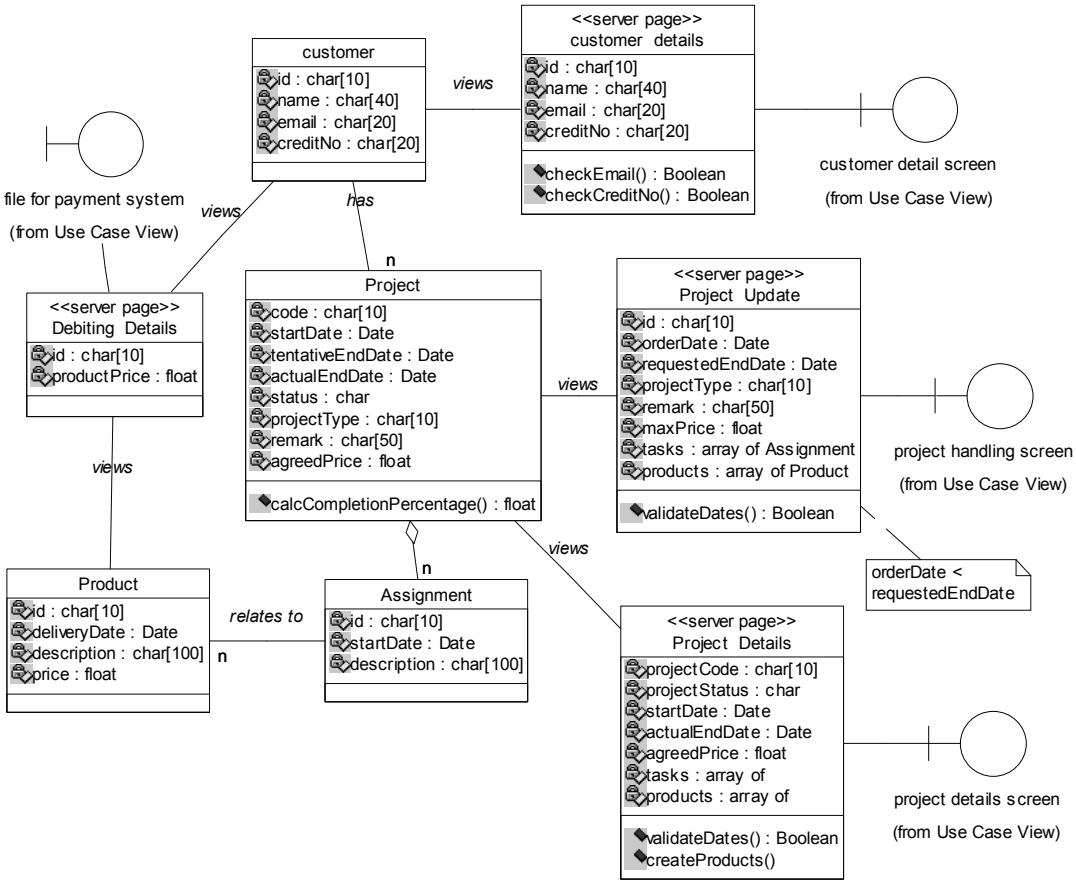


Figure 77. Conallen's UML specification of the project management system – Class diagram

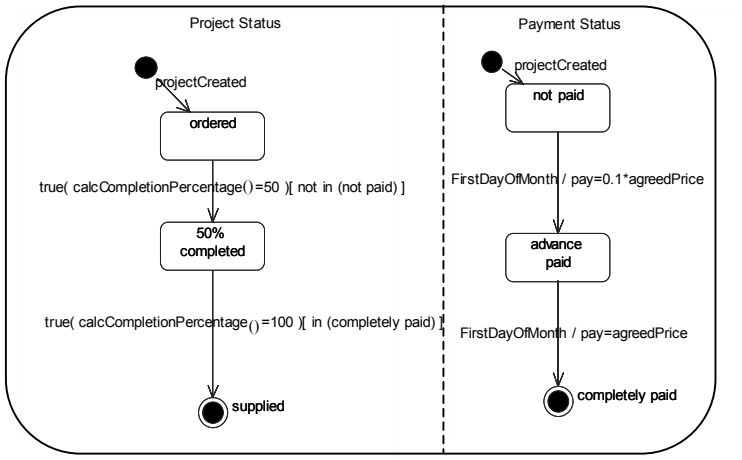


Figure 78. Conallen's UML specification of the project management system – Statechart of project and payment status

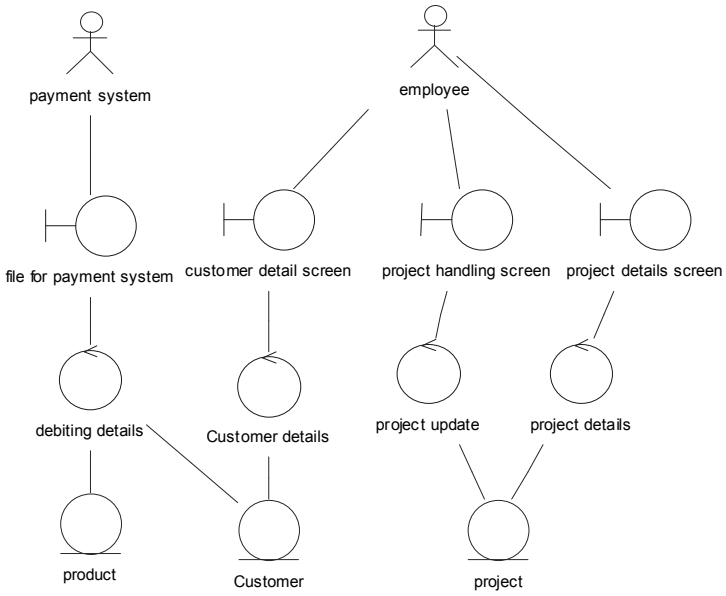


Figure 79. Conallen's UML specification of the project management system – Site map diagram

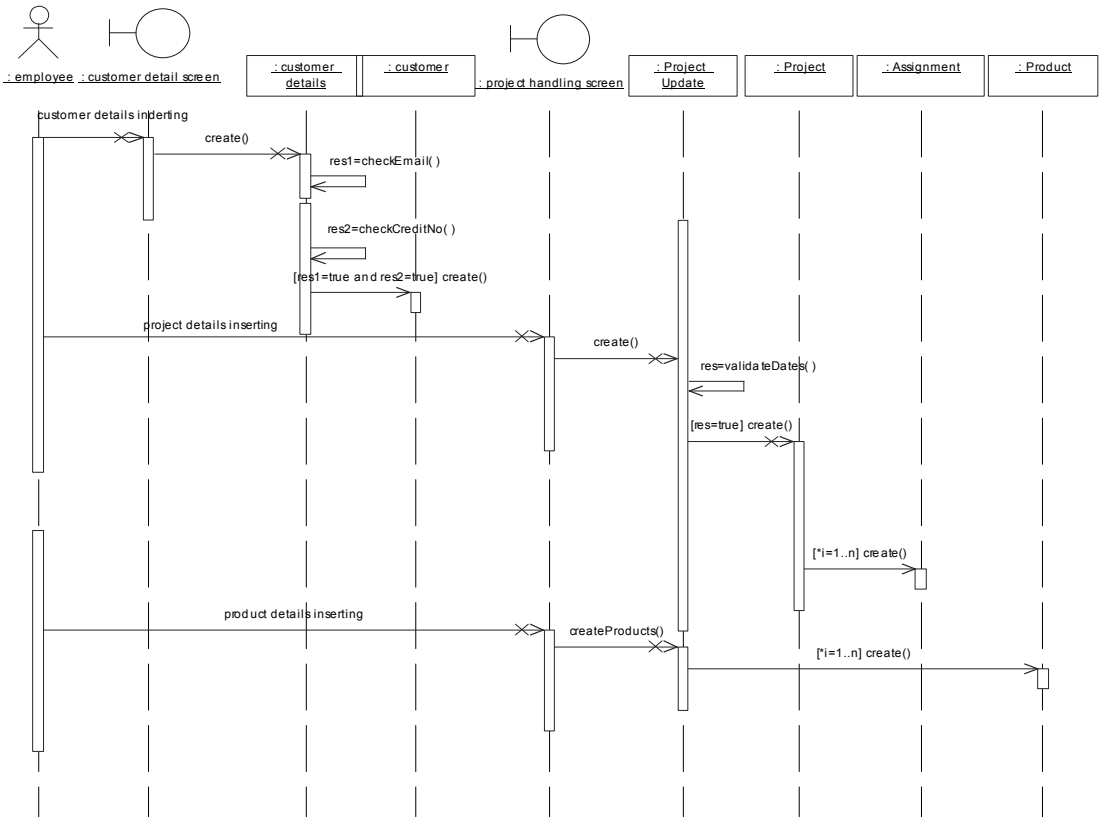


Figure 80. Conallen's UML specification of the project management system – Sequence diagram of Project Order Handling

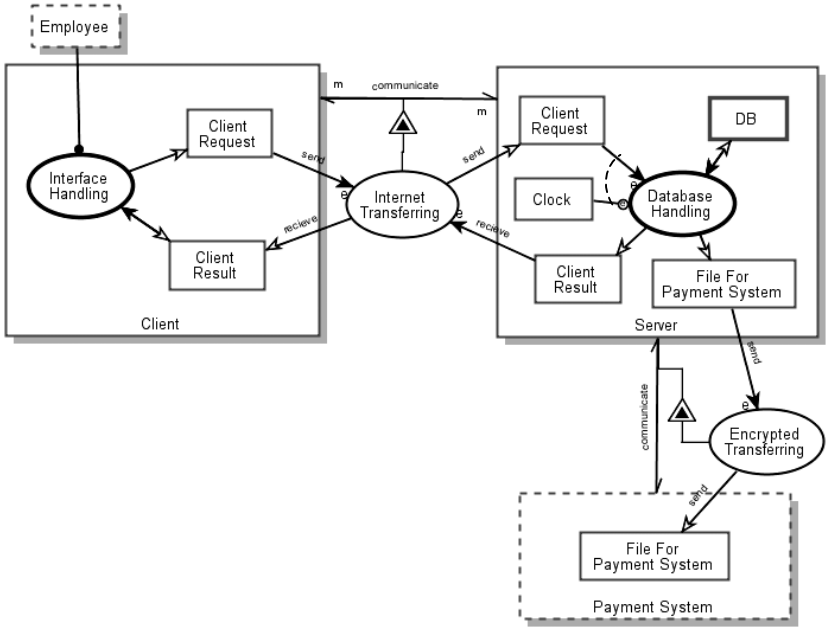


Figure 81. OPM/Web specification of the project management system – Top level diagram

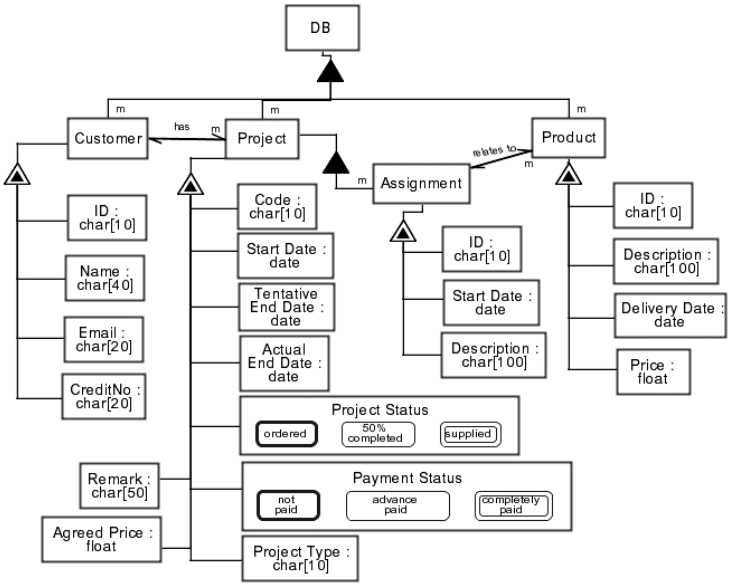


Figure 82. OPM/Web specification of the project management system – DB unfolded

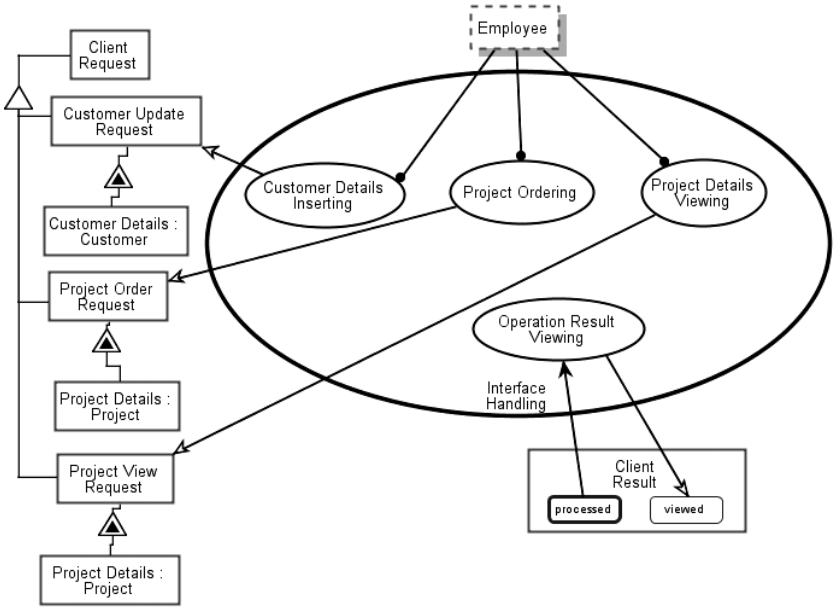


Figure 83. OPM/Web specification of the project management system – Interface Handling in-zoomed

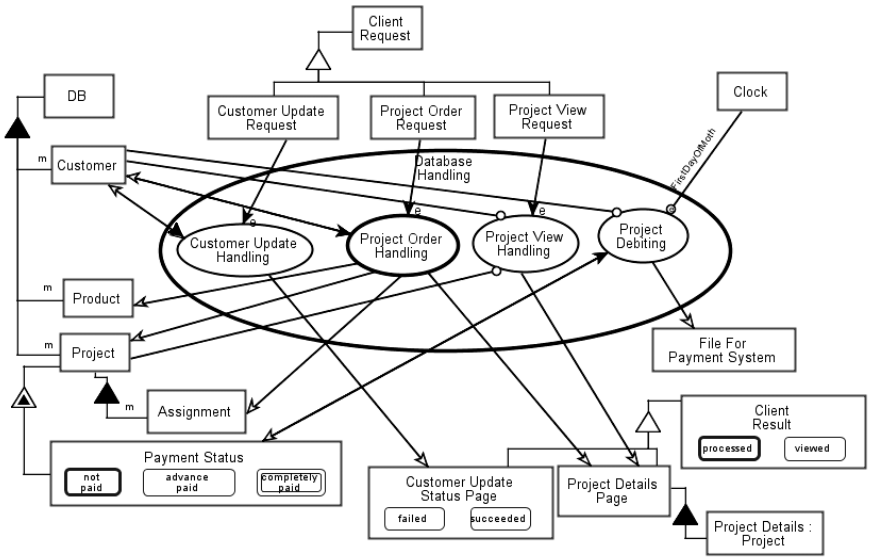


Figure 84. OPM/Web specification of the project management system – Database Handling in-zoomed

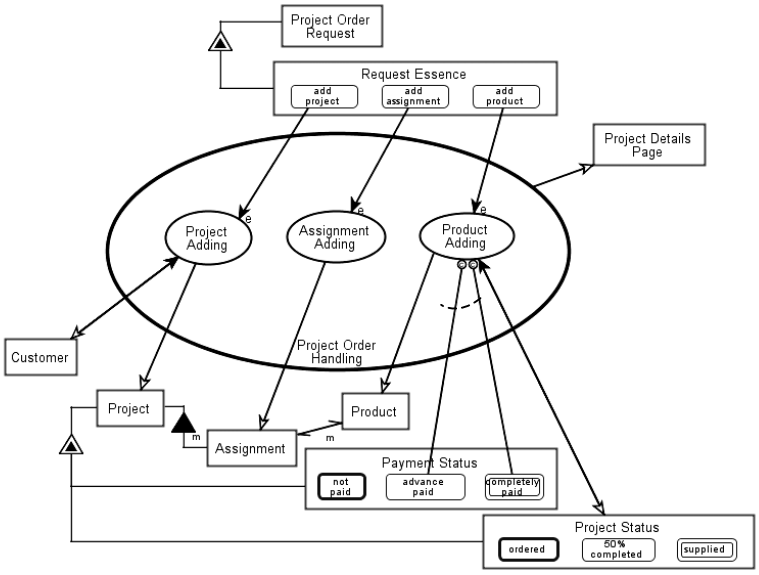


Figure 85. OPM/Web specification of the project management system – Project Order Handling in-zoomed

Answer all the following questions about the project management system model.

1. What are the classes which form the application's database?
2. Does the **structure** of the system support the following query: "who is the customer that ordered a specific product?" Explain.
3. What is the trigger of project order handling? From which diagram did you conclude it?
4. Is it possible that only the advance (10%) of a project of which 50% had been completed was paid? Explain.
5. What database classes are affected by project order handling? How? (i.e., are they created, destroyed, or changed?)
6. What is the navigation order in the application? How did you conclude it?
7. What is the internal architecture (nodes and links) of the system?
8. What are the system activities from the moment the employee connects to the site and till he/she gets the project details? From which diagrams did you conclude it?
9. Add to the model a possibility to view a report of all the projects which 50% of their assignments were completed, but the projects have not supplied yet. The report will be viewed as a result of a manager request after he/she inserts a date range for the report.

Figure 86. Questions related to the project management system

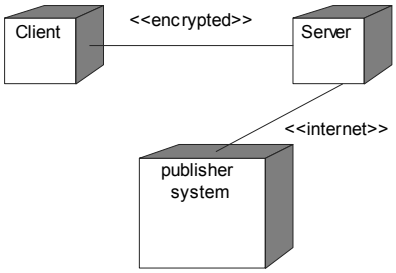


Figure 87. Conallen's UML specification of the book ordering system – Deployment diagram

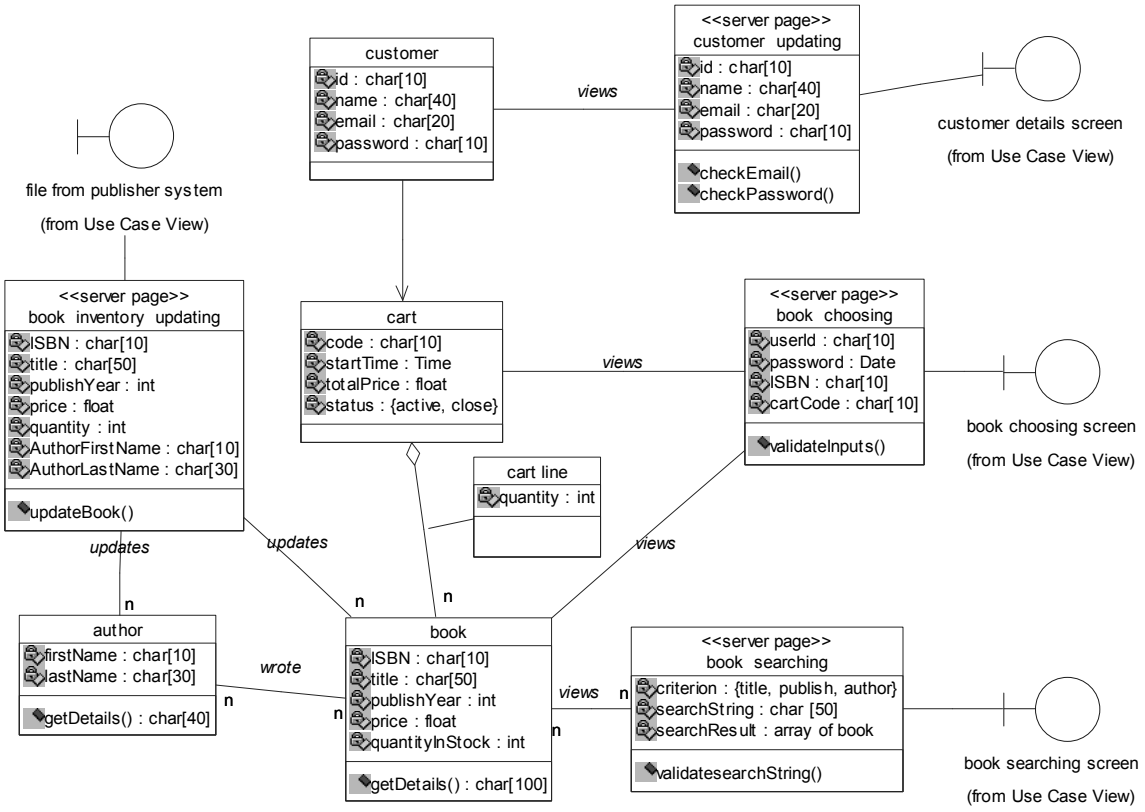


Figure 88. Conallen's UML specification of the book ordering system – Class diagram

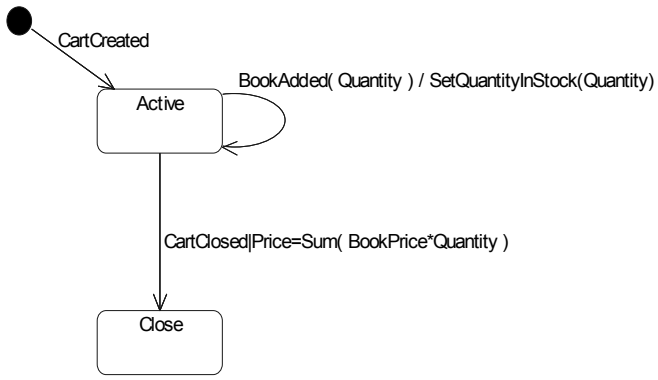


Figure 89. Conallen's UML specification of the book ordering system – Statechart of cart status

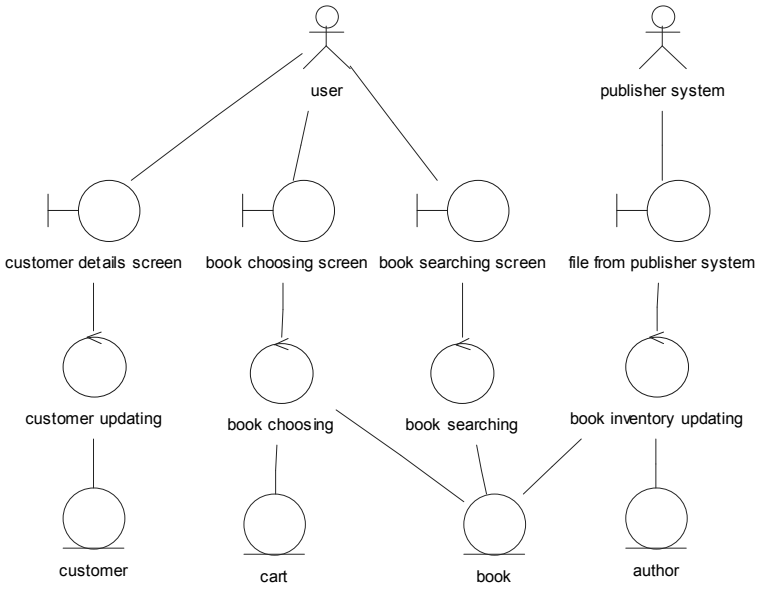


Figure 90. Conallen's UML specification of the book ordering system – Site map diagram

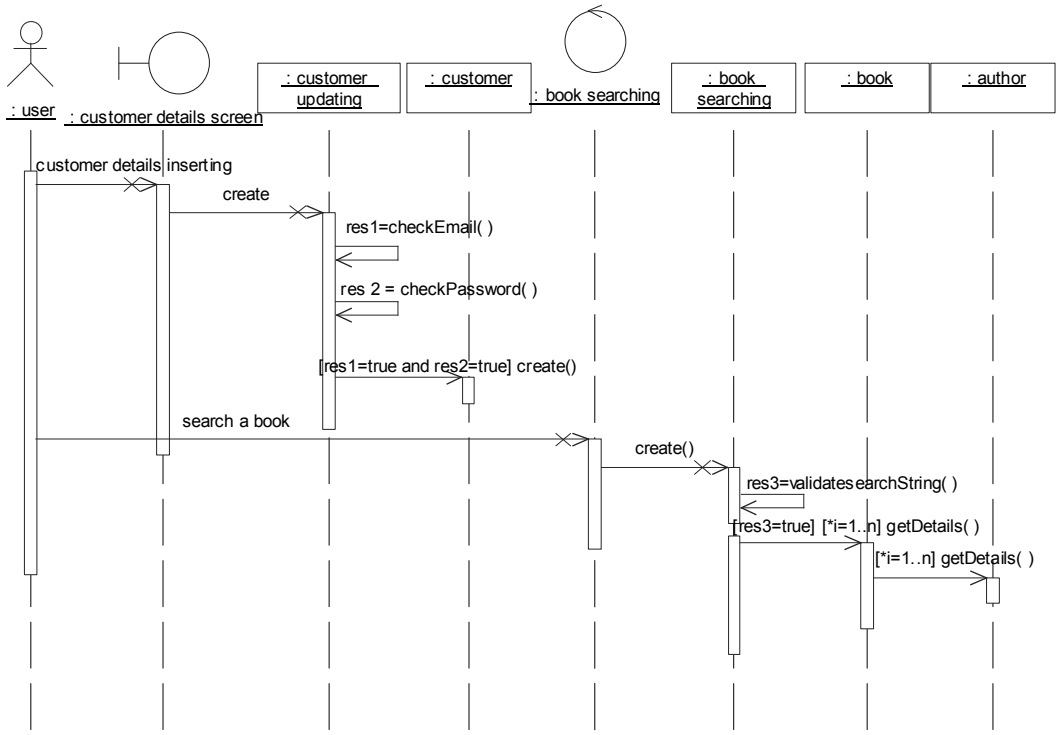


Figure 91. Conallen's UML specification of the book ordering system – Sequence diagram of book searching

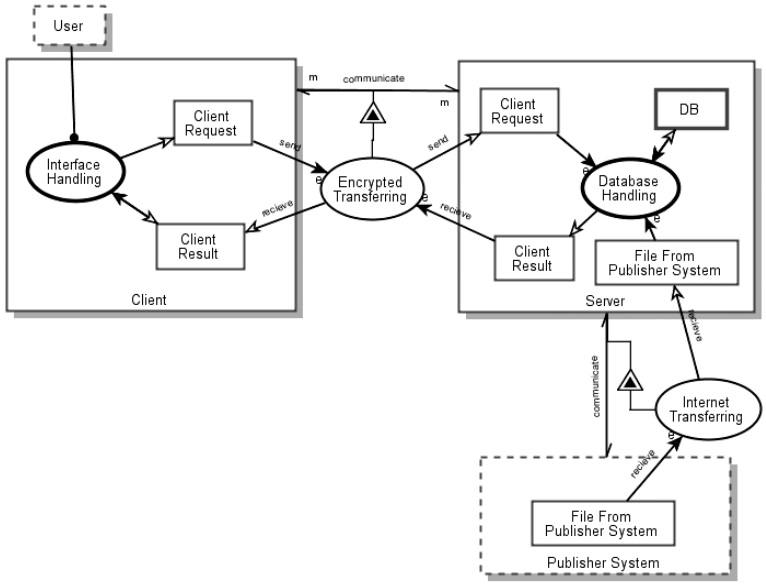


Figure 92. OPM/Web specification of the book ordering system – Top level diagram

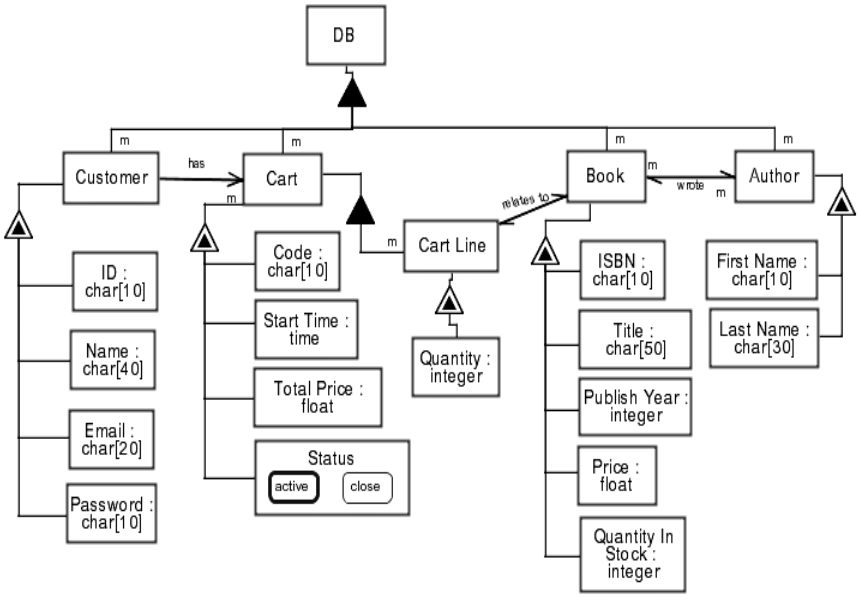


Figure 93. OPM/Web specification of the book ordering system – DB unfolded

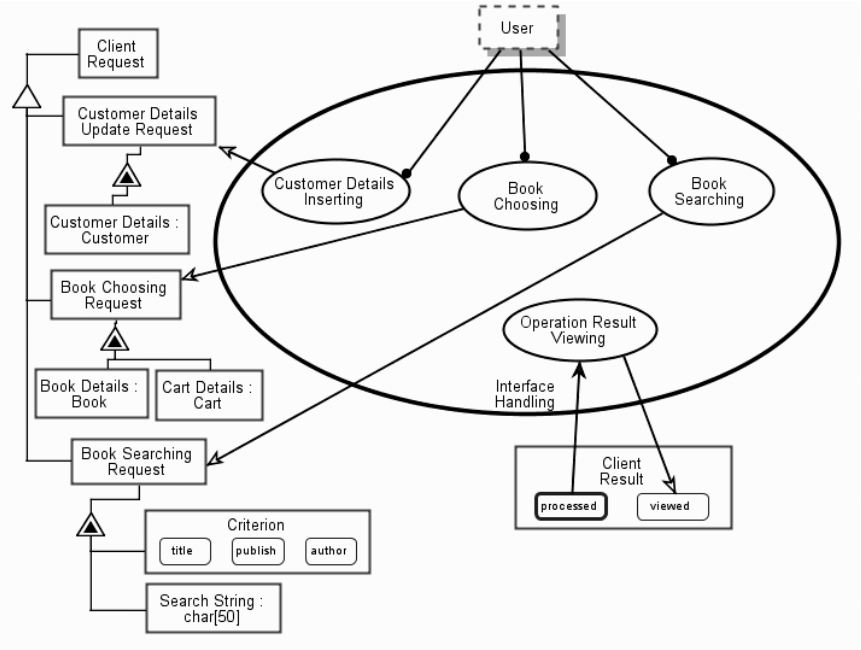


Figure 94. OPM/Web specification of the book ordering system – Interface Handling in-zoomed

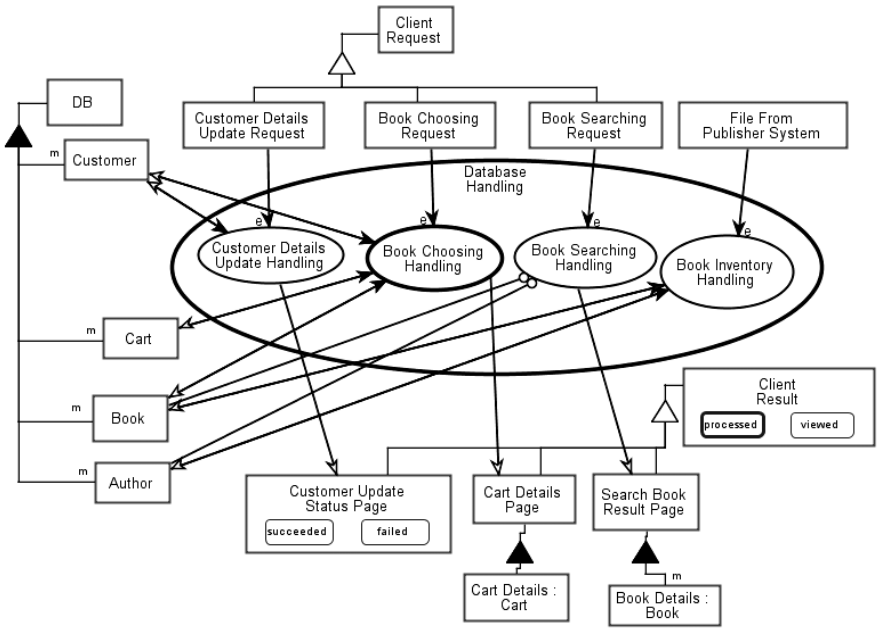


Figure 95. OPM/Web specification of the book ordering system – Database Handling in-zoomed

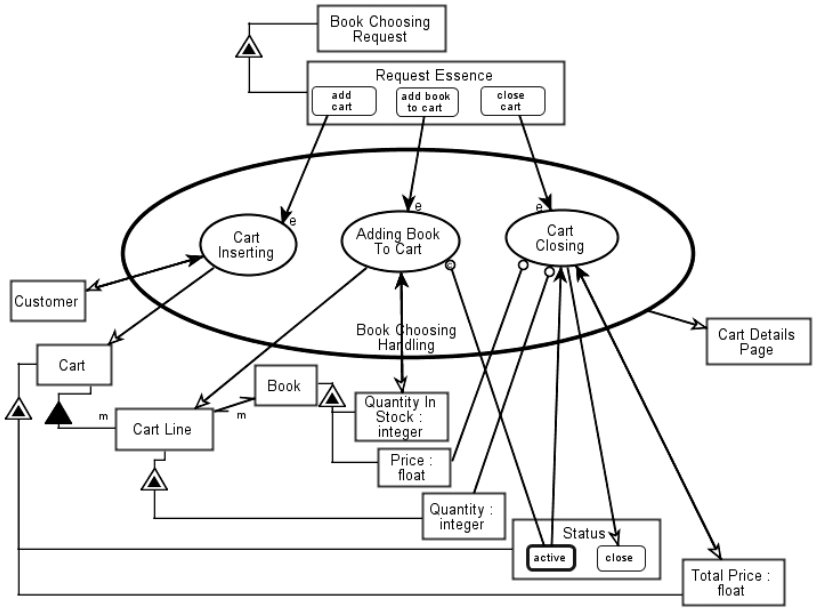


Figure 96. OPM/Web specification of the book ordering system – Book Choosing Handling in-zoomed

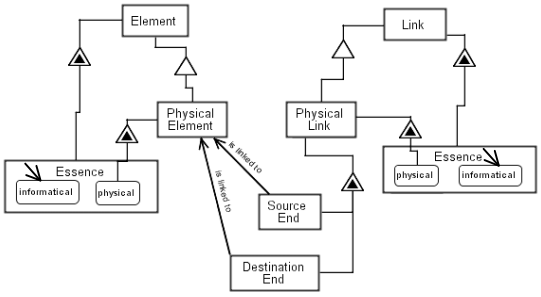
Answer all the following questions about the book ordering system model.

1. Which types of pages can the user view? What is the information presented at each page?
2. Does the **structure** of the system support the following query: ”who are the customers that ordered a specific book?” Explain.
3. What is the trigger of customer details update handling? From which diagram did you conclude it?
4. What are the inputs and outputs of book searching? What are the database classes used in this process?
5. What database classes are affected by customer details update handling? How? (i.e., are they created, destroyed, or changed?)
6. What is the navigation order in the application? How did you conclude it?
7. What is the internal architecture (nodes and links) of the system?
8. What are the states of cart from the moment it is created till it is closed? What are the activities in each state and how does the system transform between the states? From which diagrams did you conclude it?
9. Add to the model a possibility to print a report of all the active carts. The report will be automatically printed at the beginning of each week.

Figure 97. Questions related to the book ordering system

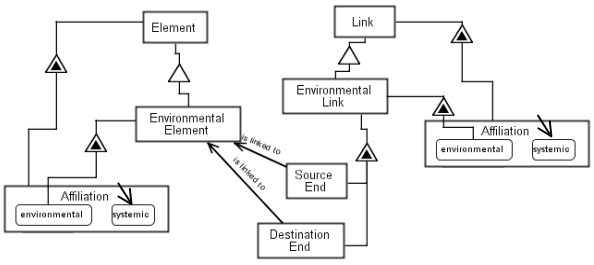
Appendix D. Definitions of Link Essence, Affiliation, and Scope

There are three rules in OPM for determining the essence, affiliation, and scope values of a link. The first rule, the **Link Essence** rule shown in Figure 98, states that a **physical Link** connects two **physical Elements**. The second rule, the **Link Affiliation** rule specified by the OPM model in Figure 99, states that an **environmental Link** connects two **environmental Elements**. The third and last rule, the **Link Scope** rule depicted in Figure 100, requires that the **Scope** value of a **Link** is the widest of the **Scope** values of the two connected **Elements**, where **public**, **protected**, and **private**, are the widest, intermediate and most narrow **Scope** values, respectively.



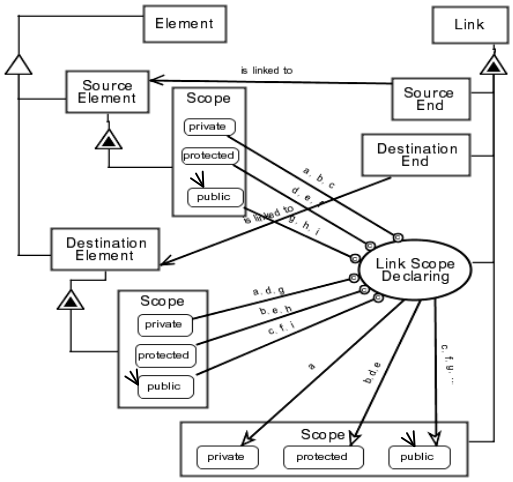
Element exhibits **Essence.**
Essence can be i**nformatical**, which is the default, or **physical.**
Physical Element is an **Element**, the **Essence** of which is **physical.**
Link exhibits **Essence.**
Essence can be i**nformatical**, which is the default, or **physical.**
Physical Link is a **Link**, the **Essence** of which is **physical.**
Physical Link exhibits **Source End** and **Destination End.**
Source End is linked to Physical Element.
Destination End is linked to Physical Element.

Figure 98. An OPM specification of the **Link Essence** rule



Element exhibits **Affiliation.**
Affiliation can be **systemic**, which is the default, or **environmental.**
Environmental Element is an **Element**, the **Affiliation** of which is **environmental.**
Link exhibits **Affiliation.**
Affiliation can be **systemic**, which is the default, or **environmental.**
Environmental Link is a **Link**, the **Affiliation** of which is **environmental.**
Environmental Link exhibits **Source End** and **Destination End.**
Source End is linked to Environmental Element.
Destination End is linked to Environmental Element.

Figure 99. An OPM specification of the **Link Affiliation** rule



Source Element is an **Element.**
Source Element exhibits **Scope.**
Scope can be **public** by default, **protected**, or **private.**
Destination Element is an **Element.**
Destination Element exhibits **Scope.**
Scope can be **public** by default, **protected**, or **private.**
Link exhibits **Source End**, **Destination End**, and **Scope**, as well as **Link Scope Declaring.**
Source End is linked to Source Element.
Destination End is linked to Destination Element.
Scope can be **public** by default, **protected**, or **private.**
Following path **a**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **private** and **Scope** of **Destination Element** is **private.**
Following path **a**, **Link Scope Declaring** yields **private Scope** of **Link.**
Following path **b**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **private** and **Scope** of **Destination Element** is **protected.**
Following path **b**, **Link Scope Declaring** yields **protected Scope** of **Link.**
Following path **c**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **private** and **Scope** of **Destination Element** is **public.**
Following path **c**, **Link Scope Declaring** yields **public Scope** of **Link.**
Following path **d**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **protected** and **Scope** of **Destination Element** is **private.**
Following path **d**, **Link Scope Declaring** yields **protected Scope** of **Link.**
Following path **e**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **protected** and **Scope** of **Destination Element** is **protected.**
Following path **e**, **Link Scope Declaring** yields **protected Scope** of **Link.**
Following path **f**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **protected** and **Scope** of **Destination Element** is **public.**
Following path **f**, **Link Scope Declaring** yields **public Scope** of **Link.**
Following path **g**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **public** and **Scope** of **Destination Element** is **private.**
Following path **g**, **Link Scope Declaring** yields **public Scope** of **Link.**
Following path **h**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **public** and **Scope** of **Destination Element** is **protected.**
Following path **h**, **Link Scope Declaring** yields **public Scope** of **Link.**
Following path **i**, **Link Scope Declaring** occurs if **Scope** of **Source Element** is **public** and **Scope** of **Destination Element** is **public.**
Following path **i**, **Link Scope Declaring** yields **public Scope** of **Link.**

Figure 100. An OPM specification of the **Link Scope**

OCL Alternative Rule Formulation

OCL can be used as an alternative to OPM for defining the static invariants expressed before. These constraints demonstrate the declarative nature of OCL vs. the procedural nature of OPM. While OCL defines invariants which should be preserved in the model in any point of time, OPM defines a lower level processes which check the consistency rule and update the models accordingly.

a. A physical link connects two physical elements.

```
Link
(self.essence = #physical)
implies (self.source_End.essence = #physical and
self.destination_End.essence = #physical)
```

b. An environmental link connects two environmental elements.

```
Link
(self.affiliation = #environmental)
implies (self.source_End.affiliation = #environmental and
self.destination_End.affiliation = #environmental)
```

c. The scope value of a link is the widest of the scope values of the two connected elements.

```
Link
if (self.source_End.scope = #public or
                self.destination_End.scope = #public) then
                        self.scope = #public
        else if (self.source_End.scope = #private and
                self.destination_End.scope = #private) then
                        self.scope = #private
                else
                        self.scope = #protected
                endif
endif

if (self.scope = #public) then
                (self.source_End.scope = #public or
                self.destination_End.scope = #public)
else if (self.scope = #protected) then
                        (self.source_End.scope <> #public and
                        self.destination_End.scope <> #public and
                        (self.source_End.scope = #protected or
                                self.destination_End.scope =
                                #protected))
                else
                        (self.source_End.scope = #private and
                        self.destination_End.scope = #private)
                endif
endif
```

Appendix E. OPM Metamodel Constraints in OCL

(1) A result link cannot represent a condition that enables a process execution.

```
Procedural_Link
self.type = #resulting implies self.condition = #no
```

(2) An event link cannot be a result link neither represent a condition.

```
Event_Link
self.type <> #resulting
self.condition = #no
```

(3) In-Zooming of an entity instance makes its in-zoomed element instances visible, while out-zooming makes them un-visible.

```
Entity_Instance
(self.oclIsTypeOf(state_Instance)) implies
self.zooms_into -> forAll (ei | ei.oclIsTypeOf(state_Instance)
Entity_Instance :: In-Zooming()
post: result = self.zooms_into -> forAll (ei | ei = #visible)
Entity_Instance :: Out-Zooming()
post: result = self.zooms_into -> forAll (ei | ei = #invisible)
```

(4) Unfolding of a thing instance makes its unfolded element instances visible, while folding makes them un-visible.

```
Thing_Instance :: Unfolding()
post: result = self.unfolds_into -> forAll (ei | ei = #visible)
Thing_Instance :: Folding()
post: result = self.unfolds_into -> forAll (ei | ei = #invisible)
```

(5) State expressing of an object instance makes its state instances visible, while state suppressing makes them un-visible.

```
Object_Instance :: State Expressing()
post: result = self.owns -> forAll (s | s = #visible)
Object :: State Suppressing()
post: result = self.owns -> forAll (s | s = #invisible)
```

(6) Two entities can be connected via a single type of procedural link within an OPM view.

```
View
// Defining useful functions
EntityInstanceSetInCurrentView(e)≡
```

```
self.entity_Instance->select(is_an_appearance_of_an=e)
ConnectingProceduralLinkInstances(ei1, ei2)≡
self.procedural_Link_Instance-> select((source_End=ei1 ∧
destination_End=ei2)∨(source_End=ei2 ∧ destination_End= ei1))
// The constraint
self.entity_Instance.is_an_appearance_of_an -> forAll(e1, e2|
ConnectingProceduralLinkInstances(EntityInstanceSetInCurrentView(e1),
EntityInstanceSetInCurrentView(e2)) -> forAll (pli1, pli2 | pli1.
is_an_appearance_of_a.type = pli2. is_an_appearance_of_a.type))
```

(7) The direct link between two entities is the most abstract link between their refineables. If this link cannot connect the relevant entities according to the metamodel constraints, no link will be shown in the more abstract level.

```
OPM_Component::Consistency_Checking(): Boolean
// Defining a useful function
SingleConsistencyChecking(ei1, ei2, pli)≡ ((pli.source_End=ei1 ∧
pli.destination_End=ei2) implies
(self.Procedural_Link_Instance-> select(source_End ∈ ei1.zooms_into ∪
ei1.unfolds_into ∪ ei1.owns ∧
destination_End ∈ ei2.zooms_into∪ei2.unfolds_into∪ei2.owns)->forAll (pli2 |
pli.isMoreAbstractThen[12](pli2))))
// The constraint
self.OPM_View -> forAll (ei1, ei2: entity_Instance, pli: Procedural_Instance |
SingleConsistencyChecking(ei1, ei2, pli))
```

[12] `isMoreAbstractThen` is a function derived from Table 14.

Appendix F. OPCAT – Object-Process CASE Tool[13]

OPCAT (Object-Process CASE Tool) is an integrated system development software environment that supports system development using OPM. Being an OPM-based CASE tool, OPCAT enjoys the advantages of supporting most of the system development lifecycle tasks, starting from requirement analysis, through system design and implementation, to system testing, simulation, and validation. Since OPM enables modeling system dynamics and control structures, such as events, conditions, branching, and loops, the generated implementation can definitely be more advanced than a mere standard skeleton code. Moreover, OPM's ability to capture the system's structure and behavior in a single view also enhances OPCAT simulation capabilities and makes it most suitable for interactive testing and validation.

Case Tool Utilization

CASE tools have been developed with the objective of assisting developers in producing high quality software systems and products. To this end, CASE tools are designed to relieve the system architects and developers from mundane software engineering activities, leaving them more time to focus on the non-trivial, insight- and creativity-demanding tasks. Over the years, several studies have surveyed the way organizations use CASE tools. Lending and Chervany [55] found out that "it was difficult to find companies using CASE tools." Even in the companies that did use CASE tools, the extent of their deployment was very small. The CASE tool features that these companies employed were divided into two groups: analysis functionality (e.g., testing for consistency between a process model and a data model), and

[13] A version of this appendix was published in the International Conference of Enterprise Information Systems (ICEIS'2003) [25].

transformation functionality (e.g., generating executable code in several languages). The overall result for using features from both functionalities was low.

McMurtrey et. al. [61] surveyed the use of CASE technology inquiring professionals from different company types (insurance, manufacturing, consulting firms, etc.). They focused on the most popular features that CASE tools possessed and the gap between them and the developer needs. The features most often cited as being needed and used were the ability to represent a design in terms of data models and process or flow models. This reflects the fact that representing the model's structure and behavior aspects is the most useful aspect of current CASE tools.

In order to overcome some of the CASE tools flexibility drawbacks, a new type of CASE tools, called meta-CASE tools, or Computer Aided Method Engineering (CAME) tools, was introduced. These tools feature flexible metamodeling facilities that users can reconfigure to support whatever metamodel they wish to deploy. Examples of such tools are MetaEdit+ [94] and AToM^3 [54]. However, these tools are not widely used, since employing them is not a trivial task.

Method engineering and CASE tool designers have paid little, if any, attention to the human cognition theory. For example, the cognitive theory has shown that the human information processing system involves separate channels for processing visual and verbal material, and that the processing capability of each channel is quite limited [60]. The existing CASE tools address only the visual channel, neglecting the verbal one.

OPCAT Overview

OPCAT[14] has been developed as an academic project. It is designed to support the entire system development lifecycle through OPM. The two main benefits of OPCAT over existing

[14] OPCAT can be download for free from http://www.objectprocess.org/

object-oriented CASE tools are its bimodal graphic-textual single view representation and its simulation capability. The bimodal representation of OPCAT increases OPM accessibility to heterogeneously skilled users engaged in the system development process. The intuitive, bimodal model representation enables development teams consisting of system architects and domain experts to jointly engage in the development process on an ongoing basis. Such collaboration, which is not feasible with other CASE tools, is highly desirable, because it enables requirements to be put to test while modeling. Improving the system documentation quality is yet another benefit of OPCAT's bimodal representation, since the textual representation provides the system documentation.

OPCAT's simulation capability enables “running” a system model, testing its functionality against the requirement specifications, and debugging them at the model level, prior to the beginning of the implementation phase. OPCAT capabilities are demonstrated through a case study of a travel management system. The system manages company employee professional travels, including travel request, approval, finance and expense reporting.

The Bimodal Graphic-Text Representation

Catering to the modality principle of cognitive theory, OPCAT enables modeling systems graphically via an OPD-set and textually using OPL, a subset of English. OPCAT automatically translates an OPD-set into its equivalent OPL paragraph and vice versa. This way, users who are not familiar with the graphic notation of OPM can validate their specifications by inspecting the OPL sentences, which are automatically generated on the fly in response to the user's graphic input. Another cognitive principle – the limited channel capacity – is addressed in OPM through the abstraction/refinement mechanisms. These provide for creating diagrams and corresponding OPL paragraphs that are limited in size, thereby avoiding information overload and enabling comfortable human processing. The relatively small set of OPD symbols and corresponding OPL sentence types increases the

accessibility of OPM to both system architects and domain experts. The automatic translation into an OPL script also improves the documentation quality of the developed system. The automatic implementation generation, currently under development, will ensure that the specification designed by the system architects and endorsed by the domain experts is indeed reflected without any translational gap in the actual system.

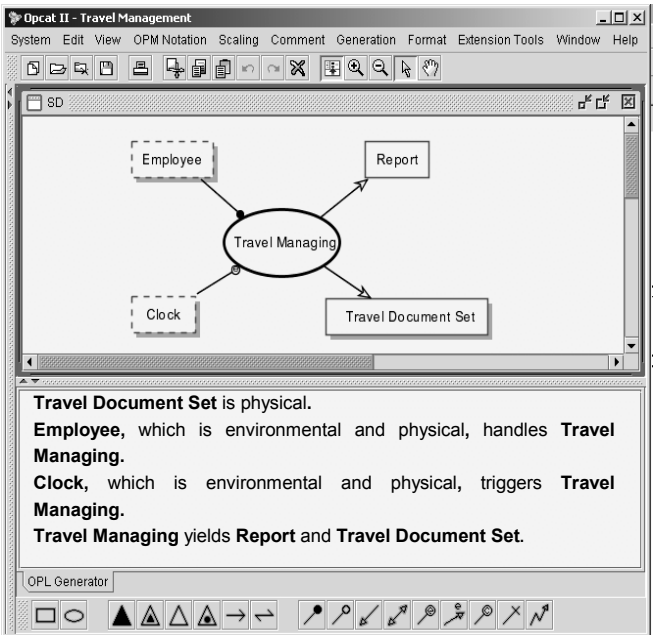


Figure 101. The OPCAT GUI showing the top-level specification of the travel management system

Figure 101, which is a snapshot of an OPCAT 2 screen, shows the bimodal OPD-OPL representation for the travel management system. The graphic window in the upper part of the screen shows the top-level OPD, while the lower part of the screen is the text window, which contains the equivalent OPL paragraph. Interpreting the OPD or the OPL paragraph, the model specifies that the **Travel Managing** process is handled by the **Employee**, which is an environmental (dashed) and physical (shadowed) object, linked to **Travel Managing** via an agent link. The corresponding OPL sentence that expresses this is: "**Employee,** which is environmental and physical, handles **Travel Managing.**" **Travel Managing** is also triggered by the physical and environmental **Clock**, which generates external timing events. **Report** and **Travel**

Document Set (which, as the shadow denotes, is physical) are the artifacts resulting as objects from the execution of **Travel Managing**.

OPCAT 2 performs extensive syntax checking, starting from the validation of simple constraints, such as checking that two objects are not linked via a procedural link, continuing with complex constrains, such as disallowing a loop within a generalization-specialization hierarchy in an OPD, and ending with inter-OPD consistency checking operations.

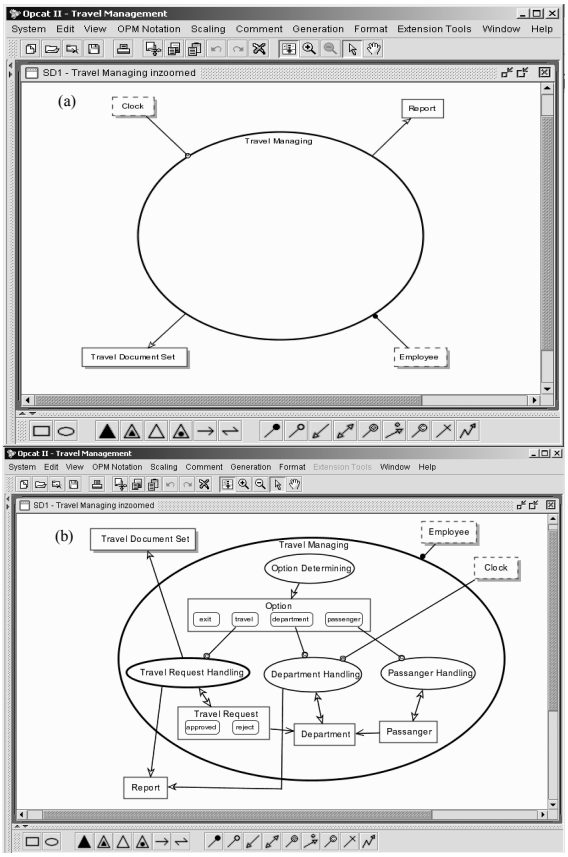


Figure 102. (a) The new OPD, created in response to the user's in-zooming operation on the **Travel Managing** process. (b) The OPD after the user has filled in details within the in-zoomed **Travel Managing** process.

To specify the details of the behavior of **Travel Managing**, the developer can use the in-zooming refinement mechanism. Applying this refinement on **Travel Managing** in Figure 101 yields a new OPD shown in Figure 102(a), titled "Travel Managing in-zoomed." The in-zoomed **Travel Managing** process appears enlarged in the center of the newly generated

diagram. All the entities connected to **Travel Managing** in the top level specification are also connected to it in the new OPD with the same link types.

The developer can now specify the sub-processes of **Travel Managing** and any pertinent interim objects within its elliptical frame, as shown in Figure 102(b). This refinement specifies that through the **Option Determining** sub-process, the **Employee** first chooses between the **Option** states: **travel**, **department**, **passenger**, and **exit**. This selection is a condition to the occurrence of the appropriate process, which can be one of **Travel Requesting**, **Department Handling**, **Passenger Handling**, or nothing.

Simulation and Dynamic System Testing

Being both object- and process-oriented, OPM enables designing the structural and behavioral aspects of a system in the same model. The fact that these two central system aspects are represented in the same diagram type, OPD, which is the only OPM graphic view, enables OPCAT to visually simulate the behavior of the system being developed. This is an important capability, as it enables system architects to dynamically examine the system at any stage of its development and verify with the domain experts that it addresses the client requirements and expectations. Presenting live animated demonstrations of system behavior, instead of static, printed models, enhances the communication between system architects and clients. This dynamic testing capability enhances the static testing option through examining the OPD set and comparing it to the corresponding OPL script.

Both static and dynamic testing can be carried out on an ongoing basis at any point in time along the system analysis and design processes to detect discrepancies, inconsistencies, and deviations from the intended goal of the system. As part of the dynamic testing, the simulation enables designers to track each of the system scenarios (also known in UML terminology as use cases) before writing a single line of code that implements the modeled behavior. Any detected mistakes or omissions are corrected or added at the model level,

saving costly time and efforts that would be needed to do so at the implementation level. Avoiding and eliminating design errors as early as possible in the system development process and keeping the documentation up-to-date contribute to shortening the system's delivery time ("time-to-market").

Although some UML supporting CASE tools provide simulation tools, the lack of a single clean formalism for expressing processes in UML and the fact that the dynamic views are separate from the static ones, makes such simulations much less comprehensible, as they can only run on a subset of the nine UML diagram types (sequence diagrams, state machines, or collaboration diagrams), overwhelming the limited human cognitive channels and making it extremely difficult to grasp the behavior of the system in its entirety.

OPCAT simulation is performed graphically on the model itself. It is affected by following parameters: process duration, step duration, and reaction time, where each of them can be specified as a fixed number of time units or as some probability distribution function with the pertinent parameters. Such non-deterministic duration definitions enable the simulation of real-life situations as in discrete event simulation systems.

After determining these parameters, the designer may manually activate entities, in particular enabling objects (agents and/or instruments) that are connected to system processes via external event links. By default, all the objects that are not created by processes in the model are defined as active, but the user can override this default. For example, simulating the travel management system behavior, in the initial situation only **Employee** and **Clock** are active (grayed). These objects existed before the **Travel Managing** process started. **Travel Managing** itself is not active, because no (internal or external) event has triggered it yet. The user can then start, stop, pause, and continue the simulation. The user can also set up breakpoints within the system model, run the simulation forward or backward any specified number of steps, or track it step by step. At each point in time during the simulation, all the

active OPDs, i.e., all the diagrams that contain active processes, are tiled and shown simultaneously to the user on the same screen.

The simulation algorithm determines the next step according to process activation rules derived from the OPM semantics. The guidelines of these rules are as follows. A process becomes active when its containing process is active and its turn activation time is due according to the time progression along the vertical time axis in the OPD (from top to bottom). A process becomes active when an event link connected to it is active. A process is executed if its pre-condition set holds. After executing a process, its post-condition set holds.

Continuing with the simulation of the travel management system, in order to invoke the **Travel Managing** process, the user has to activate the agent link from **Employee** to **Travel Managing**, simulating the employee action in the real system. As a result of this activation, **Travel Managing** and two of its internal objects, **Department** and **Passenger**, which already exist and are not generated by the system, become active. Note that **Option** does not become active yet, since it is created by the system in this scenario. When the in-zoomed **Travel Managing** process becomes active, it makes its first sub-process, **Option Determining**, active too for an interval of time determined by the process duration parameter. When **Option Determining** terminates, **Option** becomes active and **Option Determining** reverts to be non-active, as shown in Figure 103. Since **Option** has no initial or default states, the simulation must wait for the user to determine (select) the state of **Option**. The user simulates the **Employee** choice in the real system by manually activating one of **Option** states. This way, designers can selectively simulate use cases within the modeled system. Assuming that the designer activated **travel**, the simulation can continue to the next step, which is to activate the **Travel Requesting** process. The simulation algorithm now examines the pre-conditions of this process: **Option** needs to be in its **travel** state. Since this pre-condition set holds, **Travel Requesting** executes, creating **Travel**

Request, **Travel Document Set**, and **Report**. Since this is the last sub-process of the **Travel Managing** super-process, **Travel Managing** terminates, ending the simulation as depicted in Figure 104.

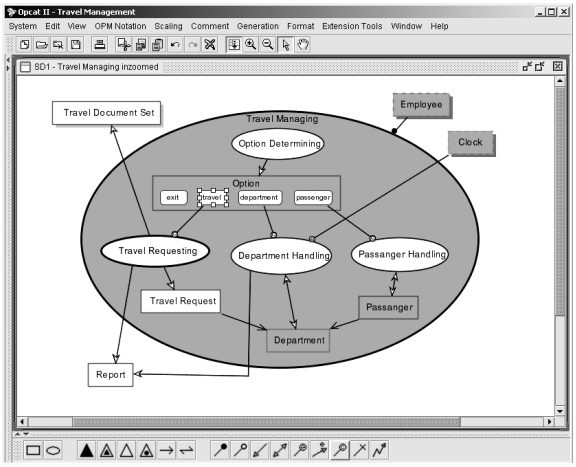


Figure 103. The situation of the travel management system after the **Option Determining** sub-process of **Travel Managing** has terminated and **Option** was generated

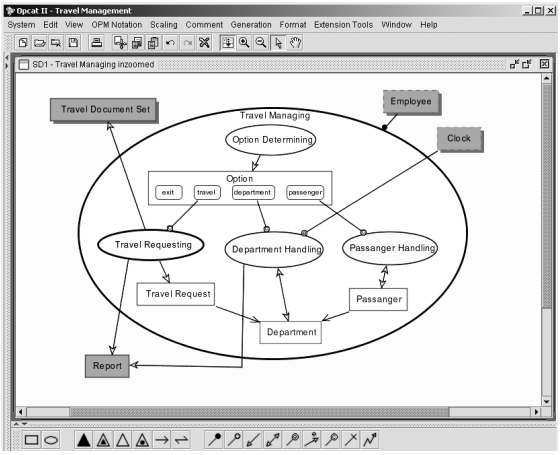


Figure 104. The final situation of the travel management system after running the simulation

If **Travel Managing** is triggered again, the number (multiplicity) of **Travel Document Set** and **Report** would increase, denoting that the same class has several instances created during consecutive executions of **Travel Managing**.

Summary and Work in progress

OPCAT, an OPM-supporting CASE tool, is an integrated software and system development environment that exhibits a number of unique features. OPCAT implements two important cognitive theory principles, the modality principle and the limited channel capacity principle. To implement the modal principle, OPCAT provides a dual, graphic and textual model

representation. The human limited channel capacity is addressed by implementing the various abstraction/refinement mechanisms OPM offers.

Designed to support OPM, OPCAT uses the universal OPM ontology, specified in Part 4, to provide the basic elements – objects, processes, states, and structural and procedural links – required to model a wide range of system types, including structure-oriented, process-oriented, and reactive systems. These elements are the building block required to provide the enhanced expressiveness developers seek. Hence, OPCAT addresses the expressiveness needs that cause the arising of the meta-CASE tools.

Another major benefit of OPCAT, which is made possible due to OPM's coherent single model representation, is its advanced simulation capabilities. Simulation helps visualize the operation of the system at any level of detail, providing an additional tool for early error detection and correction.

OPCAT has been studied and used in an undergraduate system analysis course for the past two years. Students' responses to OPCAT are enthusiastic. They indicate its reliability, user friendliness, ease of use, accessibility to untrained users, and inspiring simulation capabilities.

OPCAT is currently undergoing major expansion. The following work in progress is underway.

- Analysis and design document generation: OPCAT document generator facilitates selective generation of general information, OPDs, OPL paragraphs, and element dictionary. The documents are produced in an HTML format according to user-defined templates that provide flexibility of the resulting artifacts.
- Implementation generation: The implementation generator is designed to support conversion rules to various target programming languages and databases. Using OPL grammar, system developers will have to define once the translation rules from each OPL

template to a specific language, and the implementation generator will automatically generate the system from its OPL script. Since OPM describes also the behavioral aspects of systems, the generated implementation will not be just a skeleton code.

- OPM-to-UML conversion: Since UML is the standard modeling method within the software engineering community, a conversion utility from OPM to UML is being developed. OPCAT generates uses case, class, sequence, Statecharts, activity, and deployment diagrams from the single OPM model using XML Metadata Interchange (XMI) [73] standard. The inverse UML-to-OPM translation direction is also planned to be incorporated as part of OPCAT. This bi-directional conversion is an important utility, as it enables system developers to enjoy the integrity and user friendliness of OPM and OPCAT while not having to worry about not sticking with the industry standard; transformation to (and soon also from) UML is always available.
- Future features, aimed at further enhancing the usability of OPCAT, include support of automatic layout, a requirement management module, automatic test case generation, configuration management, intelligent knowledge base querying, weaving and merging mechanisms for integrating several OPM models to a single system, and a multi-user version, which enables collaboration of project teams.

References

1 Allen, R., R. Douence and D. Garlan, Specifying and Analyzing Dynamic Software Architectures, In Fundamental Approaches to Software Engineering, volume 1382 of Lecture Notes in Computer Science, E. Astesiano, Ed., Springer-Verlag, Lisbon, Portugal, 1998, pp. 21-37.

2 AOSD, The Aspect-Oriented Software Development site, http://aosd.net/

3 AspectJ Web Site. http://www.eclipse.org/aspectj/

4 Back, R.J.R. and Kurki-Suonio, R. Decentralization of Process Nets with Centralized Control. Distributed Computers, 3, 1989, pp. 73-87.

5 Barber, K.S., Graser, T.J. and Jernigan, S. R. Increasing Opportunities for Reuse through Tool and Methodology Support for Enterprise-wide Requirements Reuse and Evolution. Proc. of the 1st International Conference on Enterprise Information Systems, 1999, pp. 383-390.

6 Baumeister, H., N. Koch and L. Mandel, Towards a UML Extension for Hypermedia Design, In Proceedings of the 2nd International Conference on the Unified Modeling Language- Beyond the Standard (UML'99), volume 1723 of Lecture Notes in Computer Science, R. France and B. Rumpe, Eds., Springer-Verlag, Fort Collins, CO, 1999, pp. 614-629.

7 Becker, U. D^{2}AL – A design-based aspect language for distribution control. Proc. of the European Conference on Object-Oriented Programming (ECOOP), 1998. http://trese.cs.utwente.nl/aop-ecoop98/papers/Becker.pdf

8 Bloomer, J. Power Programming with RPC. O'Reilly and Associates, 1992.

9 Booch, G. Object-Oriented Analysis and Design with Application. Benjamin/Cummings Publishing Company, Inc., 1994.

10 Bosch, J. Superimposition: A Component Adaptation Technique. Information and Software Technology, 41 (5), 1999, pp. 257-273.

11 Bouge, L. and Francez, N. A Compositional Approach to Superimposition. Proc. of ACM POPL88 Symposium, 1998, pp. 240-249.

12 Carzaniga, A., G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. Proceedings of the 1997 International Conference on Software Engineering, pp. 22-32, 1997.

13 Ceri, S., P. Fraternali and A. Bongio, Web Modeling Language (WebML): a modeling language for designing Web sites, In Proceedings of the 9th World Wide Web Conference (WWW9), Computer Networks, Amsterdam, the Netherlands, 2000, pp. 137-157.

14 Chakravarthy, S. and D. Mishra, SNOOP: An expressive Event Specification Language for Active Databases, Data and Knowledge Engineering journal 14, 1, 1994, 1-26.

15 Clarke, S. Extending standard UML with model composition semantics. Science of Computer Programming, Elsevier Science, 44 (1), 2002, pp. 71-100. http://www.cs.tcd.ie/people/Siobhan.Clarke/papers/SoCP2001.pdf

16 Clarke, S. and Walker, R.J. Composition Patterns: An Approach to Designing Reusable Aspects. Proceedings of the International Conference on Software Engineering, 2001, pp. 5-14.

17 Clark, T., Evans, A., and Kent, S. Engineering Modeling Languages: a Precise Meta-Modeling Approach. http://www.cs.york.ac.uk/puml/mmf/langeng.ps

18 Conallen, J., Building Web Applications with UML, First Edition, Addison-Wesley, Reading, MA, 1999.

19 Constantinides, C. A., Bader, A., and Elrad, T. An Aspect-Oriented Design Framework for Concurrent Systems. Proc. of the European Conference on Object-Oriented Programming (ECOOP), 1999, pp. 340-352.

20 Czarnecki, K., Eisenecker, U. W. and Steyaert, P. Beyond Object: Generative Programming. Proc. of the Aspect-Oriented Programming Workshop at ECOOP'97, 1997, pp. 1-8.

21 Dale, J. and D. DeRoure. A Mobile Agent Architecture to Support Distributed Resource Information Management. Proceedings of the International Workshop on the Virtual Multicomputer, 1997. http://www.mmrg.ecs.soton.ac.uk/publications/archive/dale1997b/vim97.pdf

22 Domínguez, E., Rubio, A.L., Zapata, M.A. Meta-modelling of Dynamic Aspects: The Noesis Approach. International Workshop on Model Engineering, ECOOP'2000, pp. 28-35, 2000.

23 Dori, D., Object-Process Methodology - A Holistic Systems Paradigm, Springer Verlag, Heidelberg, NY, 2002.

24 Dori, D. Why Significant UML Change Is Unlikely. Communications of the ACM, 45 (11), 2002, pp. 82-85.

25 D. Dori, I. Reinhartz-Berger, A. Sturm, OPCAT - A Bimodal CASE Tool for Object-Process Based System Development. Proceedings of IEEE/ACM 5th International Conference on Enterprise Information Systems (ICEIS 2003), pp. 286-291, 2003.

26 D. Dori, I. Reinhartz-Berger, An OPM-Based Metamodel of System Development Process, accepted to the International Conference on Conceptual Modeling ER'2003.

27 D'Souza, D. and Wills, A.C. Objects, Frameworks and Components with UML – The Catalysis Approach. Addison-Wesley, 1998.

28 Eckstein, S., Ahlbrecht, P. and Neumann, K. Increasing Reusability in Information Systems Development by Applying Generic Methods. Proc. of the 13th International Conference CAiSE'01, LNCS 2068, 2001, pp. 251-266.

29 Firstenberg, Y., Katz, S. and Shmueli O. An Object-Oriented Program Accelerator Using Impersonation, Technion Computer Science Department Technical Report, CS-2002-06, 2002.

30 Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and Mixins. Proceedings of ACM Symposium on Principles of Programming Languages, 1998, pp. 171-184

31 Foo, S., P. C. Leong, S. C. Hul, and S. Liu, Security Considerations in the Delivery of Web-Based Applications: a case study, Information Management and Computer Security 7, 1, 1999, 40-49.

32 Frakes, W. and Terry, C. Software Reuse: Metrics and Models. ACM Computing Surveys, 28 (2), 1996, pp. 415-435.

33 Franklin, S. and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages, Budapest, Hungary, Springer-Verlag, pp. 21-36, 1996.

34 Franklin, M. and S. Zdonik. Data In Your Face: Push Technology in Perspective. Proceedings of the ACM SIGMOD international conference on Management of Data, pp. 516-519, 1998, http://www.cs.berkeley.edu/~franklin/Papers/datainface.pdf

35 Fraternali, P., Tools and Approaches for Developing Data-Intensive Web Applications: A Survey, ACM Computing Surveys 31, 3, 1999, 227-263.

36 Fugetta, A., G. Picco, and G. Vigna. Understanding Code Mobility. IEEE Transactions on Software Engineering, 24, 5, 1998, pp. 342-361.

37 Gamma, E., Helm, R., Johnson, R. Vlissides, J.O. Design Patterns: Abstraction and Reuse of Object-Oriented Design. Proc. of the European Conference on Object-Oriented Programming (ECOOP), LNCS 707, 1993, pp. 406-431.

38 Garzotto, F. and L. Mainetti, HDM2: Extending the E-R approach to hypermedia application design, In Proceeding of the 12th International Conference on Entity Relationship Approach (ER'93), R. Elmasri, V. Kouramajian, and B. Thalheim, Eds., Dallas, TX, 1993, pp. 178-189.

39 Garzotto, F., P. Paolini, and D. Schwabe, HDM – A Model Based Approach to Hypertext Application Design, ACM Transactions on Information Systems 11, 1, 1993, 1-26.

40 Graham, I., Henderson-Sellers, B., and Younessi, H. The OPEN Process Specification. Addison-Wesley Inc., 1997.

41 Gray, R., D. Kotz, G. Cybenko, and D. Rus. Mobile Agent: Motivations and State-of-the-art Systems. In Bradshaw J. M. (Ed.), Handbook of Agent Technology, AAAI/MIT Press, 2000. ftp://ftp.cs.dartmouth.edu/TR/TR2000-365.ps.Z.

42 Grundy, J. Multi-Perspective Specification, Design and Implementation of Software Components using Aspects. International Journal of Software Engineering and Knowledge Engineering, 10 (6), 2000, pp. 713-734.

43 Halpin, T. and Bloesch, A. A Comparison of UML and ORM for Data Modeling. Proceedings of the third International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'98), 1998.

44 Harel, D., Statecharts: a Visual Formalism for Complex Systems, Science of Computer Programming, 8, 1987, 231-274.

45 Henderson-Sellers, B., OML: proposals to enhance UML, The Unified Modeling Language (UML'98): Beyond the Notation, volume 1618 of Lecture Notes in Computer Science, J. Bezivin and P.A. Muller, Eds., Springer-Verlag, Mulhouse, France, 1998, pp. 349-364.

46 Henderson-Sellers, B. and Bulthuis, A. Object-Oriented Metamethods, Springer Inc., 1998.

47 Hillegersberg, J.V., Kumar, K. and Welke, R.J. Using Metamodeling to Analyze the Fit of Object-Oriented Methods to Languages. Proceedings of the thirty first Hawaii International Conference on System Sciences (HICSS'98), 1998.

48 Isakowitz, T., E. A. Stohr, and P. Balasubramanian, RMM: A Methodology for Structured Hypermedia Design, Communication of the ACM 38, 8, 1995, 34-44.

49 Katz, S., A Superimposition Control Construct for Distributed Systems, ACM Transactions on Programming Languages and Systems 15, 2, 1993, 337-356.

50 Kersten, M. and G. C. Murphy, Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming, In Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99), ACM SIG-PLAN Notices, Denver, CO, 1999, pp. 340-352.

51 Kim, Y.G. and March, S.T. Comparing Data Modeling Formalisms for Representing and Validating Information Requirements. Communications of the ACM, 38 (6), 1995, pp. 103-115.

52 Klein, C., A. Rausch, M. Shiling, and Z. Wen. Extension of the Unified Modeling Language for Mobile Agents. On Siau, K. and Halpin, T. (Eds.), The Unified Modeling Language: Systems Analysis, Design and Development Issues, Idea Group Publishing Book, pp. 116-128, 2001. http://www4.in.tum.de/~rausch/publications/2001/MobileUML.pdf

53 Lange, D., An Object-Oriented Design Approach for Developing Hypermedia Information Systems, Journal of Organizational Computing, 6, 3, 1996, 269-293.

54 Lara, J. and Vangheluwe, H., 2002. Using AToM^3 as a Meta-CASE Tool. Proc. of the 4th Int. Conference On Enterprise Information Systems (ICEIS'2002),.

55 Lending , D. and Chervany N.L., 1998. The Use of CASE Tools. Proc. of the Conference on Computer Personnel Research, pp. 49-58.

56 Lester, N. G., Wilkie, F. G. and Bustard, D. W. Applying UML Extensions to Facilitate Software Reuse. The Unified Modeling Language (UML'98) - Beyond the Notation. LNCS 1618, 1998, pp. 393-405.

57 Lin, M. and B. Henderson-Sellers, Adapting the OPEN methodology for Web development, In Proceedings of the 6th Annual Conference of BCS Information Systems Methodology Specialist Group: Methodologies for Developing and Managing Emerging Technology Based Information Systems, A.T. Wood-Harper, N. Jayaratna and J.R.G. Woods, Eds., Springer-Verlag, Salford, UK, 1999, pp. 117-129.

58 Lowe, D. and B. Henderson-Sellers, Characteristics of Web Development Process, In Electronic Proceeding of the International Conference Advances in Infrastructure for Electronic Business, Science, and Education (SSGRR'2001), 2001, http://www.ssgrr.it/en/ssgrr2001/papers/David%20Lowe.pdf

59 Mapelsden, D., Hosking, J., and Grundy, J. Design Patterns Modelling and Instantiation using DPML. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), 2002. http://www.jrpit.flinders.edu.au/confpapers/CRPITV10Mapelsden.pdf

60 Mayer, R.E. Multimedia Learning. Cambridge University Press, New York, 2001.

61 McMurtrey, M. E., Teng, J.T.C., Grover, V., and Kher, H. V., 2000. Current utilization of CASE technology: lessons from the field. Industrial Management & Data Systems, 100 (1), pp. 22-30.

62 Mens, T., Lucas, C. and Steyaert, P. Giving Precise Semantics to Reuse and Evolution in UML. Proc. PSMT'98 Workshop on Precise Semantics for Modeling Techniques, 1998.

63 Mezini, M. and Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 1998, pp. 97-116.

64 Mili, H., Mili, F. and Mili, A. Reusing Software: Issues and Research Directions. IEEE transactions on Software Engineering, 21 (5), 1995, pp. 528-562.

65 Muscutariu, F. and M.P. Gervais. On the Modeling of Mobile Agent-Based Systems. http://www-scr.lip6.fr/homepages/Marie-Pierre.Gervais/MATA2001.pdf

66 Nielsen, J. Hypertext and Hypermedia: the Internet and Beyond. Academic Press, 1995.

67 Nielsen, J., User Interface Directions for the Web, Communications of the ACM, 42, 1, 1999, 65-72.

68 Object Management Group (OMG). UML 1.4 - UML Semantics. OMG document formal/01-09-73, http://cgi.omg.org/docs/formal/01-09-73.pdf

69 Object Management Group. The common object request broker: Architecture and specification. Technical Report Version 2.0, 1995, http://www.infosys.tuwien.ac.at/Research/Corba/OMG/cover.htm

70 Object Management Group (OMG). Meta Object Facility (MOF) Specification. OMG document formal/02-04-03, http://cgi.omg.org/docs/formal/02-04-03.pdf

71 Object Management Group (OMG). Extensible Markup Language (XML), http://www.w3.org/XML/

72 Object Management Group (OMG). Software Process Engineering Metamodel (SPEM), version 1.0, OMG document formal/02-11-14, http://www.omg.org/technology/documents/formal/spem.htm

73 Object Management Group (OMG). XML Metadata Interchange (XMI), version 1.2. http://www.omg.org/cgi-bin/doc?formal/2002-01-01

74 Odell, J., H.V.D. Parunak, and B. Bauer. Extending UML for Agents. In Wagner, G., Lesperance, Y., and Yu, Er. (Eds.), proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, Austin, TX, pp. 3-17, 2000.

75 OPEN, The OPEN Web site, http://www.open.org.au/

76 Otero, M.C. and Dolado, J.J. An Initial Experimental Assessment of the Dynamic Modeling in UML. Empirical Software Engineering, 7, 2002, 99. 27-47.

77 Peleg, M. and D. Dori, Extending the Object-Process Methodology to Handle Real-Time Systems, Journal of Object-Oriented Programming, 11, 8, 1999, 53-58.

78 Peleg, M. and D. Dori, The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods, IEEE Transaction on Software Engineering, 26, 8, 2000, 742-759, http://iew3.technion.ac.il:8080/Home/Users/dori/Model_Multiplicity_Paper.pdf

79 Perrault, D., A Study of Business Rules Concept for Web Application, Master Thesis, Faculty of Engineering, Politecnico di Milano, Milano, Italy, 1998.

80 Rational Software. Rational Unified Process for Systems Engineering – RUP SE1.1. A Rational Software White Paper, TP 165A, 5/02, 2001, http://www.rational.com/media/whitepapers/TP165.pdf

81 Reinhartz-Berger, I., Dori, D. , and Katz, S. Developing Web Applications with OPM/Web. workshop on Data Integration over the Web (DIWeb'01), CAiSE, 2001, pp. 47-61.

82 Reinhartz-Berger, I., Dori, D., and Katz, S. OPM/Web - Object-Process Methodology for Developing Web Applications. Annals on Software Engineering (ASE) - Special Issue on OO Web-based Software Engineering, 13, pp. 141-161, 2002.

83 Reinhartz-Berger, I., Dori, D. , and Katz, S. Modeling Code Mobility Paradigms in OPM/Web. The Israeli Workshop on Programming Languages & Development Environments, IBM, 2002. http://www.haifa.il.ibm.com/info/ple/papers/code.pdf

84 Reinhartz-Berger, I., Dori, D., and Katz, S. Open Reuse of Component Designs in OPM/Web. 26th annual international Computer Software and Applications Conference (COMPSAC'02), pp. 19-26, 2002.

85 Renaud, P. E. Introduction to Client/Server Systems: A Practical Guide for Systems Professionals. Wiley & Sons, 1993.

86 Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ, 1991.

87 Siau, K. and Q. Cao, Unified Modeling Language (UML) – A Complexity Analysis, Journal of Database Management 12, 1, 2001, 26-34.

88 Schwabe, D. and G. Rossi, Developing Hypermedia Applications using OOHDM, In Electronic Proceedings of the 1st Workshop on Hypermedia Development Processes, Methods and Models (Hypertext'98), ACM, Pittsburg, KS, 1998, http://heavenly.nj.nec.com/266278.html

89 Schwabe, D., G. Rossi, and S. Barbosa, Systematic Hypermedia Application Design with OOHDM, In Proceedings of the 7th ACM conference on Hypertext, ACM, Washington DC, WA, 1996, pp. 116 – 128.

90 Shoval, P. and Shiran, S. Entity-Relationship and Object-Oriented Data Modeling – an Experimental Comparison of Design Quality. Data & Knowledge Engineering, 21, 1997, pp. 297-315.

91 Siau, K. and Cao, Q. Unified Modeling Language (UML) – A Complexity Analysis. Journal of Database Management 12 (1), 2001, pp. 26-34.

92 Stamos, J. and G. Gifford. Remote Evaluation. ACM Transactions on Programming Languages and Systems, 12, 4, pp. 537-565, 1990.

93 Suzuke, J. and Y. Yamamoto, Extending UML with Aspects: Aspect Support in the Design Phase, In Proceedings of the 3rd Aspect-Oriented Programming (AOP) Workshop at the Europe Conference on Object-Oriented Programming (ECOOP'99), volume 1628 of Lecture Notes in Computer Science, R. Guerraoui, Ed., Springer-Verlag, Lisbon, Portugal, 1999, pp. 299-300.

94 Talvanen, J. P., 2002. Domain-Specific Modelling: Get your Products out 10 Times Faster. Real-Time & Embedded Computing Conference, http://www.metacase.com/papers/Domain-specific_modelling_10X_faster_than_UML.pdf

95 Van Gigch, J. P. System Design Modeling and Metamodeling. Plenum press, 1991.

96 Verheijen, G.M.A. and Van Bekkum, J. NIAM: An Information Analysis Method. In Olle et al. 1986, pp. 289–318.

97 Vilain, P., D. Schwabe and C. S. de Souza, A diagrammatic Tool for Representing User Interaction in UML, In Proceedings of the 3rd International Conference on the Unified Modeling Language- Advancing the Standard (UML'2000), volume 1939 of Lecture Notes in Computer Science, A. Evans, S. Kent and B. Selic, Eds., Springer-Verlag, York, UK, 2000, pp. 133-147.

98 W3C Consortium. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl

99 Warmer, J.B. and A. G. Kleppe, The Object Constraint Language: Precise Modeling with UML, First Edition, Addison-Wesley, Reading, MA, 1998.

100 What is metamodelling, and what is a metamodel good for? http://www.metamodel.com/