

Chapter 5

Fundamental Concepts of the Real World and Their Mapping into UML

5.1 Abstraction, Concept, Domain, Ontology, Model

5.1.1 Abstraction

Modeling is a complex task, an attempt to capture the intricacies of real-world situations, to describe the characteristics of real-world objects, their relationships, and the way objects communicate together to evolve. The modeling process is based on *abstractions* and *concepts*. An abstraction is a mental process of taking a thing, *material* (mobile phone) or *immaterial* (electromagnetic wave), *real* (person) or *abstract* (his emotional state), pruning all details that are not relevant for a *particular purpose*, naming it, giving it a short description to be able to manipulate and work with the *abstraction* (as a result) issued from this complexity reduction process. An abstraction is therefore both a process and a result of this process. The object or the idea that results from an abstraction cannot be a specific detailed thing (“Ferrari Car” cannot be an abstraction but “Car” can be an abstraction).

Another meaning of abstraction in engineering disciplines is *information hiding*. *Data abstraction* results in hiding data that are not required to be visible at the interface of an object or a component. This meaning of the term “abstraction” is captured as the process of making black boxes.

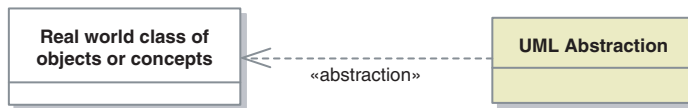
Another meaning of abstraction refers to the way models must be designed. At the early stage of the development process, it is necessary to abstract away implementation aspects and considering only on role that objects must play in a system. This process is called model abstraction. The MDA is based on this kind of abstraction.

The last meaning of abstraction refers to level abstraction that is central to managing complexity. Usual antonyms for “abstraction” are concretization, generalization, specialization, etc. according to its meaning variations. The following table summarizes all the meanings and their mappings in the UML.

Table 5.1.1.1 Mapping of Abstraction concept and different meanings in the domain of modeling

<i>Name of the concept</i>	<i>UML mapping</i>	<i>UML definition</i>
Abstraction	Relationship in Dependencies package	1. An abstraction is a <i>relationship</i> that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints.

1 UML Graphic Notation



For each object or concept in the real world, we build a UML abstraction to model this object or this concept. The Abstraction metaclass is standardized in the Dependencies package of the Kernel (Fig 7.15, Superstructure Book). The dependency relationship is oriented from the model element towards the element being modeled.

Other meanings in modeling	Comments
2. <i>Mental process for conceptualization or the product that results from this process</i>	An abstraction is a mental process of taking a thing, material or immaterial, real or abstract, pruning all details that are not relevant for a particular domain, naming it, giving it a short description to be able to manipulate it in a model. The result of this process is also called an abstraction (or sometimes a concept). An “abstract” of a scientific paper, which states the key ideas without details, is an abstraction of this paper.
3. <i>Data abstraction and information hiding</i>	Concept of black box. Irrelevant data, not pertinent to users, are hidden at the interface of an object/component. The result of the data abstraction and information hiding is also called an abstraction, e.g. a class or a component is the result of data abstraction.
4. <i>Model abstraction or design abstraction</i>	At any stage of the development process, a model must ignore all details of subsequent models. For instance, implementation details are not considered at design phase. The result of a model abstraction is an abstract model taken at any level of the MDA concept.
5. <i>Level abstraction</i>	The idea of level abstraction is central to managing complexity. At any level, a set of abstractions is used to describe a system. Each abstraction taken at any level may be decomposed to a new set of abstractions for the next level.

The definition of the *Abstraction* relationship in the UML is mapped to the first common meaning in Table 5.1.1.1. Data abstraction or information hiding is currently one of the most important features of object paradigm. The model abstraction or design abstraction is recently substituted by the newly coined

term MDA. The level abstraction governs all decomposition process and is central to managing complexity. We can add specific stereotyped dependencies for other meanings if necessary.

5.1.2 Concept

A *Concept* by its nature is abstract and a universal mental representation of things. An “abstract concept” is therefore a pleonasm. Knowledge is expressed in concepts and people use concepts in their daily lives. Many dictionaries give a general definition for concept as “An abstract or general idea inferred or derived from specific instances”. This general definition make the term “concept” very close to the result of the mental process defined for abstraction (first meaning in Table 5.1.1.1). Any concept needs, in the first stage, an abstraction (as process) that formats it, and then the abstraction (as result) is later enriched with properties, connected to remaining concepts of its domain. A concept is therefore a result of a process named *conceptualization* (different from “conception” that is equivalent to design) and *conceptual* is the resulted state of something that can be considered as building from a network of linked concepts.

So, if an abstraction is a low-level process (or a result of this process), *a concept is a more finished and ready to use*. If the concept is very elementary, its meaning is very close to the term “abstraction.” Any model element from the UML (*Class, Object, Event, Part, Component, Relationship, Association*, etc. are model concepts). All other meanings of the term “abstraction” (data abstraction and information hiding, model abstraction or MDA, level abstraction), except the first one (mental process for conceptualization) are therefore concepts. For instance, “data abstraction” is a concept.

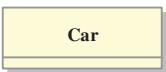
Concept mapping was developed in early 1960 by Novak (1990) and was a technique for representing knowledge in graphs (network of concepts). The result is a *concept map*. All concept maps are not necessarily concepts but specific concept maps compose new concepts. Concepts are useful in all human activities and are fundamental tools to communicate complex ideas and knowledge, to help learning, to generate new ideas (e.g. *Object, Classification*, and *MDA*), to design complex communication structures (e.g. long texts, hyper-media contents), to diagnose understanding or misunderstanding, etc. Concept maps are very close to semantic networks developed by Quillian (1968) for formal knowledge representation.

The UML does not define explicitly a metaclass Concept as for Abstraction. In fact, all model elements of the UML are *modeling concepts* (or metaconcepts) and what we try to create with the UML are, for a large part, *application concepts*.

Concept maps, semantic networks, and recent ontology engineering can be supported by the UML with class diagrams. Conceptual modeling is the process of identifying, analyzing, describing, and graphically representing concepts and

Table 5.1.2.1 Mapping of a simple concept that necessitates just a categorization process with a Class

Name of the concept	UML mapping	UML definition
Simple concept	Class	None



Example:
For each concept in the real world, a simple class (Element) or a composite class (Car) may be used for expressing this concept. The Concept metaclass is

not defined formally in UML but all model elements in UML are actually basic concepts.

Element that is the metaclass of UML is a concept. *Car* is also a concept, but *Car* is a user-defined composite class made of thousands of elementary classes.

constraints of a domain with a tool like the UML that is based on a small set of basic meta concepts. Ontological modeling is the process of capturing, describing relevant entities or categories of a domain (in an ontology of that domain) using an ontology specification language that is based on a set of basic ontological categories or entities. The UML may be used as a support for ontological modeling but it is necessary in this case to prepare this set of basic ontological categories or entities. So using the UML as ontological tool necessitates some preliminary works. Moreover, the UML addresses only the knowledge representation of ontology and there is no support for inference or exploitation of this ontology.

Complex concepts need more elaborate representations. For structural concepts, *structural diagrams* (class diagram containing some connected classes, component diagram, composite structure diagram, and object diagram) are sufficient, but concepts explaining system dynamics may require both structural and behavioral diagrams. In fact, a complex concept may require a complete model to explain it. When we model a system like a Car, it is a composite and a complex concept, which involves a hierarchy of concepts underneath.

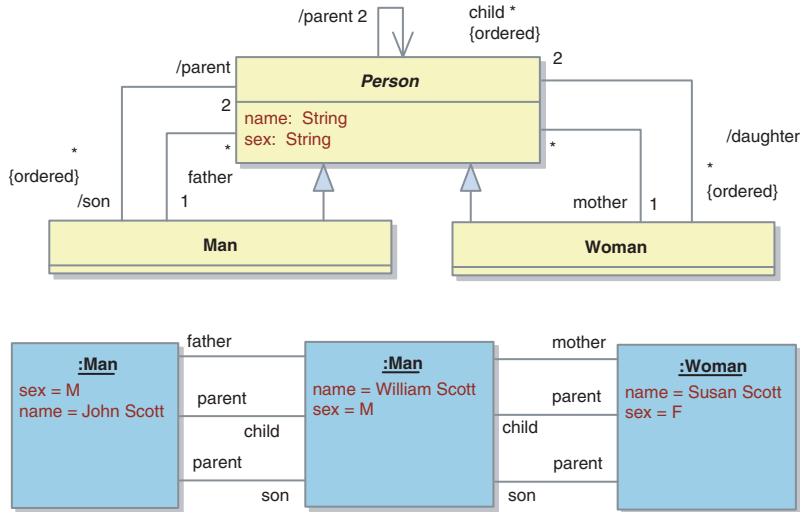
5.1.3 Domain

The term “domain” used in modeling means a field of study, an area of activity and knowledge characterized by a set of concepts and terminology understood by practitioners in this area. Physics, Mechanics, Electronics, Biology, etc. are examples of engineering domains. A domain may in turn be decomposed into subdomains. As the decomposition tree is rather large and the relative position is difficult to locate, we call any subdomain of any level simply a domain or a subdomain, and the prefix “sub” is contextual as a subdomain is simply a domain under the current domain taken as the reference level.

Table 5.1.2.2 Mapping of a “Complex Concept” concept and example

Name of the concept	UML mapping	UML definition
	Class, Component, Object, and Composite Structure Diagrams.	
Complex concept	Behavioral diagrams may be required to explain dynamic concepts (complex concept may join “model”)	None

Example of structural concept



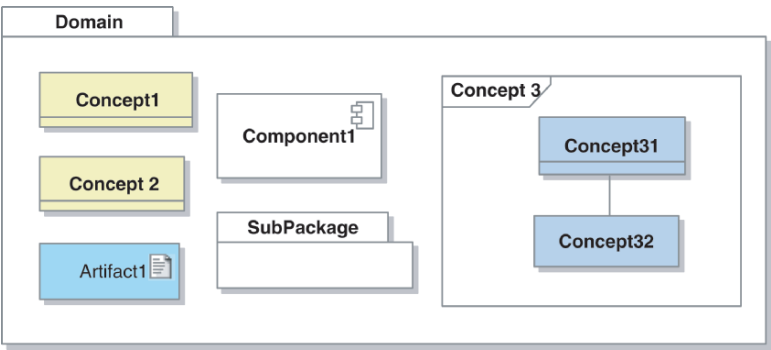
Class and object diagrams may be used to explain non elementary concepts. The concept of Family is built from three elementary concepts: Person, Man, and Woman, connected by a set of relationships. Person is an abstract class, both Man and Woman classes derive from the abstract class Person. Three objects, two instantiated from Man class and one instantiated from Woman class, together with the network of relationships, is a specific instance of the Family concept. In common language, the Class diagram represents the Family concept. The Object diagram represents a declarative knowledge.

In the domain of Computing, the ACM has classified various domains as: Literature, Hardware, Computer Organization, Software Engineering, Data, Theory of Computation, Mathematics of Computing, Information Technology, Methodologies, Computer Applications, and Computing Milieux. Subdomains of Methodologies are: Symbolic and Algebraic manipulation, AI, Computer Graphics, Image Processing and Computer Vision, Pattern Recognition, Simulation, Modeling and Visualization, Document and Text Processing, and Miscellaneous. So, the subject of this book is classified into the domain Simulation, Modeling, and Visualization, which is a subdomain of Methodologies.

Table 5.1.3.1 Mapping of a Domain concept in the UML

Name of the concept	UML mapping	UML definition
Domain	Package Diagram as a <i>Container</i> storing concepts (classes, components, artifacts, etc.). Tagged values may be added to give more precision about Package contents	None

Example:



Domain is a set of concepts, knowledge (instances), artifacts identified for describing a class of applications. Concepts are mapped into classes, class diagrams, components, etc. UML packages can be considered as containers for storing classes, objects, relationships, components, diagrams, artifacts, etc. necessary to give a full description of a domain.

Domain Engineering is the activity of collecting, classifying, organizing, and storing concepts in a particular domain as assets that can be reused when building new systems. Domain Engineering targets a product and thus needs Domain Analysis, Design, and Implementation as all other products.

To describe an application, a *General Linguistic Domain* that affords scientific vocabularies, constructs will accompany every description. As that package is *omnipresent in any description*, it is not mentioned in the description or is implicitly included. When describing a multidisciplinary system, many packages are required. For instance, to describe a robot that is an electromechanical device controlled by a microprocessor-based software, we need at least three domains represented graphically with three UML packages (Mechanics, Electrical Engineering, and Computer Science) and a General Linguistic Domain.

5.1.4 Ontology

A domain provides only a vocabulary for identifying, communicating, classifying concepts and knowledge about a domain. Ontology is a modern term with various meanings. Ontology is originally a term in philosophy that means “theory of existence. Gruber (1993) defines ontology as a specification of a

(domain) conceptualization. The web site www.owlseek.com provides an interesting definition of ontology:

We can never know reality in its purest form; we can only interpret it through our senses and experiences. Therefore, everyone has their own perspective of reality. An ontology is a formal specification of a perspective. If two people agree to use the same ontology when communicating, then there should be no ambiguity in the communication. So, an ontology codifies the semantics used to represent and reason with a body of knowledge. Ontologies can be written in various forms or languages.

If we fusion all elements of the two previous definitions with what a domain is, an ontology does not include only a set of concepts plus a set of knowledge (domain) but it structures this domain, has a representation of meaning and constrain interpretation of concepts. Moreover, with the development of ontological tools in the market, ontology provides a basic structure and a framework around which a knowledge base can be built. Knowledge is always updatable to insure that ontology always reflect realities, facts, knowledge, beliefs, goals, predictions, etc. of the moment of a given domain.

Ontology Engineering is a very important technological issue. Ontology tools are software programs that can be deployed through many platforms. They have syntactical verification tool to verify new entries. Semantic errors can be elevated to the rank of knowledge so ontology must have some kind of retroaction and metrics to verify its validity when confronted with real applications. Ontology clarifies the structure of knowledge.

When modeling a database of a university, we have students, professors, employees, females, males etc. but an employee may be student at part time. Further, ontological analysis show for instance that student or employee are not human categories like males or females but only roles.

So, ontology will clarify concepts and help reasoning. Second, ontology enables knowledge communication and sharing. The richer the ontology is in expressing meanings, the less the potential for ambiguities in creating requirements, in communicating knowledge. Today, ontology has grown beyond philosophy and has joined the information technology and AI.

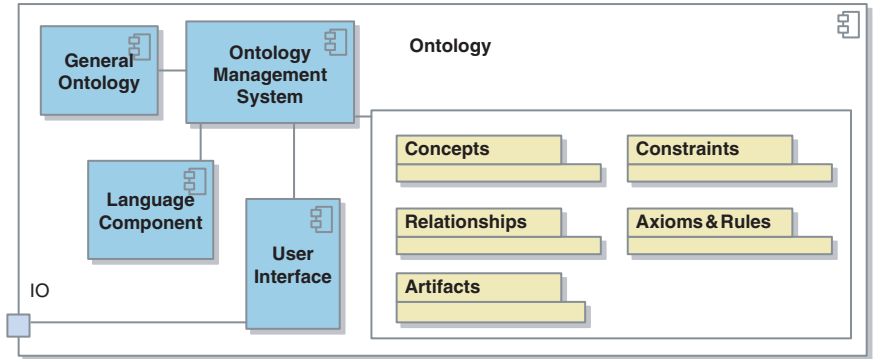
If we cannot give an exact definition of ontology, we notice that ontologies are targeted to be very helpful in the near future. They are planned to be used as:

1. *Common vocabulary* for practitioners inside specific domains (usage identical to that of a domain previously mentioned)
2. *Terminology standardization* (close to the previous point with an official standardization)
3. *Conceptual schemas* for building relational or object databases exploiting existing schemas and patterns (database usage)

Table 5.1.4.1 Mapping of an Ontology concept in the UML

Name of the concept	UML mapping	UML definition
Ontology	Component Diagram as a <i>Container</i> and all of its contents. Component Diagram in the version 2 of UML is more abstract and is not implementation oriented, so it can be used to store heterogeneous structures. A Package Diagram may be used within the Component Diagram at any level.	None

Example:



An ontology is first a domain but it is more than a collection of entities. Ontology is structured and incorporates semantics. It has properties, operations and is best represented by a Component Diagram. All kinds of diagrams or Packageable Elements can be inserted in an ontology.

4. For building and exploiting *knowledge bases*, answering competence questions
5. For supporting *model transformation* in the new MDA architecture, etc.

In fact, ontologies must assist us in every step in any engineering process needing a precise vocabulary, sharing existing knowledge, reusing available patterns, and artifacts. Tale 5.1.4.1 shows some possible mappings of ontology representation into the UML.

5.1.5 Model

In the software field, the perception of importance of model is increasing but traditional development, mainly code-centric, still predominates. Developers sketch their ideas with some diagram drafts but discard them quickly once the code has been implemented. Scientific disciplines use models to get a clear

Table 5.1.5.1 Mapping of a “model” concept into the UML

Name of the concept	UML mapping	UML definition
Model	<p>A <i>model</i> in its simplest form may be a <i>concept</i>. In this case, all mappings for the concept can be applied to a model. In its complex form, a model can be a whole system. So, if a model can be represented by a UML model elements and diagrams, then:</p> <p><i>All 13 diagrams of UML</i> may be used and combined at infinitum with artifacts to make any model</p>	<p>Omnipresent in the UML but there is no Model metaclass defined</p>

Examples:

- A model of a *Person at the requirement analysis* is simply a role and represented as a model element with a pictorial “stick man” taken from the use case diagram
- A model of a *Person in a database at design phase* can be represented with a class with structural attributes
- A model of a *Person in a real-time system at design phase* can be represented with a class with structural and behavioral attributes and operations
- A model of a *Person considered as a social agent* at the design phase can be represented by a class, a component, or more
- The UML itself is a metamodel and need diagrams and artifacts to describe it, etc.

understanding of what they developed to predict the behavior of their systems, to communicate ideas among developers or stakeholders.

For instance, an astrological model that makes statements on masses, positions, velocities of planets must be able to predict their positions x years after.

However, a model is not the reality but an abstract and close representation of this reality. If physical or mathematical models are based on formula or deterministic statements, most human models (medical, social models) give only some level of accuracy and their acceptance threshold fluctuates with space, time, and cultural standards. The validity of a model is its usefulness, its accuracy in predicting the future, and its reuse for similar problems. At the starting point, it must adhere to some standards of knowledge and rules to ease its communication and understanding. A model is therefore a container containing a rich set of artifacts, diagrams, etc. describing concepts, artifacts that describe and communicate the understanding of the “thing” been modeled.

A metamodel is a model to make models. It packs modeling elements, axioms and conventional rules that must be verified when establishing user models. The UML is a metamodel, defines model elements, and suggests graphical notations and a set of rules that must be verified for any valid UML models. As descriptions of the UML can be made using itself, the UML is a reflexive metamodel or

self-descriptive metamodel. This capacity of the UML is outstanding as it can represent a model that need several metalevels to describe it (UML has been broken into four levels M0–M3 in Chapter 4).

The classical definition of a model must be adapted to the new MDA paradigm that separates application knowledge from specific implementation technology. The consequence is an enhancement of the possibility of reuse of proven high-level patterns, an improved maintainability due to better separation of concerns, and better consistency due to a better direct mapping between domains and models.

5.2 Structural, Functional, and Dynamic Views of Systems

Communicating architectures to stakeholders are a matter of representing them in an unambiguous, readable form with information appropriate to them. As stakeholders may be of different disciplines, the communication task is not easy. Architecture is the only means to an end. Information that stakeholders can infer at the end is more valuable than information about just the architectures. The challenge would be how to select the information to be presented and how to select the sequence to maximize the understanding. A complex system would be impenetrable if all elements are exposed at the same time. The strategy is therefore multifaceted and follows the general concept of top-down design. But, this layering process or this tree view process itself is not sufficient. The level is still too complex to be handled as a whole. At each level, we must introduce only one facet of the system known as its selective or partial view. “A view is there fore a reduced set of coherent concepts used at any level to describe a system.” We can apprehend the view concept as a special lighting to bring out selective details. For instance, in medical context, X-ray machine tuned differently allows us to see different organs of a human body and ultrasonic probe complements some diagnoses. By combining all structural, functional, and dynamic views of any system, we force it to deliver all its secrets.

The UML has adopted solely two views to describe systems, Structure and Behavior, and melts functional and dynamic views inside a unique behavioral view. There is an interdependence of the functional and dynamic views and most diagrams are of combined nature as they mix these three aspects to some extend.

The class diagram is first classified as a structural diagram but the functional view appears clearly when class operations are detailed. Activity diagrams mix both functional concepts (actions, activities) to dynamic and evolutionary aspects (states, control flows). Use case diagram is clearly functional.

So, the description would be more precise if the behavioral aspect could be subdivided, at the application level, into two subdivisions, *functional*

(responsibility, potential capacity of actions, and interventions) and *dynamic* (evolutional aspects).

After half a century of functional modeling, designers generally agree that function is the most important concept in determining basic characteristics of a product or a system.

As a first reaction when approaching any system, we ignore the “How” and we want to know only if it “Can” or “Cannot” fit our needs (function availability).

Recently, the functional view is resuscitated by Business Processes that have emerged as an important aspect of enterprise computing landscape (B2B, Web Services, etc.). In the UML, the functional view is supported by main concepts as *Action*, *Activity*, and *Operation* metaclasses. There is no “Function” Metaclass and the term “function” is returned to the *General Linguistic Domain*. But, at the implementation level, this term reappears as fundamental programming concepts (C++, C#, or Java “functions”). Table 5.2.0.1 shows abstractions in three views and the impoverishment of the vocabulary passing from application domains towards the programming domain.

5.3 Concepts of the Functional View: Process and Business Process Modeling

The vocabulary of *process modeling* is rather rich: function, process, transformation, service, method, task, thread, action, activity, operation, datastore, dataflow, controlflow, etc. *Business process modeling* is a convergence of *process modeling* and *business modeling*. Process modeling is a general modeling domain and business modeling is an application domain that is business oriented. They have been melted to build a new concept “business process.” Terms as vision, mission, influence, assessment, goal, objective, strategy, tactic, policy, conformity, resource, task, etc. are part of business application domain. At the UML side, we have only four metaclasses representing functional aspects: *Action*, *Activity*, class *Operation*, and use case. The mapping will depend upon the interpretation given to new merged concepts of the BP model. Figure 5.3.0.1 illustrates this process with an UML-compatible diagram.

In this figure, the BP model is built from three domains: Process, Business, and General Linguistic. Once the concepts of the BP model is clearly defined with a proposed semantic, the Mapping activity takes each BP model concept, applies the mapping rules, finds the correct UML metaclasses and proposes a UML metaclass (or a component or a diagram pattern) that maps each input concept in the BP model into a corresponding output “UML compatible concept”. Most of the time, as this mapping process can be done manually and quickly, it is not interesting to automate it, but the mental work would be the same. We must be guided by a set of self compatible rules to insure that this mapping can work most of the time (if not anytime).

Table 5.2.0.1 Vocabulary depletion when evolving from application domains towards programming domain

	<i>Application Domains</i>		<i>UML Domain</i>	<i>Programming Domain</i>
Structural view	<i>– (Unlimited) description vocabulary for structural part of all systems –</i> System, problem, world, solution, software, data, database, data warehouse, data repository, knowledge base, object, class, agent, property, characteristic, variable, parameter, specification, rule, etc.		Class, association class, interface, object, datastore, component, package, node, artifact, part, composite structure, port, profile, stereotype, property, relationship, association, connector, dependency, namespace, multiplicity, role, expression, constraint, etc.	Class, object, attribute, variable, parameter, namespace, component, program, code, web service, etc.
Behavioral view	Functional View or Process view	Operation, function, service, procedure, action, task, process, business process, script, inference, transformation, etc.	Operation, use case, action, activity, event, condition, state, state machine, control flow, data flow, trigger, signal, message, time interval, time constraint, interaction, lifeline, etc.	Method, function
	Dynamic View	State, evolution, tendency, movement, evolution, collaboration, interaction, event, condition, interruption, time, etc.		Execution, execution path, thread, program state

5.3.1 Concepts used in Process Domain

Before mapping a modern application like BP model, let us start with the mapping of a more elementary Process Model.

This mapping will account for the mapping of all Process methodologies in the past to a unique UML-compatible Process Model. Some software companies still make use of Process methodologies (Stevens et al., 1974; Yourdon and Constantine, 1979; DeMarco, 1979; Gane and Sarson, 1978; Ward and Mellor, 1986) and it would be interesting to achieve such an exercise. Moreover, the exercise is interesting by several aspects:

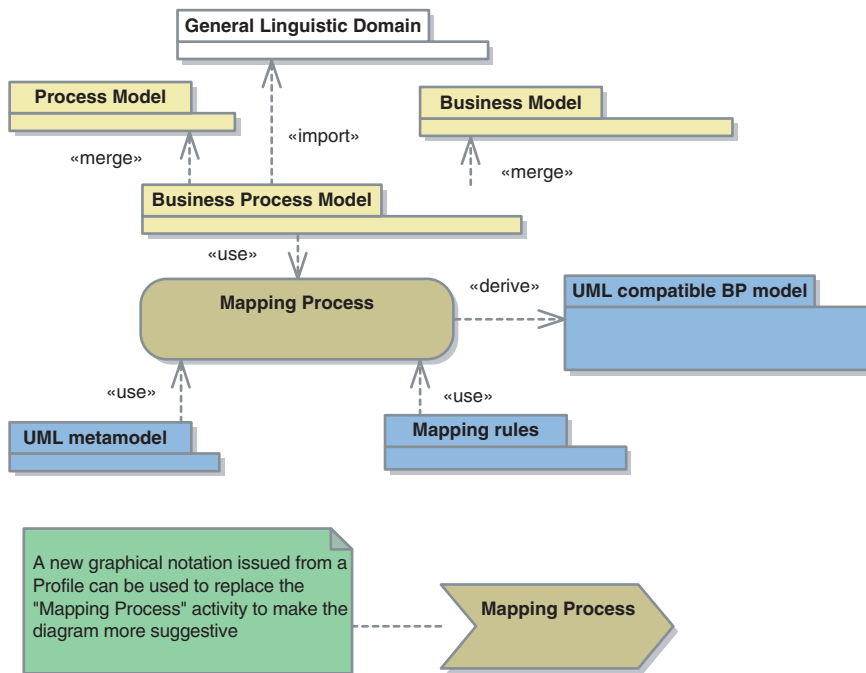


Fig. 5.3.0.1 Mapping of the Business Process Model into a UML compatible BP model. The Mapping Process BP is represented with a UML activity but could be replaced by a more suggestive graphical notation. All nodes are UML packages. All connections are stereotyped dependencies

1. Functional concepts are fundamental.
2. There is a *huge amount of projects* developed with older process methodologies, with DFDs, CFD (Control Flow Diagrams) that currently operate in industrial environment or in governmental organizations that *need*

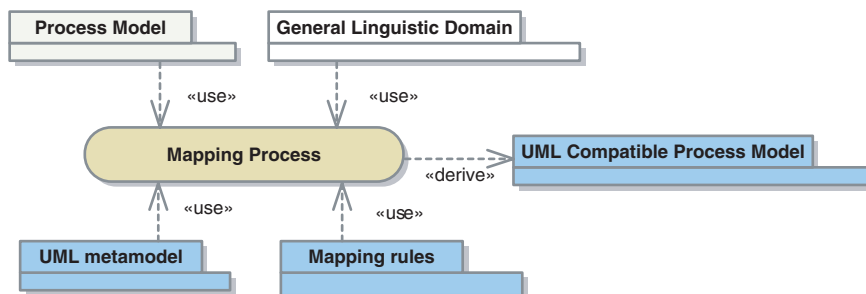


Fig. 5.3.1.1 Mapping of the Process Model directly into a UML compatible Process model

conversion, totally or partially, to the new object paradigm, only for maintenance or lightweight extensions.

3. Some companies involved in real-time systems still develop everyday systems with the CFD and DFD.
4. It would be a *bad idea to think that process aspects are less important* today than yesterday just by the presence of object technology, object methodologies, and the UML. The development methodology does not change the nature and the “process part” of a project. Object technology will help us best in organizing models, get a clean and well-decomposed system, and enhance our business approach.
5. *Business Process technology* has adopted the object view and the OMG has endorsed the idea and this win-win strategy will lead to better tools.

Hereafter, we start examining the fundamental concepts deployed with process technology, the DFD, and CFD in the past. That does not mean that we must keep all those concepts for new development, but this attitude would ensure a smooth evolution. If we take the web site Wikipedia (available at: <http://www.wikipedia.com>) as a reference and look for definitions of all functional terms (alphabetically classified), the following are the results with our comments:

Action: (Lack of computer-related definition, only in philosophy) certain kind of thing a person can do.

Action is a metaclass in UML and means a “fundamental unit of behavior specification” or “fundamental unit of executable functionality.”

Activity: Generally action

Circular definition is very common with elementary concepts in General Linguistic Domain. In UML, an activity specifies the “coordination of executions of subordinate behaviors”, using a control and data flow model. Activity is a metaclass defined in UML.

Function: (In computer science and depending on the context and programming language) a mathematical function, any subroutine or procedure, a subroutine that returns a value, a subroutine which has no side-effects

Leibniz first applied this word to mathematics around 1675, and according to Morris Kline, he is the first to use the phrase “function of x”. The term “function” is used to implement object “operation” (UML terminology) or object “method.”

Job: Piece of work or a task

In some version of UNIX, processes running under control of the shell are known as “jobs”. This terminology is potentially confusing, since the term job is more commonly used to refer to an entire session, i.e. to everything that a computer is doing. The old gang knows that in FORTRAN environment, everything was a “job” in IBM machines (compiler job, linker job, execution job, etc.). This term is now obsolete and has been returned to the General Linguistic Domain.

Method: (Science) codified series of steps taken to complete a certain task or to reach a certain objective

In object technology, a method is the processing that an object performs. When a message is sent to an object, the method is implemented to take into account the message and perform the desired action.

Operation: In its simplest meaning in mathematics and logic, an operation combines two values to produce a third (e.g. addition). Operations can involve mathematical objects other than numbers

Operation is a metaclass in UML and correspond the object “method” in object technology. Common meaning of “operation” is a “planned activity”. In Assembly language programming, operation is equivalent to “assembler instruction”.

Process: Naturally occurring or designed sequence of changes of properties/attributes of a system/object

A process normally comprises a number of activity steps. If not assimilated to its elementary “activity” component, a medium size process has a goal, needs resources, data, can be assisted by computers, humans to progress through those activity steps. This term has been retained to open a new era for workflow management study and business process automation.

Procedure: Subroutine or method (computer science), a portion of code within a larger program

The PASCAL programming language made use of this term to designate a subroutine or a portion of code equivalent to a C function block. Some authors assimilate procedure to “algorithm”. Wikipedia defines an algorithm as a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state. Notice the circular reference of “algorithm” to “procedure.”.

Service: (Lack of appropriate definition for computer science) economic activity that does not result in ownership

We can say that objects provide “services” through operations (UML level) or methods (implementation level). In large and distributed systems, it is useful to view the system as a set of services offered to clients in a service-based architecture. “Service” is used mainly in Client/Server architecture (Web Services is specific services in Web environment). In any system, if an object provides an operation, owns this operation, and executes this operation only though external object requests, we are in the context of client/server architecture and objects operations may be called “object services.”

Task: Part of a set of actions which accomplish a job, problem or assignment

In Operating System Domain, process and task are synonyms. Some authors make use simultaneously of three terms (process, task and thread) and distinguish between “a privileged task” and “a non privileged process” in an Operating System. Others consider only two concepts “processes and threads” as OS buzzwords and the term “task” is returned to the General Linguistic Domain.

Thread: Sequence of instructions which may execute in parallel with other threads

In Operating System Domain, a “thread” characterizes a lightweight process or simply a single path of execution through a program. If we refer to Microsoft terminology, a thread is the basic unit to which an operating system allocates processor time. Many threads can execute inside a process (e. g. multithreading).

In .Net Framework Developer’s Guide, “an operating system that supports preemptive multitasking creates the effect of simultaneous execution of multiple threads from multiple processes. It does this by dividing the available processor time among the threads that need it, allocating a processor time slice to each thread one after another. The currently executing thread is suspended when its time slice elapses, and another thread resumes running. When the system switches from one thread to another, it saves the thread context of the preempted thread and reloads the saved thread context of the next thread in the thread queue. The length of the time slice depends on the operating system and the processor. Because each time slice is small, multiple threads appear to be executing at the same time, even if there is only one processor. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors.”

Transformation: (No general definition in computing domain). A data transformation converts data from a source data format into destination data

This term was used in the past when dealing with “data transformation” in Dataflow Diagram. It loses now its importance and is returned to common vocabulary.

Processes and all process-like entities need *inputs*, *outputs*, *data flows*, *control flows* (connecting processes together), *data repositories* (acting as data sources and data sinks). To support this process view of system, all those entities should compose a minimum and coherent working set. We will clarify those notions later, for the moment, let us quote the minimum list to achieve a workable set of concepts for process modeling.

Datastore:

A datastore designates a storage element representing locations that store data, objects. The notion of datastore exists in UML under DataStoreNode defined in Activities Package. In UML, it is considered as “central buffer node for non transient information”. So, datastore can be understood as a source or a sink of non transient information.

Terminator:

This notion was introduced in the past in some process methodologies to represent external entities with which the current system communicates. It can be a person, a group of persons, an organization, another system, a computer, etc.

In UML, Terminator will be mapped, according to this definition to Actor metaclass in a use case diagram, to objects “external to the current system”. It would be interesting to notice that process methodologies admitted curiously in the past, the existence of “objects” outside their systems but not inside.

Data flow:

A data flow is a flow of data, information, objects between any combinations of datastore/process/terminator.

In Object Technology, data flow concept is replaced by object flow. In UML, there is no *Dataflow* metaclass but an *ObjectFlow* metaclass is defined in BasicActivities package and refined in CompleteActivities package.

Control Flow:

The notion of control flow has existed in some methodologies oriented to real time system development, e.g. SART (Structured Analysis Real Time). In UML, a metaclass *ControlFlow* has been defined in the BasicActivities package as an “edge that starts an activity node after the previous one is finished”.

5.3.2 Examples of Process Modeling and their Conversions into UML Diagrams

Without adding application difficulties to the mapping problem, we choose a classic “order processing” to show concepts used in Process modeling: processes, datastores, terminators (external entities), data flows, and control flows. We let in the background the correctness of the application logic itself and focus on the way the mapping is conducted to translate any process model into the UML model.

The Figure 5.3.2.1 shows a process diagram. Owing to the round circles used to represent processes, this kind of diagram was in the past, commonly called:

1. Bubble diagram
2. Bubble chart
3. DFD
4. Process model
5. Workflow diagram
6. Function model

Some methodologies (Gane and Sarson, 1978) make use of the same concepts but the round circle has been replaced by rectangles with rounded corners. So, graphical notations may vary from one author to another, but as long as the concepts are the same, all those diagrams are of the same type.

The terminator *Customer* is repeated twice but they are the same. This repetition avoids drawing complicated connections. Terminators are considered as external to the *Order Processing* system (so the contents of a Terminator cannot be changed). Dashed arrows represent control flows and regular arrows are data flows. Datastores or stores are named Repositories in this application and are

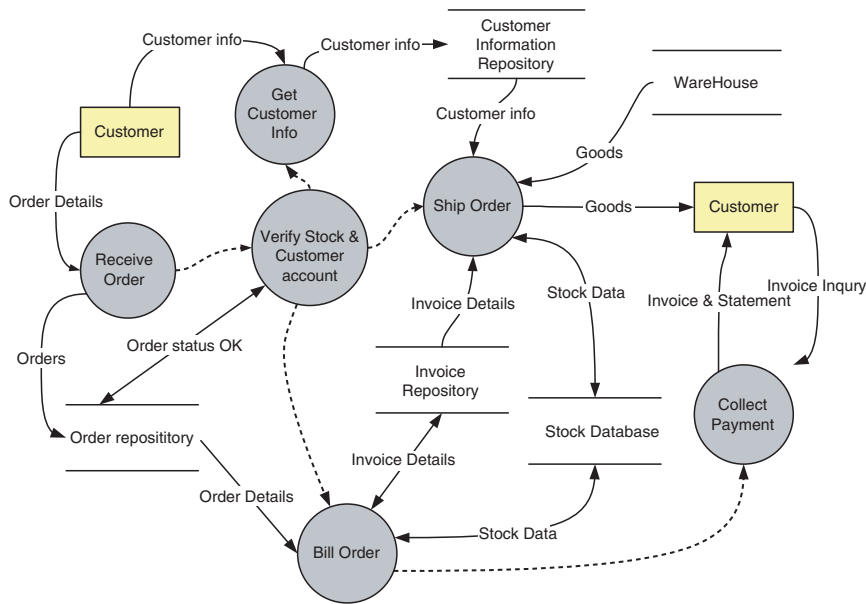


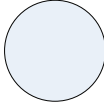

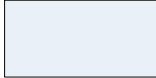
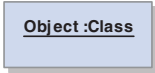

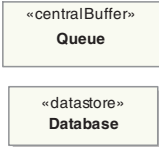

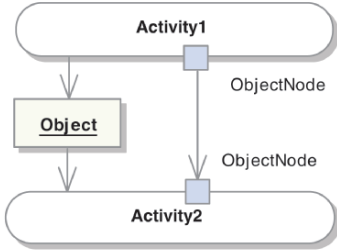
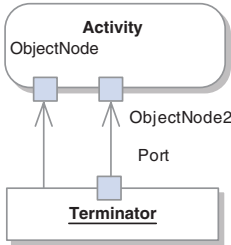
Fig. 5.3.2.1 Bubble process model of an Order Processing. Bubbles are processes, “Customer” is Terminator. Datastores are represented with two horizontal bars. Arrows are data flows and dashed arrows are control flows

represented by two horizontal bars. Datastores in this example depict an object repository (*Warehouse*) or a database (*Customer Information Repository*, *Order Repository*, *Invoice Repository*, and *Stock Database*). No precision is done at this step on the term *Repository* so a datastore may represent a database or a repository (metallic cabinets) for printed documents (orders and invoices).

The conversion from a DFD into an activity diagram is not straightforward and there exist some subtleties:

1. *Objects in an activity diagram are tied to object flows; they are not vertices but edges.* With the current version of the UML, the notion of “object” supported is object accompanying object flows. In this case, objects are attached to edges (connections) but not really to vertices (nodes), even if the diagram considers them as vertices. The activity diagram has also a notion of “partition” that identifies objects responsible of all activities located in this partition. Once more, partitions are not vertices but act as superposed layers on the activity diagram. So, to summarize, in the current activity diagram, we cannot draw an object freely as in an object diagram. For instance, we cannot illustrate that an object activates an activity. This difficulty can be patched for the moment with an object flow (edge) drawn directly between an object (vertex) and an activity

Table 5.3.2.1 Mapping rules of a simple Process Model into the UML

Process Model	UML model	Comment
Process 	Activity, Action 	<p>A Process can be mapped into Action or Activity in the UML.</p> <p>Process model does not have an equivalent structural view of systems, so Operation of Class has no equivalent. The same problem occurs for use case. Use Case metaclass is reserved for Requirements Analysis, but Process model does not distinguish development phases.</p>
Terminator 	Object 	<p>The Terminator can be mapped into an Object with eventually a defined classifier.</p>
Datastore 	CentralBuffer-Node, DataStoreNode 	<p>UML proposes two options: CentralBufferNode and DataStoreNode.</p> <p><i>Example:</i> A warehouse is better mapped into a CentralBufferNode and all databases into DataStoreNode.</p>
Data flow 		Dataflow <p>Data flows are replaced in Object context by object flows. An object flow can be differentiated from a control flow as it has at least at one side an object, an object node or an activity parameter node.</p>
Data flow between Process and Terminator		<p>The UML allows drawing an object flow between a Terminator object and an activity (not control flow). But the UML interprets that the Terminator itself as the object of the exchange. So, to avoid any ambiguity, object node, activity parameter node, and Port can be used to create data flows between Process and Terminator.</p>

(cont.)

Table 5.3.2.1 (Continued)

Process Model	UML model	Comment
<p>Control flow</p>		<p>Controlflow</p> <p>Control flows in the UML can be drawn freely between activities/actions. But we cannot draw a control flow between an object (playing the role of Terminator) and an activity. As a temporary solution, we can draw an object flow between an object and an activity and interpret this connection as a control issued from Terminator. A stereotype can be added and if necessary a name to avoid misinterpretation.</p>

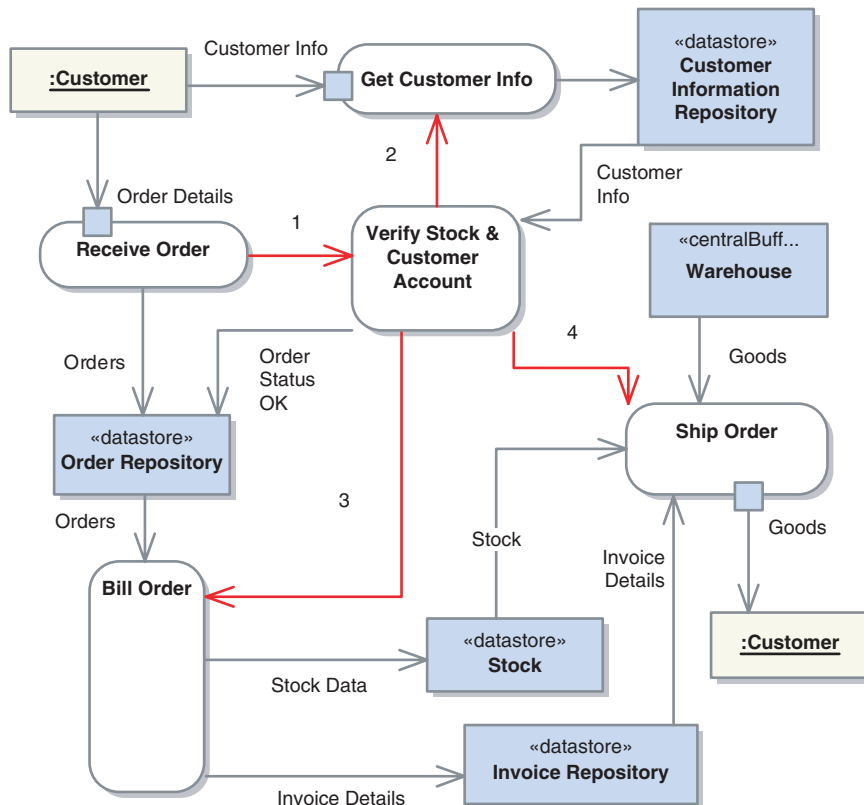


Fig. 5.3.2.2 Conversion of the DFD “Order Processing” into an activity diagram. Control flows are numbered to help interpreting good sequence order. Ports can be added to <<datastore>> and typified if needed

- (vertex) but interpreted as a control issued from the object. A stereotype and a name can be added to avoid misinterpretation (see last line of Table 5.3.2.1).
2. *Objects flows cannot be bidirectional.* Normally, when the process “Bill Order” can write to Stock (for instance to reserve some stocks for further delivery), we supposed that the write permission entails automatically the read permission. If we adopt this rule locally (by a constraint imposed to some diagrams or to the whole project), we don’t have to represent two object flow arrows, one in the read direction and one in the write direction. Object flows are directional in the current version of the UML and there is no bidirectional object flows.
 3. The *CentralBufferNode* concept is used to represent the Warehouse as goods entering a warehouse are supposed to leave it quickly. All databases are represented by *DatastoreNode* concept that can be used as a data sink as well. There must be some difficulties in grayed zone when data or objects are stored with intermediate duration. In this case, the best would be using a stereotype describing exactly the type of datastore we want to create.

5.3.3 Building a Proprietary Methodology

It is usual to see that big companies have in the past designed their proper methodologies, their proper graphical notations, and interpretation rules. This situation has in the past given rise to several thousands of methodologies and it would be a useless task to create an inventory for all of them. The question for those companies would be “is it possible to align smoothly towards the UML without having to convert existing designs?” Two extension mechanisms are already mentioned. With the UML, the second class extension can be done via “Profiles” (set up a UML Profile with proprietary names). The first class extension can be realized at the Infrastructure level but the risk of encountering incompatibilities using UML tools must be weighted. We suggest a third strategy named the “compatible strategy” described in Figure 5.3.3.1. This strategy lets the UML semantics intact and suggests another way of interpreting fundamental concepts with available graphical notations and connection rules.

5.3.4 Business Process Domain

The functional view of systems remains at the foreground scene contrarily to some prognostics of object theorists of the first wave. Processes become more structured and find their strength mainly in business models. To deal with business processes, three levels of intervention are separated in the way processes are approached in an enterprise environment:

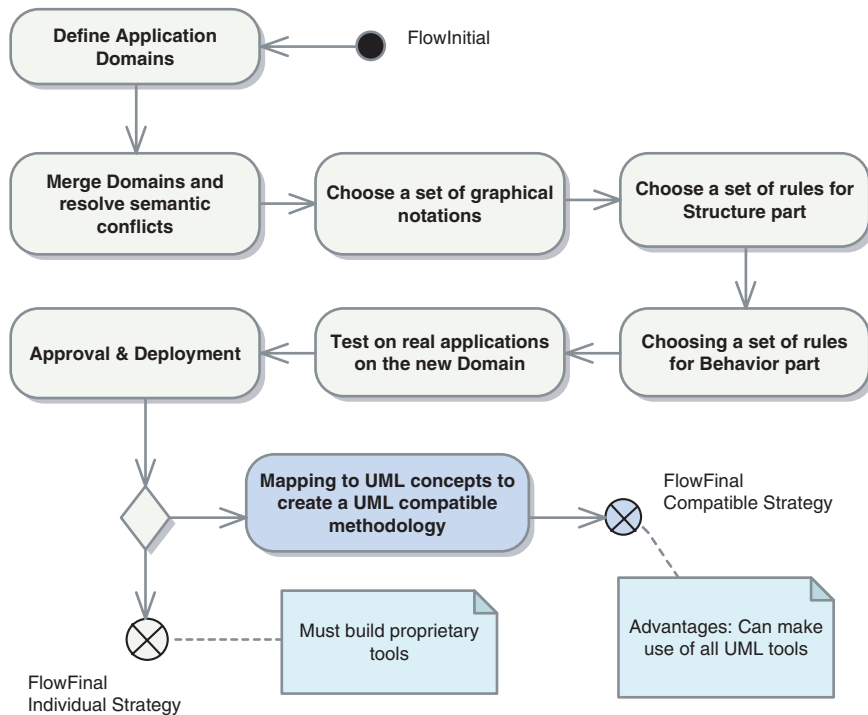


Fig. 5.3.3.1 Compatible strategy for adapting an existing methodology to the UML. While keeping the proprietary methodology, a mapping table can be created to convert concepts of the proprietary methodology into UML concepts (Profiles can be used in some steps)

1. The *operational level of business processes* that is the deployment of a designed process to manage daily operations with customers, suppliers, partners, etc. To deal with this level, a *Business Process Modeling Notation* (BPMN) has been standardized by the BPMI (available at: www.bpmi.org) and the document is currently available on the OMG site
2. The *conceptual level of business processes* known as *Business Process* or *Workflow Management*. This level specifies, describes, evaluates, uses metrics on workflows, optimizes, tries to automate parts of processes when possible, to adapt business processes to the moving environment (technology, competitor, supplier, customer, regulation, etc.)
3. The *metaconceptual level of business processes* identified by the OMG as the *Business Motivation Model* (BMM). This level takes into account higher-level concepts like organization missions, goals, objectives, strategies, tactics, policies, internal, and external influencers, organization assessments to influence the way business processes must be designed, taking into account rules at the workflow management level. The BMM

has been standardized and the document is currently available on the OMG site.

To explain the difference between these three levels, let us start with the first operational level of business processes.

A business process is a process in the business domain and a workflow is concerned with the information required to support each step of the business cycle. The BPMI has developed a standard BPMN and in June 2005, the BMPI and the OMG groups announce the merger of their BPM activities.

The new Business Modeling & Integration Domain Task Force mission is to develop specifications of integrated models to support management of an enterprise. These specifications will promote inter and intra-enterprise integration and collaboration of people, systems, processes, and information across the enterprise, including business partners and customers.

The final specification of the BPMN has been adopted in February 1, 2006 by the OMG and can be consulted freely on the OMG site [available at: www.omg.org]. This specification is out of scope of this book, but we are concerned with the mapping of business applications into the UML. A limited but principal set is illustrated. The rationale behind the BPMN comes from the observation that business people are very comfortable with visualizing business process in a flow-chart format instead of the activity diagram proposed in the UML.

BPD (Business Process Diagram) is a diagram designed for use by the people who design and manage business processes. The BPMN provides a formal mapping for BPEL4WS (Business Process Execution Language for Web Service) that is a standard copyrighted by BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems.

Fundamentally, the BPD is much closer to the UML activity diagram, if not nearly the same. Graphical notations are enriched and oriented towards business process applications to make the diagram more appealing to business people that are not familiar with the stripped esthetics of a technical diagram like the UML activity diagram. Some concepts come from metaclasses borrowed from the UML Interaction Suite or State Machine (e.g. Event).

In BPD, three concepts have been selected to represent functional concepts: *process*, *sub-process*, and *task*. Process, subprocess are all UML activities. A task can be mapped into the UML action or sometimes into activity. The concept of *Process* is *intrinsically hierarchical* in the BPMN. Processes may be defined at any level from enterprise-wide processes to that performed by a simple person. A Business Process is a “set of activities that are performed within an organization or across organizations.”

A task is reintroduced as an atomic activity that is included within a process. It is used when the work in the process is not broken down to a finer level of process model detail. This proposal of *task* for substituting to *elementary*

process avoids the same sound for different things inside a same domain. A *sequence flow* and a *message flow* replace the control flow and data flow in an activity diagram.

To support all the Business Suite, SBVR (Semantics of Business Vocabulary and Business Rules) is explained in a 390-page book that defines the technical vocabulary and rules for documenting the semantics of business vocabulary, business facts, and business rules. The proposal of this book is to formulate technical English in such a way that *quantification* (each, some, at most one, at most n , etc.), *logical operations* (p and q , p or q , p if q , it is not the case that p , etc.), *modal operations* (it is obligatory that p , it is impossible that p , must, must not, always, etc.), and specific keywords (the, a, an, a given, that, is of, etc.) could be recognized and semantics extracted for computation.

In Figure 5.3.4.1, we have two *Pools* (Supplier and Buyer). *Pool* is equivalent to *ActivityPartition* in an activity diagram that represents Participants in the order process. Small round circles at the left are Start Event (equivalent to *InitialNode* metaclass in UML) and the same small circles with a more accentuated line thickness are End Event (equivalent to *FlowFinalNode* metaclass). All rectangles with rounded corners are processes or tasks (tasks are elementary

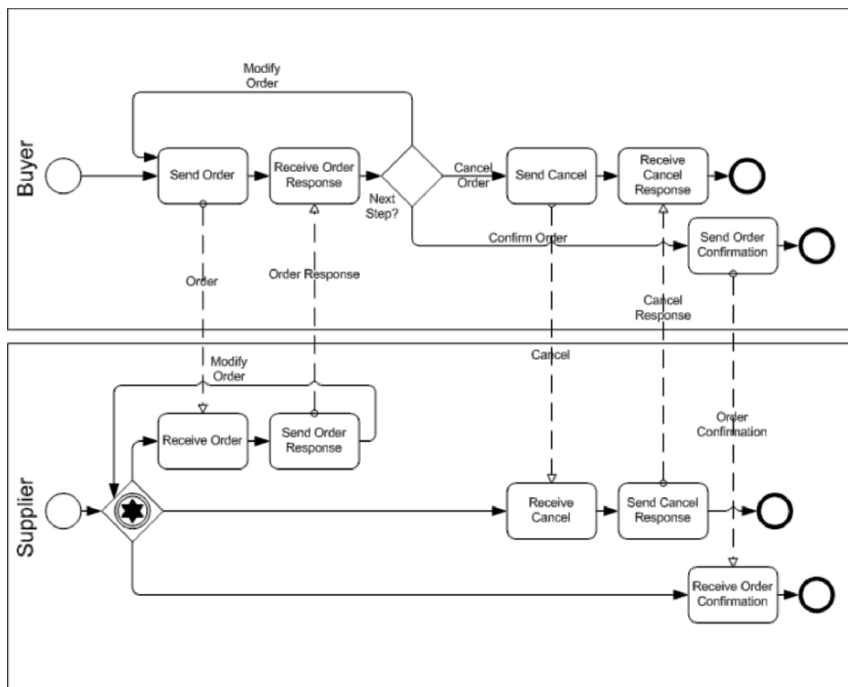


Fig. 5.3.4.1 Overview of Business Process Modeling Notation (BPMN) of an Order Processing in the site of www.bpmi.org

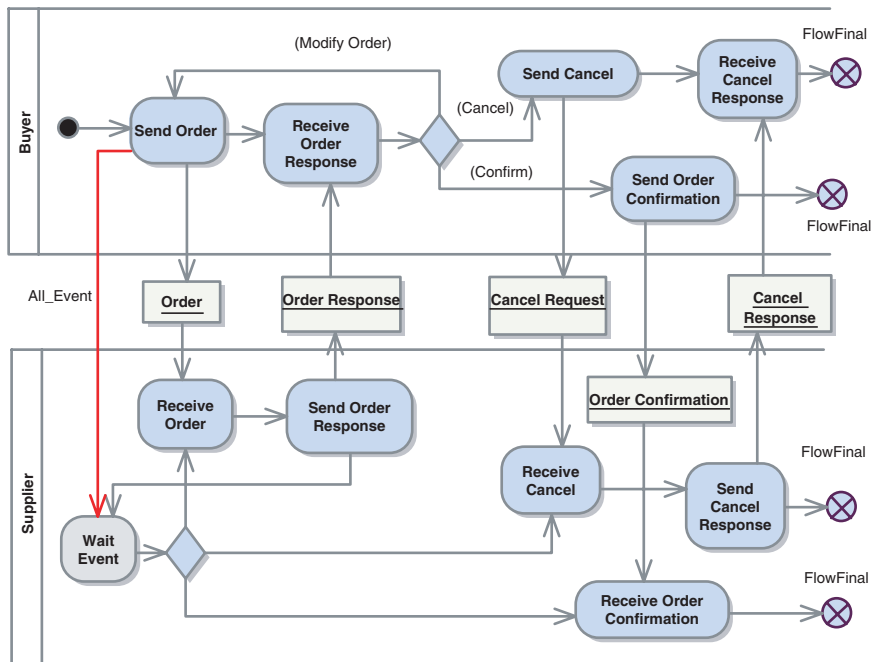


Fig. 5.3.4.2 Order Processing diagram converted from Business Process Modeling Notation (BPMN) to activity diagram notation. This diagram observes the original intent of its author

processes). They can be mapped into *Action* metaclass or eventually *Activity* metaclass.

DecisionNode metaclass in an activity diagram (represented with diamonds) are called *gateway controls* in the BPMN. The gateway in Supplier pool that contains a David star is a subcategory called “event based” (Exclusive OR type).

All solid arrows are *sequence flows* used to show the direction of process execution (equivalent to *control flow* in the UML). Dashed arrows crossing the pools are *messages* exchanged between Supplier and Buyer Pools. The notion of object flow in an activity diagram is replaced in the BPMN by *message flow* without requiring any object node placed somewhere on the trajectory.

We have converted the diagram in Figure 5.3.4.1 into its equivalent in Figure 5.3.4.2. All vertical arrows are object flows, except *All_Event* that is a control flow. When *Send Order* activity from the Buyer receives a first “token,” it sends a token to *Wait Event* activity, waiting for an object *Order* sent by Buyer’s *Send Order* activity and received by Supplier’s *Receive Order* activity. The token moves afterwards in the Supplier’s *Send Order Response* Activity that sends an *Order Response* back to the Buyer. Later, the token is refunded to *Wait Event*.

If the Buyer decides to modify the order, a new *Send Order* process will be initiated and the loop in the Buyer activity may be executed any number of times needed to obtain a full agreement between Buyer and Supplier.

Later, two scenarios may occur: an order confirmation or an order cancellation. The Supplier will react always from the *Wait Event* activity, according to the intention of the author of the BPD of Figure 5.3.4.1.

This example shows that the BPMN can be mapped into standard UML, with the activity diagram. As the activity diagram and the BPD are very similar, it is possible that they will converge more closely in the near future.

The BPMN targets some support in the future for process involving an intense collaboration between objects (choreography) and service-oriented architecture.

5.3.5 Business Process Management and Workflow

We have seen through examples that business processes in an organization are how to carry out fundamental tasks such as serving clients, buying, selling, delivering, etc. Some parameters are of prime importance for an organization that wants to optimize business processes.

1. Customer satisfaction
2. Efficiency of business operation
3. Increasing quality of products or services
4. Reducing production cost
5. Reducing operation cost, etc.

All those parameters need a constant *reengineering of business processes* to adapt themselves to the moving environment (technology, competitor, supplier, customer, regulation, etc.). *BPM* or *Workflow Management* is the branch that studies how well business processes are conducted in an organization. Sometimes, workflow management is considered more local or as addressing the same concept at a smaller scale than the BPM, but the frontier between these two concepts would be in the future a matter of convention.

Workflow Management (Georgakopoulos et al., 1995) is the activities of modern organizations. It involves coordination of daily activities, management of organizations resources, administration of internal information system; the whole process must comply with some business logic. Contemporary workflow management has been preoccupation in early 1970 with office automation (Ellis and Nutt, 1980), which were oriented towards the automation of human-centric processes. One of the famous implementation of the BPM was the Six Sigma concept initiated in the early 1980s at Motorola Company (Pande and Holpp, 2002). If we want to reach the workflow management ancestors, Fayol and Taylor in the first half of the 20th century were notorious pioneers in the domain of workflow management.

Another direction of workflow automation targets low level processes only. For instance, in the domain of Web Services, WSCI (Web Service Choreography Interface; available at: <http://www.w3.org/TR/wsci/>), WSCL (Web Services Conversation Language; available at: <http://www.w3.org/TR/wscl10/>), or BPEL4WS (available at: <http://www.bpmi.org/>) do not contains any human actors/agents and are rather focused only on technical co-ordination of interenterprise processes.

The market seems to seek a new standard for the BPM or workflow management. At the writing of this book, the last conference of this group (BPM Think Tank) was organized at Arlington, Virginia in May 2006 that gathered the business and technology experts in order to shape the BPM domain in the future. On the site of the OMG, people can still access a *Workflow Management Facility Specification* of 96 pages dated from April 2000.

5.3.6 Business Motivation Model

The notion of *motivation* is introduced to characterize this metalevel of business processes. If, at the conceptual level, we can say why a given business process is ill organized or takes so much time to execute, and what is the remedy to improve a given process, at this metalevel, we must be prepared to the question “if an enterprise prescribes a certain approach for its business activity, it ought to be able to say why.” So the question is targeted to the “raison d’être” of a business process and not the fact of admitting its existence as a fatality and stumbling on it year after year. A motivation model is thus recently coined to be able to give a correct answer to this inquisition. In addition, modern organizations tend to take into account new criteria that overstep the bounds of its proper structure, its internal components, and resources. Modern organizations, besides being constantly concerned about reducing the cost of doing business and optimizing profit, are more sensitive to the following discourses:

1. Better understand business by supporting internal information systems
2. Better prepared to deal with global competition
3. Identify new business opportunities
4. Rapidly develop new services and products
5. Set a business apart from competitors

Examples of this metalevel of business processes can be found in the literature:

BMM, 2005

[http://www.omg.org/news/meetings/ThinkTank/presentations/T-10_Hall_2.pdf

www.bpmi.org

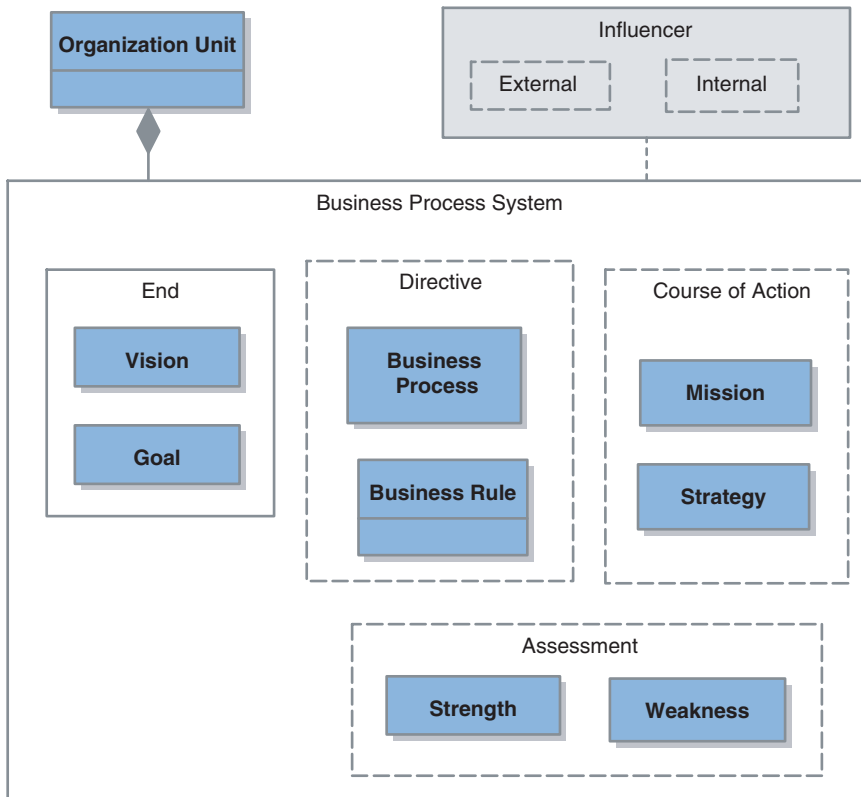


Fig. 5.3.6.1 Oversimplified version of the Business Organization as inspired from drafts on the OMG site to explain the mapping of business processes

www.businessrulesgroup.org

Audris Kalnins and Valdis Vitols

[http://melnais.mii.lu.lv/audris/Modeling_Business.pdf]

Once more, the BMM is outside the scope of this book, but its metamodel exemplifies a very interesting process model structured as a class diagram instead of a dynamic diagram like an activity diagram. So doing, from a fundamental viewpoint, it is not mandatory that processes must be mapped into class operations, actions, or activities. A whole process may constitute an entire class but this mapping must be justified by solid criteria.

Figures 5.3.6.1 and 5.3.6.2 summarize the BMM model. If we oversimplify the description of the BMM, a BMM starts often to define a *Business Organization* that is responsible for several *Business Processes*. Organizations are created with *missions*. To accomplish missions, organizations must have *visions* composed of *goals* quantified by *objectives*. Each mission is planned with a *strategy*

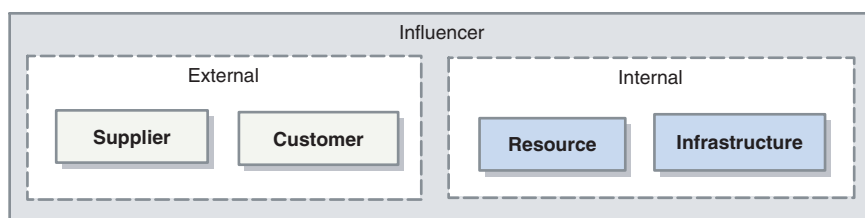


Fig. 5.3.6.2 Reduced version of the Influencer as taken from the OMG site

implemented by some *tactics*. Strategies and tactics must be compliant to *business rules* derived from *business policies*. Missions own some *risks* but have potential *rewards*.

The context of the organization is made of *internal and external influencers*. Influencers are objects inside or outside an organization that may affect the organization. They represent anything that has the power and at a less extent the possibility to influence and persuade a decision maker.

Customer assessments are beliefs (strength, weakness of the organization) that guide customer behavior. They may represent processes conducted to get these parameters. ISO 9126 defines assessment as a formal evaluation of a particular set of processes.

For a business organization, minimally, external influencers comprise two classes *Supplier* and *Customer*. Internal influencers are minimally *Infrastructure* and *Resource* classes. If internal classes must be developed, they can constitute abstract classes at the top of a derivation hierarchy to reach specific resources and particular piece of Infrastructure.

If an *Organization* owns a *Business Process Automation System*, Figure 5.3.6.1 gives some usual classes that could be defined. First a *Vision* class allows instances of a set of visions for the organization. *Goals* are business plans. A complete hierarchy of goals and subgoals can be built to realize missions that correspond to the visions of this organization.

The term *mission* can have many meanings (responsibility, even sometimes interpreted as vision) but the meaning adopted in the BMM is a series of planned operations to reach some goals. In this case, a mission can be modeled as a set of activities (or composite activity) within an activity diagram. To deploy a mission, we need a strategy instantiated from the *Strategy* class that gives recipes (knowledge) to build missions. Missions are ongoing operational activities of an organization.

Business processes realize courses of action (missions), provides processing steps, sequences, structure, interactions, and set up events. Business processes are guided by business rules. Business rules, organization units, and business processes are put into the BMM via placeholders but their exact definitions are in other business models.

5.3.7 Mapping a Process

All previous examples show the richness of process modeling and demonstrate that there is not a unique mapping for a process. In the UML, there are several metaclasses that can be identified as candidates for a process mapping. Inside a project, a process can be mapped to more than one metaclass if it is used in multiple views. Table 5.3.7.1 discusses all the possibilities of mapping a process, of any complexity, into the UML metaclasses. An object is only a “*role*” played by a real object, for instance, the Supplier or the Buyer is simply a role played by an object Person. That does not exclude the possibility that Person may buy manufactured objects that he sells.

To comment the possibility of mapping a process to a class diagram (e.g. business process), Figure 5.3.7.1 shows an example of a business process into a class diagram. A small process that can be executed individually by an object/component, map into an operation hosted by a class or a component. For collaborative tasks that need the contribution, sometimes equal of all participants, it would be hard to find a hosting object. For instance, if an enterprise starts a product line that needs the full cooperation of its staff, we can easily determine who must take the decision. But, is it really natural to affect this process to the Director object, the Board of Directors objects or the enterprise itself? The production process comprises a complete hierarchy of activities and the same problem of affecting operations to objects may be found as well at intermediate levels.

Affecting the business process to a Director means that this person is the only person, irreplaceable that could execute or instantiate the product line unless the operation is designed as a service in which case the Director works under the authority of another person (either the situation is very specific for this enterprise, either the solution exposes some incoherence). Moreover, if the Director is the only person that can start the production line, he holds all business processes of his company. This design is very centralized and may put a hard burden on the Director without any flexibility.

If we affect the business process as an operation of the Enterprise object and decide that one person belonging to the Board of Directors may instantiate the production process by invoking an operation of the Enterprise object, this choice seems to be an acceptable solution too. But, operation is a piece of behavior so states of objects depend upon current states of all their operations. For an enterprise that has hundreds of production lines, facing complexity, we cannot really combine all states and behaviors of each production line to build up composite state or behavior of the Enterprise object itself. So, if the solution could be acceptable for an enterprise with only one production line, it does not hold for an enterprise with several production lines. If a solution fits in very few

Table 5.3.7.1 Mapping of process into the UML metaclasses

UML metaclass	Diagram	Comment on the mapping
UseCase	Use case diagram	This metaclass is used mostly in requirement analysis, it must not be considered as being part of a final design or a run-time component, so it can represent all process or collaboration or service while specifying any system.
Operation of a class	Class diagram	<p>An operation is a behavioral feature of a classifier and as such can represent any process or service of an object. The condition is the natural and easy identification of the object that is responsible for the execution of this operation or this service.</p> <p><i>Example:</i> <code>send_order()</code> is an operation of the Buyer and <code>receive_order()</code> is an operation of the Supplier.</p> <p>This category of mapping includes both private and public operations of a class. A <i>private operation</i> is accessible by the object only. It can be called by a private or a public operation of this object. A <i>public operation</i> is an operation that can be called by any object including itself.</p>
Operation of an interface	Class diagram	<p>If the operation cannot be affected to any class and needs a collaboration of many objects to achieve it, possibly, it could be interpreted as a service accomplished by an interface from the Client viewpoint.</p> <p><i>Example:</i> <code>take_call()</code> is an operation that must be affected to a generic <code><<interface>></code> Vendor if the company is a long-distance call Provider. Each time we call to 1-800 numbers, a different person takes the call. Thus, <code>take_call()</code> is an operation of an interface and it could be realized by any Employee of this company.</p> <p>Normally, interfaces are designed with public operation only by its semantics.</p>
Collaboration and CollaborationUse	Composite Structure diagram	<p><i>Collaboration</i> is functionality, a macroscopic task that is executed by calling for collaboration of more elementary instances. A <i>CollaborationUse</i> (collaboration occurrence) is a collaboration applied to a specific context. A collaboration shows the roles and connections required to accomplish a specific task.</p> <p>In the context of a composite structure diagram, a process can be mapped to represent an activity executed by the collaboration of many objects. <i>Collaboration</i> or <i>CollaborationUse</i> belong to the structural view and shows only participating objects (parts) in the “process” but do not give any dynamic or evolutionary information.</p> <p><i>Example:</i> See Section 4.9.4</p>

(cont.)

Table 5.3.7.1 (Continued)

<i>UML metaclass</i>	<i>Diagram</i>	<i>Comment on the mapping</i>
<i>Action</i>	Activity diagram or State Machine diagram	<p>As a functional unit of executable functionality, an action can map an elementary process. At the same time, it corresponds to a class operation in a class diagram.</p> <p>Typically, an action is executed by an object (the action can be initiated by the object itself or an external object), a well-identified composite object, or even a component. In the case of collaboration between several objects, it is recommended to map into an activity.</p>
<i>Activity</i>	Activity diagram or State Machine diagram	<p>All processes not identified as actions could be represented in the dynamic view as activities.</p> <p>Activity can be found in the definition of state (Section 4.13), but activity or process is not state (a state is a set of clichés of a system taken while executing the process).</p>
<i>Interaction</i>	Sequence and Communication diagram	An interaction is a unit of behavior so it can be part of a process or eventually a whole process.
<i>Fragment</i>	Sequence diagram	An interaction fragment is like an interaction itself so it can be part of a process or a whole process.
<i>Complete sequence diagram</i>		May represent part of a process, a process or collaborative process between many objects (lifelines).
<i>Complete activity diagram</i>		May represent part of a process, a process or collaborative process between many objects.
<i>Complete use case diagram</i>		May represent part of a process, a process, or collaborative process between many objects in the requirements specification phase.
<i>Complete class diagram</i>		<p><i>Example:</i> business process</p> <p>As a class contains slots to store class operations, theoretically, any operation could be transformed into a class. This possibility could be misused as we can be tempted to create artificial classes just as a container for operations.</p> <p>However, there exist several situations that this mapping is justified.</p>
<i>Component or complete system (set of diagrams and artifacts)</i>		<p><i>Example:</i> Inscription process in a university. Project management, military operation, etc.</p> <p>In the business domain or in organizations, the initial view of all systems is mostly functional. Terms like “process,” “management of . . .” show that some people reason uniquely in terms of processes, tasks, and functions. So systems are built to achieve processes and objects and components are only resources. This fact justifies why high-level processes could be mapped into a component or a whole system composed of all UML diagrams and artifacts to explain the process model. A whole system can be implemented to perform a single process and the role of the process is highlighted instead of the supported pool of objects.</p>

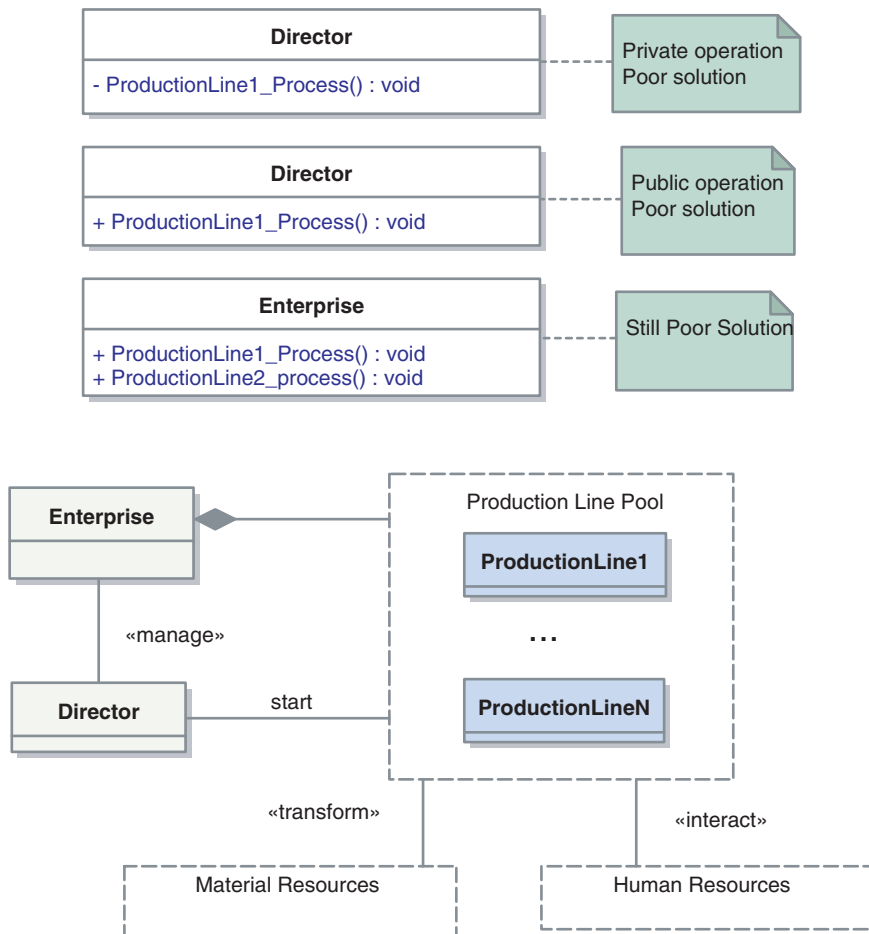


Fig. 5.3.7.1 Mapping Business Processes in an Enterprise (see text)

cases, it must be systematically suspected as a poor solution though not really incorrect.

The last solution retained is a *pool of business processes*, deriving from one or many classes. This design gives to a business processes many flexibilities and the view of systems being developed is *functional* at the top level instead of object. But this choice does not mean that we backtrack to the old functional methodology. Instead of considering how to activate structures and how to coordinate all the existing objects/agents/components to execute business processes, we start viewing a business process as a whole and independent entity. A business process can have attributes, holds various state variables to account for its evolution. We lay out all the steps to perform the process, decompose those steps

if necessary, identify objects, agents, or components that participate into the realization of the process, inventory all necessary resources to run the process. The whole system is still developed with object concepts and recipes. In the business domain, people often develop systems by first considering functions, tasks, or processes. As long as the final system adheres to solid object, design rules.

To summarize and give an answer to the question whether a process could be mapped into a full class or not, we recommend to map complex and collaborative processes (business processes are among them) into full classes as done in the BMM to facilitate their design, development, and monitoring. The characteristics of such a functional approach of developing business processes can be summarized with the following points:

1. *Business processes are considered as independent objects.* Each business process is considered as an instantiated object. The enterprise or the organization holds many business processes and controls their activation and deployment. The aggregation–composition relationship is used between Enterprise object and business process Pool. The business process is not an internal operation of the Enterprise object but an object held by the Enterprise.

The production line may have its own states that could be monitored independently from all the states of actors that contribute to the realization of the business process (Director, Enterprise, Employee, etc.)

2. *Objects/Agents are considered as participants in the collaboration.* Agents are omnipresent along the business process execution. They are considered as human resources. Some tasks of business processes are described as agent activities.

The Director agent starts the production line. He may control and monitor the execution of the production process but the process cannot be part of his definition. He has an important “role” to play but nothing really belongs to him. Employees who participate in the production line are collaborative agents. They offer their services against salary.

3. All other objects are products, *material resources*, or *services*. We do not distinguish products that the enterprise manufactures from resources it consumes to produce products. In fact, all material objects are combined, and transformed, and the final products are only last states or results of a long transformation process so the distinction between final products and resources used to produce them is only a matter of personal preference.



Fig. 5.3.8.1 Datastores used in the past with DFD. There is no datastore defined in the BPMN and BMM

5.3.8 Mapping a Datastore

The datastore models a collection of data. In a DFD (Fig. 5.3.8.1), the graphical notation of a datastore is two parallel lines (Yourdon-DeMarco) or some variants (Gane-Sarson). For a software engineer, it is tempting to look at datastores as short- or long-term memories, files, databases, or objects like optical disks or mechanical cabinets to store paper documents.

The UML has two concepts, `<<DatastoreNode>>` and `<<CentralBuffer Node>>` to map datastores. Central buffer nodes collect object tokens for multiple “in flows” and “out flows” from other object nodes. “They add support for queuing and competition between flowing objects.” Buffering is useful when the inputs from multiple actions must be collected at the central place or when multiple actions extract objects from a common source.

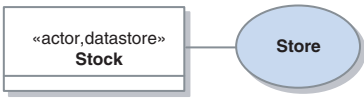
A datastore is an element used to define permanently stored data. A token of data that enters into a datastore is stored permanently, updating tokens for that data that already exists. A token of data that comes out of a datastore is a copy of the original data.

A datastore models a database in an activity diagram. A central buffer node materializes an object queue (for instance a system receives objects from a process but those objects are quickly consumed by another process and there is always a predetermined or limited number of objects inside the store). This number depends upon the size of the central buffer node. We can easily imagine the central buffer node as a “stock of merchandises” or a “printer buffer memory” that receives, in one end, characters to be printed from the processor and evacuates the same characters to the printer at the other end. Those two models of datastores are differentiated solely on the persistence of data inside the store. In a datastore node, the persistence is much longer (data can still be deleted in a database). In a central buffer node, we have just a “temporary” persistence. Objects in an enterprise warehouse are managed in such a way that input flow compensates output flow and there is roughly a constant number of products ready to deliver for a couple of days. This number is calculated by financial and business parameters like the delay imposed by the supplier, the intrinsic cost of the product (cash flow aspect), the speed of the input and output flows,

or some imponderable consideration as temporary and non recurrent business opportunities (special low cost series), etc.

The range of datastores found in the real world is a rich set starting from a dumb and unstructured memory towards intelligent storage systems. Table 5.3.8.1 gives some possible mapping to the UML metaclasses.

Table 5.3.8.1 Mapping of a datastore into the UML concepts

UML metaclass	Diagram	Comment on the mapping
(Non human) Actor	Use case diagram	<p>The metaclass Actor in the UML can be stereotyped with a <i>nonhuman</i> form to represent later a datastore.</p> 
Class	Class diagram	<p>Class can be used to represent all objects starting from a dumb and unstructured memory towards intelligent storage systems. For instance:</p> <p>Dumb and unstructured memory Hardware cabinet, shirt pocket, desk drawer, etc. Files, directories, DVD units Any kind of tank, pool, or container Databases, databases with RDBMS or ODMNS (relational or object Database Management Systems) Storage space for enterprise stock SAN (Storage area network) Any intelligent storage system with internal operating system.</p>
Component	Component diagram and Composite Structure Diagram	<p>A component differs from a class in that it has an internal structure that could be eventually explained (white box view), but generally kept hidden (black box view). A class can be used instead of a component if the internal structure is not known or can never be reached or detailed.</p> <p>A component is therefore generally used for datastores that has a minimal structure, intelligence, or internal operating system.</p>
Node	Deployment diagram	<p>A node is a computational resource upon which artifacts may be deployed for execution. Node can be used in PSM (Platform Specific Model) or in implementation model to represent for instance a SAN or a server dedicated to a storage or a database.</p>
Device	Deployment diagram	<p>Device is a physical Node and can be used for physical datastore with implementation details.</p>

5.3.9 Mapping Control Flows and Data Flows

In the UML, the data flow concept has been converted into the more general object flow concept, so data are considered as objects that embrace both data and objects. Control flow still remains in the UML with the same name but does not have the same meaning if compared to control flow in classical SART (Structure Analysis Real Time) methodologies.

Control flow in the UML (Fig. 5.3.9.1) is used to indicate the direction of the flow of control, to implement fork, join conditions, etc. Its nature has evolved towards the notion of transition (transfer from an activity to another activity) in a Petri Net (Peterson, 1981) but does not only mean a signal that activates or wakes up a sleeping or inactive process. Let us reproduce an arrangement of the DFD diagram in an acquisition experience before discussing subtleties in detail.

In this acquisition arrangement, we have two processes “Acquire and Convert” and “Read and Display” that are not decomposed. Full weight arrows are data flows. They are pathways through which data are transferred between processes, external entities, and datastores. Control flows are represented as dashed arrows oriented from the source of control towards the process to be controlled. Control flows are differentiated from data by the fact that they convey signals or data used to implement a protocol, to control events in the previous data flow networks. In the past, sometimes, analysts perceived the needs to separate the data flow network from the control flow network and drew two different diagrams (SART methodologies).

It is worth noticing that some communication protocols implement dynamics of protocols by reserving a specific set of data used as signals controlling the

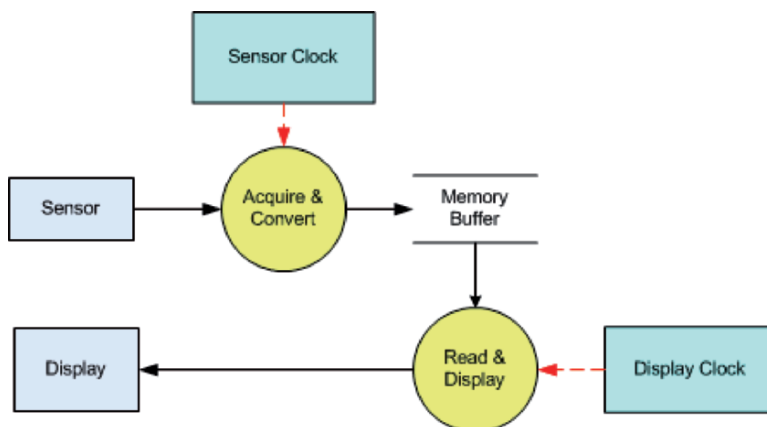


Fig. 5.3.9.1 Control flows and data flows in a DFD diagram

flow of data exchanged between a transmitter and a receptor. So, by examining the nature of data that are transmitted through communication channels, we can capture control information and differentiate them among a mass of data. For instance, in the well-known RS-232 communication lines, EOT/ACK (End of text, Acknowledge) and XON-XOFF protocols make use of some specific ASCII characters in the control block (from 00_h to $1F_h$) to regulate the flow of ASCII data through TD (Transmit Data) and RD (Receive Data) channels.

But, if we look at a large range of applications, sometimes, the distinction between data and control flows in the classic DFD is not so clear and sometimes, the process that receives data must analyze them to determine if there is some *control semantics that accompany data*.

For instance, if a clothing company receives frequently regular orders of ten thousands shirts and considers this kind of orders as regular data, the same company must reorganize its structure, seeks temporary workers if it receives an exceptional order of 200 thousands shirts. This exceptional order has simultaneously data and control meanings. First, this order is regular data if we consider that customers need what this company always produce, in occurrence shirts. Second, as the number exceeds its normal manufacturing capacity, the company must set up very quickly a temporary manufacturing structure in order to absorb this exceptional order. So, this order of “200 thousands shirts” bears some control aspects as it wakes up several uncommon processes of this company.

At the opposite end, we can find real applications that consider control signals as regular data.

For instance, all calls to 911 numbers are considered from the client side with “control semantics” but are treated as regular data by any employee working at 911 organizations.

To summarize, classic DFD diagrams allow us to display two kinds of concepts, data flow and control flow. Data in this diagram are “pure” data as it does not bear any “control semantics.” At the application level, control flow in this diagram means tacitly an agreement between the source (client side) and the destination (server or service side) to really consider it as a control. The meaning of “control” implies that the received “signal” wakes up or activates a process or some uncommon internal processes to handle data present at inputs and outputs of processes when the control signal is received. In Figure 5.3.9.1, each time the process “Acquire and Convert” receives a clock signal, it reads raw data at the input (acquire) then converts this data into a digital value. The real intent of the designer in this acquisition experience is “the Acquire process must take this clock event as reference to read the raw data and convert it into a 16 bit value.” This interpretation of the term “control” is not exactly the same in an UML activity diagram. In the UML, “A control flow is an edge that starts an activity node after the previous one is finished.” The control flow in the UML then indicates what the next process to be activated is and supports the construction of elaborate control structures. If we come back to the DFD

Table 5.3.9.1 Meaning of UML control flow, object flow, interrupt edge

<i>UML metaclass</i>	<i>Diagram</i>	<i>Meaning and indication for the mapping</i>
<i>Object flow</i>	Activity diagram	<p><i>An object flow is an activity edge that can have objects or data passing along it.</i></p> <p><i>Mapping:</i> This flow replaces all dataflows in a DFD (Data Flow Diagram) and most control flows in a DFD (see text for discussion).</p>
<i>Control flow</i>	Activity diagram	<p><i>A control flow is an edge that starts an activity node after the previous one is finished.</i></p> <p><i>Mapping:</i> This flow is a kind of “sequence flow” and indicates how activities are “fired” with time. This edge is used to implement conditions like fork, join, etc. to handle parallelism in an activity diagram.</p>
<i>Interrupting edge</i>	Activity diagram	<p>This edge is used to connect two activities together (one activity is interrupted by another activity). An InterruptibleActivityRegion may be defined to group all interruptible activities.</p>

and try hypothetically to interpret all control flows with the UML meaning, it appears that the *Clock Sensor* must first start then the control is transferred to *Acquire and Convert* by a control flow. *Acquire and Convert* is a sink for tokens. So, *Acquire and Convert* can do the job if it receives tokens from *Sensor Clock* that is supposed to be a generator of tokens. This interpretation with *Petri net logic* (towards that the UML was oriented with its version 2) fails in most DFD we try to convert. Control flow in the DFD diagram can be mapped as control or object flow and the structure of the DFD must be modified in an important way to support token movement. In this sense, the term “Sequence Flow” used in BPMN to replace “Control Flow” could sound more appropriate as the term “control flow” presents some dangerous misleading interpretation when confronting it with the common meaning interpreted by a generation of electronic engineers and process modelers.

The conversion of the DFD of Figure 5.3.9.1 to an activity diagram is more appealing as in activity diagram, partitions can be used to separate objects and clearly show the parallelism of two processes (acquisition executed by the Acquisition Card and display process executed by the Display Processor). These two objects act independently and communicate through a *Double Access Memory*. The Memory Buffer is now characterized by its type <<CentralBufferNode>> that pinpoints its buffering nature.

The most important point is the conversion of all control flows in the past to object flows (arrows connecting two objects :*Sensor Clock* and :*Display Clock* to their corresponding processes). Object nodes “raw data,” “16 bits,” “analog,”

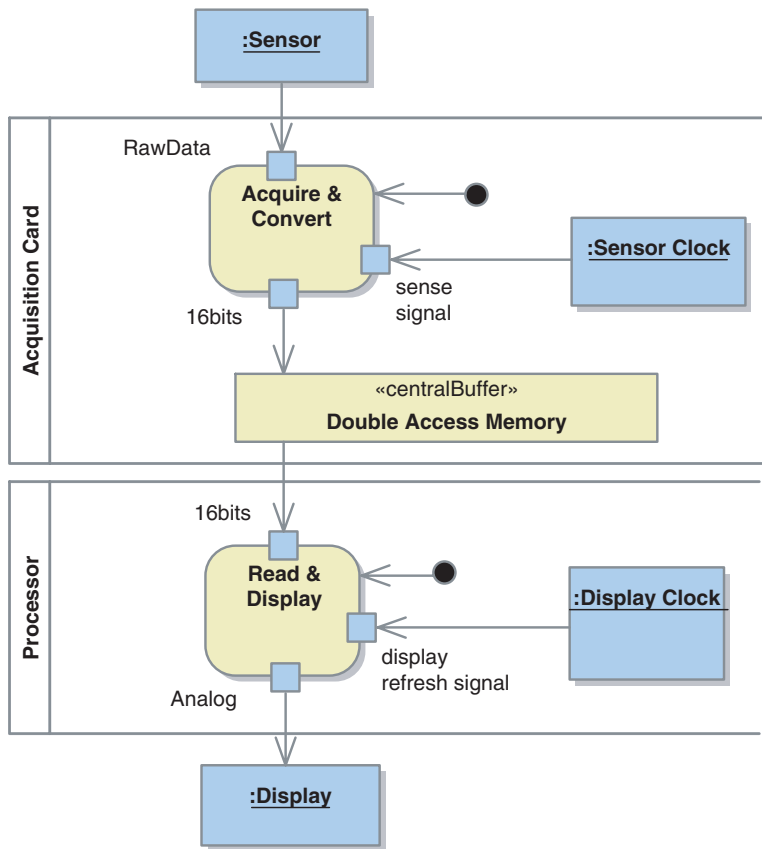


Fig. 5.3.9.2 Conversion of the DFD of Figure 5.3.8.1 into activity diagram. Signals “sense signal” and “display refresh signal” sent by clocks are used as time references to acquire and display. Signals are considered in this example as regular objects. If we represent them as control flows, when tokens traverses from Clocks to Processes, it appears as Sensor Clock and Display Clock are deactivated, that may cause interpretation problems. Actually, Clocks are source of tokens and the two processes must have token sinks. The conversion with control flows (with Petri interpretation) is more complicated

“sense signal,” and “display refresh signal” are added. This conversion seems very uncommon (may eventually surprises process analysts) but it highlights one of the many differences between a classical process view and the object paradigm.

5.4 Fundamental Concepts of the Dynamic View

As dynamic view is part of system behavior, it would be natural to continue with fundamental concepts of dynamic view as they are closely related to processes, actions, and activities. The dynamic view of a system reveals

temporal, behavioral, and evolutionary aspects of systems. Traditionally, this view is complementary to the functional view (process view) in the study of systems, mainly real-time systems. If database development gives more importance to the data view (structural view), i.e. conceptual schemas of data, if business process engineering is focused on processes (process view), real-time systems need a complete investigation of the dynamic view to be able to build systems. This view determines:

1. States of each object in the system (set of snapshots showing how an object behaves with time or under stimuli, events, etc.)
2. Composite states of many objects taken as a whole
3. List of all events, conditions that shape the behavior, reaction of objects
4. List of actions available to each object
5. List of activities available to objects or pools of objects
6. How objects collaborate together to accomplish tasks
7. Exception processing (how systems react to error or abnormal but predicted situations)
8. Possibility of concurrence

As results, we have a set of diagrams, documents, and artifacts that allow us to explain and communicate our visions of the system to be developed. The UML has identified three complementary axes in the dynamic view:

1. Activities (activity diagram)
2. Interactions (diagrams of the activity suite: sequence, interaction overview, communication, timing)
3. States (state machine diagram)

The choice of diagrams, thus axes, to represent the dynamic view of systems depends upon the nature of the system, its best format for communication purposes, tools available at the development time, policy of the company that takes over the development, and finally some dose of personal taste of developers involved in the project.

5.4.1 States and Pseudostates

A *state* is an interesting step, phase, and evolutionary stage of a system. The term “state” belongs to the UML domain and has a defined metaclass. A state of an object is defined by activities it is executing and a state does not mean necessarily “static”.

A *person is walking*, a *plane is flying* are states of Person and Plane objects.

A process in an operating system may evolve through many known states as: *ready* (runable and waiting to be scheduled), *running* (currently active and using CPU time), *blocked* (waiting for an event or an IO signal to occur).

When a Vendor in a shop is waiting for customers, he is in an idle state.

Table 5.4.1.1 Description of the UML concepts related to the real world concept of “state”

UML metaclass	Diagram	Comment on the mapping
State	State diagram	<p>A state models a situation during which some invariant conditions hold. This definition of the Superstructure Book can be translated into fixed or bounded values for state variables.</p> <p>Metaclass attributes of State metaclass are:</p> <p><i>isSimple</i> : Boolean. If <i>isSimple</i> is true, the state is non decomposable (elementary state)</p> <p><i>isComposite</i> : Boolean. If <i>isComposite</i> is True, a state is a composite state. In this case it contains at least one region. Many states, transitions, and pseudostates may fill up regions. <i>isComposite</i> means that the state is not elementary and could be decomposed into more elementary states if needed.</p> <p><i>isOrthogonal</i> : Boolean. An orthogonal state contains two or more regions. “Orthogonal” implements AND-states logic, so the main state combines all active states, one in each region.</p> <p><i>isSubmachineState</i> : Boolean. If true, the state is a complete submachine state so must be decomposed if details are needed. A submachine state may have regions too, so decomposition may be realized in both horizontal (regions) and vertical (substates of state until elementary states are reached).</p>
Activity	State diagram	In a state diagram, states are defined by main activities written under the Do Action keyword.
Action	Activity diagram	<p>Actions in state machine diagrams appear as <i>On Entry</i> action, <i>On Exit</i> action, actions executed when transitions occur.</p> <p>We may consider them as transient states. If the context of the problem requires that <i>transient states</i> must be studied thoroughly as people suspect abnormal behavior of the system, actions may be eligible to become regular state.</p> <p><i>Example</i>: A driver may decelerate a car that passes from <i>cruising speed</i> state to <i>deceleration</i> state by just <i>releasing the gas pedal</i> (very short action) and <i>pushing the brake pedal</i> (action). For a specific study “reactions of drivers over 70,” actions may become critical and justify in this case full state consideration.</p>

(cont.)

Table 5.4.1.1 (Continued)

UML metaclass	Diagram	Comment on the mapping
<i>Operation</i>	Class diagram	<p>When a system executes an operation (operations are defined in class), we talked of execution states.</p> <p><i>Example:</i> An order management process may have a series of operations as <i>take_order()</i>, <i>verify_stock()</i>, <i>prepare_delivery()</i>, <i>prepare_invoice()</i>.</p> <p>As operations are behavioral concepts, when a system is executing a given operation, it is in a “state” of executing this operation. This fact shows a thorough coupling between the functional view and the dynamic view. When a system is examined at higher level of decomposition, it is best apprehended as <i>behavioral</i>, as done in the UML.</p>
<i>(full diagram)</i>	State machine, activity, sequence, communication, timing diagrams	<p>Notice there is no metaclass “diagram” so the term “full diagram” is put into parentheses.</p> <p>A <i>state machine diagram</i> can be a submachine state before decomposition process, so it is a candidate for a composite state or submachine state.</p> <p>An activity diagram may be used to explain state evolution. Activities have a close relationship with states. A state can be defined by an executing activity and a complex activity can go through many states when executing.</p> <p>A sequence or a communication diagram is a unit of interactions so they contain state evolution of a system.</p> <p>A timing diagram show timings of systems so it also describes system states.</p>
<i>Fragment</i>	Sequence diagram	<p>A fragment is a unit of interaction. So, fragments describe system states.</p>

States may be categorized by *state variables* that are mathematical elements used by a designer to model a system or an object. State variables can be defined officially as object attributes. In this case, attributes are not structural but behavioral (functional or dynamic). A state occupies a time slot. Inside this time slot, we suppose that all values of state variables are fixed or bounded by an interval.

Let us take a simple LED (Light Emitting Diode) used as signal lamp in a wide range of applications. It has structural variables as *dimensions*, *color*, and *electrical intensity*. Dynamic variables are reduced to *state* that can take two binary values *On* or *Off*.

A mechanical switch or a push button plays the role of input devices for numerous systems. It has a dynamic variable that can take two binary values *On* or *Off*.

A shift lever of a car has normally a dynamic variable with the following values: *Park, Reverse, Neutral, Drive* and *D3* or *D2*, etc. Therefore, we have a state variable with an Enum (enumeration) value.

Water may be considered as having three physical states: *Ice, Liquid and Vapor*. If its temperature T is taken as state variable, water is *Ice* when $T < 0^{\circ}\text{C}$, *Liquid* when $0^{\circ}\text{C} < T < 100^{\circ}\text{C}$ and *Vapor* when $T > 100^{\circ}\text{C}$. Values of water temperature are bounded and made discrete by imposing limits.

To handle analog values, the same conversion to discrete values to ease the description of states is very frequent. For instance when the speed of a car passes from 0 to 65 mph, the state of the car is *accelerating*, when reaching this ceiling value; the car has reached its *cruising speed*. When passing from 65 mph to 0, the car is *decelerating*.

Dynamic attributes are not so evident to find out. Object actions and object activities must be identified before starting a dynamic study as states are directly tied to activities.

Before knowing that a state attribute of a Shift Lever object may take these values: *Park, Reverse, Neutral, Drive, D3* and *D2*, we must now that the Car object can park, can drive, can move backwards, can be in a Neutral state, etc. So, a preliminary functional study must be conducted to be able to define state values.

Pseudo states in the UML have already been studied in Section 4.13. The UML has defined *initial, final, deep history, swallow history, join, fork, junction, choice, entry point, exit point, terminate* pseudostates, adding diagrammatic artifacts used to guide the interpretation of the diagram and implement its logical structure. For instance, *initial* pseudostate in a diagram shows where to start reading the diagram and the next state after *initial* is a real state of the system.

5.4.2 From Elementary State Towards Global System State

The *State* metamodel of the UML (Fig. 4.13.0.2) defines a state with four parameters (*/isComposite, /isOrthogonal, /isSimple, /isSubmachineState*). Hence, full-featured states may have regions, can be decomposed indefinitely towards more and more simple submachine states. At the bottom of the decomposition hierarchy, we reach a simple state. When analyzing system states, we have many scenarios:

1. When an object executes an operation, states of the operation are also object states relative to this operation
2. If an object performs simultaneously many operations, we still consider individual states relative to each operation unless operations are dependent. In the latter case, combined states may be required to describe object states
3. A collaborative task is executed by many objects. In this case, each object contributes to this task with their proper operation set. Normally, the task

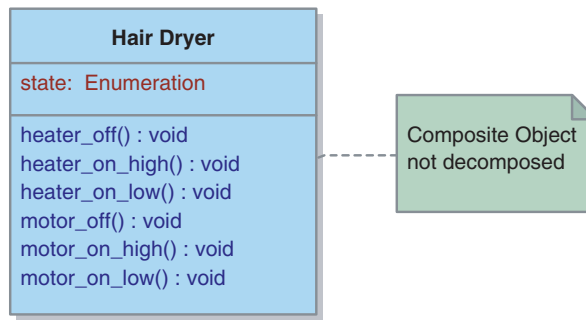


Fig. 5.4.2.1 States of the Hair Dryer not decomposed. As many as 64 states are possible at the beginning, based on six binary operations (activated or not)

has its hosting object (for instance “Production Line”) and each object has its proper states (case 1 or case 2)

Figure 5.4.2.1 represents states of a *Hair Dryer* composed of two elementary objects, a *Motor* and a *Heater*. If the object *Hair Dryer* is kept as composite object and not decomposed into elementary objects, it will have operations like `heat_off()`, `heat_on_low()`, `heat_on_high()`, `motor_off()`, `motor_on_low()`, `motor_on_high()`. Each operation offers two states *True* and *False*. With six binary variables, we reach 2^6 or 64 possible states. Some combinations are impossible and must be rejected after analysis but we focus here on the initial count.

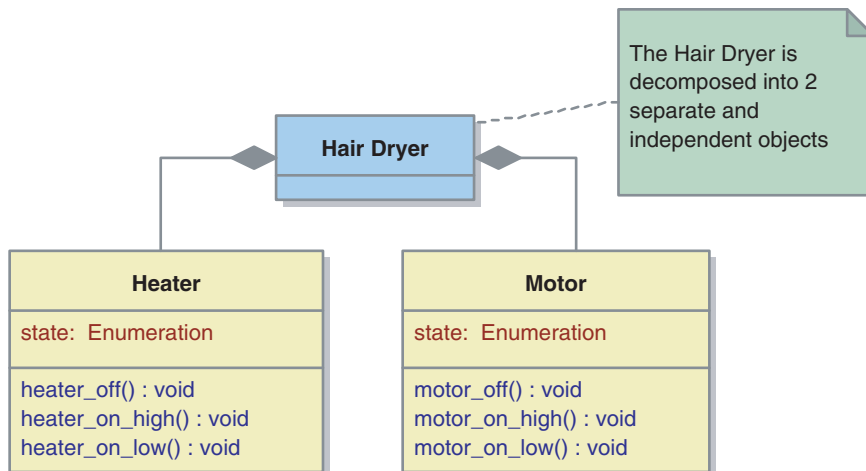


Fig. 5.4.2.2 States of the Hair Dryer decomposed into two separate objects. The number of states is 16 ($8 + 8$)

(Note: We can consider an enum type variable a *state*, but this variable is a final variable that considers an implemented subset of combinations chosen among the 64 possible states. At this stage, we are still designing the system.)

Now, if the Hair Dryer is decomposed into two separate objects, Heater and Motor, each object will have only $2^3 = 8$ states. By adding states of Heater and Motor, we have only $8 + 8 = 16$ states. We can eventually add to the composite object Hair Dryer a binary state On or Off. The final count is only $16 + 2 = 18$. So, from a dynamic viewpoint, if a system is not thoroughly decomposed and studied, whether we forget states, whether the composite system is so complex that the dynamic study cannot be conducted correctly. Moreover, semantics of composite states is more difficult to grasp. For instance, in the case of the Hair Dryer, we can only say that the Hair Dryer composite object is On or Off. Other enumeration states like “full power,” “full heater power,” “low fan,” etc. need communication details.

For an enterprise that has several running production lines, it would be easier to quote states of elementary production lines than asserting the overall state of the Enterprise object through all of its production lines. But, saying that this enterprise is at 70% of its production capacity gives sometimes helpful information for investors, stakeholders. But this information are not technical, it is synthesized by often an unknown or undefined algorithm.

In a classroom, how to quote the state of a classroom if 20% of the students are sleeping, 10% are listening to music with their ipods, 30% is listening to the teacher and the rest are doing their homework with their laptops?

“Composite states” of a composite object resulting from the combination of more elementary states of all of its components may need some rework to offer a more suggestive information. In the real world, if some aggregates form real composite objects (thousands of mechanical pieces of a car make up a composite car), a collection of objects like students in a classroom does not constitute a composite object, so the characterization and interpretation of states of such a collection are often questionable as they could be arbitrary and subjective.

Getting all significant states of a system is a difficult exercise in modeling. The most evident recipe is breaking the system into smaller chunks, decomposing composite objects into more elementary objects; studying states of elementary objects, and then reassemble elementary objects to find a signification for composite states of composite objects.

If an object executes several tasks or operations at the same time and each task corresponds to a collaborative activity engaged with different partners, we encounter the same difficulty. In this case, we must examine each task separately.

Sometimes, the problem to be solved is so complex that it must be split into more elementary smaller problems.

What is the state of a person P if he is verifying a stock for Customer A, taking an order for B, and preparing an invoice for C? In fact, P can do several

operations *take_order()*, *verify_stock()*, *prepare_delivery()*, *prepare_invoice()*. If we are studying the business process starting from the reception of the order to the collection of payment, it is recommended to consider the states of this person P through interactions with a generic customer that could be A, B or C. States retained through this long process are: *idle*, *taking_order*, *verifying_stock*, *preparing_invoice*, *preparing_delivery*, *getting_payment*.

Now, if the person P is able to work apparently on several tasks at the same time means that we can explore some parallelism with the resource used to implement this business process. A new scheduling problem of “evaluating how many customers could be handled by one person during one day” will be superposed to the previous problem. The second problem is typically a workflow management problem. If we have n Customers, new states could be added as: *idle*, *busy_with(i)*. The *busy_with(i)* is a global state that encompasses *taking_order(i)*, *verifying_stock(i)*, *preparing_invoice(i)*, *preparing_delivery(i)*, *getting_payment(i)*. A priority scheme can be added to take into account real conditions. For instance, if P is currently busy preparing a delivery for Customer i and if P receives a phone call from Customer j, he can postpone the task for i as the activity of taking a new order is more important than preparing a delivery. Final output of the workflow study could be for instance estimating the average number of customers the person P can serve during one day.

It is worth noticing that the conventional state used for state machine diagram is not elementary as it defines several components inside.

Figure 5.4.2.3 gives the structure of a standard state. It is composed of an activity that defines the state (*character/handle character* that represents the *Do Action*). While entering the state *Typing Password*, an action has been performed *On Entry* (*set echo invisible*). *On Exit*, another action *set echo normal* is activated. An action can be defined *On Event* (*help*). An equivalent activity diagram shows the equivalent of the standard state in the UML.

If *handle character* is candidate for a state, *set echo on* and *set echo normal* are also candidate for *transient states*. Transient states are temporary states that a system passes through during an insignificant time (compared to the time

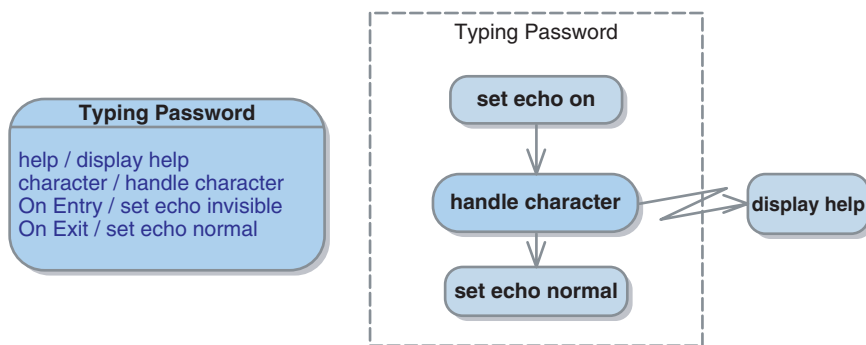


Fig. 5.4.2.3 Conventional state used with the UML is not elementary state (Fig. 15.32 Superstructure Book)

scale of the current system) before reaching an equilibrium state. Sometimes, transient states must be studied if some behavioral problems are suspected during transients. In this case, Figure 5.4.2.3 must be decomposed into three different states. This is the reason why the standard state of the UML is not really elementary.

To summarize, it is worth noticing some important points when working with states:

1. States are a very elastic concept that depends upon the level of decomposition of a system. States can be tied to object operations at high level, assimilated to activities in general cases, and possibly to elementary actions when transient states are investigated.
2. As states are states of objects through their operations and activities, the separation into functional and dynamic view are for description purposes only. Functional and dynamic study must be conducted simultaneously. The UML talked of behavioral study.
3. When decomposable, states are composite or submachine states. Composite states are states of several objects identified in regions (“horizontal” decomposition) connected through AND-states or OR-states logic. Submachine states can be decomposed “vertically” into more elementary submachine states or elementary states.
4. If object or agent states are easily understood concepts, states of a collaborative group of objects or a collection of objects (without any collaboration) are harder to describe. Interpretation is contextual and subjective if not done correctly. Generally, a precise description ends with a description of states of constituting elements.
5. To get a thorough description of a system, it is recommended to always decompose the system to its utmost elements, study states of elementary objects, then reassemble them into composite, subsystems or components, finding correct descriptions of states at any level of concern.

5.4.3 Actions and Activities

Action is defined in the UML as a “fundamental unit of executable functionality.” It would be difficult to find a clear definition of an elementary term without risk of circular definition. The UML has defined about 37 elementary action metaclasses (*CallAction*, *ReplyAction*, *SendObjectAction*, *DestroyObjectAction*, etc.) to describe a minimal usable set of actions. Users may define their own set through stereotypes if needed.

In object/agent system, we consider an action as a nondecomposable act of doing something performed by an object/agent that can modify the state of

itself or the state of its environment (other objects/agents). An action results in a *world move* and constitutes a unit of analysis while working on human interactions. Actions are contextual and they cannot be understood outside this context.

A man can perform the following action “make a weapon”. But we don’t know if he makes a weapon for hunting, for killing somebody or for selling it to get money.

Contrary to actions that are contextual, an activity is composed of a series of actions (repetitive or not), and has a more advanced structure, particularly if it has motives and goals. If actions may be unconscious, activities that are coordinated actions are goal-directed. When an activity performs, it needs *subjects* or *actors* (objects/agents or doers responsible of the activity), *objects* (what to change), operations and actions (procedural steps, processes), and all *artifacts* and *resources* needed to execute the activity. All ingredients constitute the context of this activity but after execution the activity reshapes its context. Context is not only external but is internal as well.

If an enterprise “proposes a new product”, “promotes it”, perceives the reaction of customers, changes its strategy, etc. Then, activity may change the goal of this enterprise, so may change its internal context.

There is a close relationship between an activity and the state of the object that executes this activity. Less frequently, there is also a close relationship between the definition of a state and actions. A state can be defined by rich combinations of actions and activities:

1. State defined by a *continuous action*

To maintain a speed of 65 mph, the driver must act constantly on the gas pedal. We can eventually say that the state is defined by a fixed parameter that is the cruising speed of the car, but this constant speed is controlled by the continuous action of the driver.

2. State defined by *repetitive actions* during a state

If a fax calls at an occupied number, it will enter in a *retry state*. In this special state, it will try five times to make a redial before deciding that the number cannot be reached.

3. State defined by intermittent actions

Software shows intermittent bugs. As long as programmers cannot find where bugs come from, the software will be in the debug state and cannot be sold.

4. State defined by several activities

A company enters a liquidation state and needs to clear away stock in Canada and in the USA. If the warehouse in Canada is cleared, the company is still in its liquidation state globally.

5. State defined by a *date*

Spring starts in March 23 and marks the blooming period for several spring flowers, but all species will not give flowers at this date.

As there is a close relationship between action, activities, and states, please refer Table 5.4.1.1 built for states in order to map real world actions or activities.

5.4.4 Events

The UML defines the *Event* metaclass as “An event is the specification of some occurrence that may potentially trigger effects by an object.” An event is a fact occurring in time, can capture attention of a system and cause its state change. It is always the result of a direct or indirect action or activity of an object (or system or agent). This object may be internal or external to the current system and sometimes, we cannot really identify the object that creates the event.

Turning the car key contact is viewed by the object Car as an event, but the action has been performed by the Driver object.

A click on a mouse is received as an event by the software but the action is executed by the User object.

An intruder acts as an event for an alarm system.

When the phone rings, we receive the ringing as an event but the object Phone is responsible for producing the ringing tone.

The arrival of the 737 or an Airbus or Boeing flight is an event for people waiting for their relatives.

Other events are for instance the opening of the door, the arrival of new expected software on the market, the change of the tax ratio, the disappearing of dinosaurs on the planet, etc.

An event may occur at *regular* time (clock chimes), may be synchronous to another event. “After each rain, insects appear. The appearance of the insect is synchronous to the event ‘rain’.” An event that appears regularly with time is sometimes called synchronous. In fact, time is perceived as something regular, and any event that occurs on a regular basis is perceived as synchronous with time, thus simply synchronous.

An event is said asynchronous when we cannot relate it to anything, to any other event or any timing reference. The occurrence of an asynchronous event may be possibly predictable but we cannot say exactly when this event will occur.

Synchronous and asynchronous may bear another meaning in modern communication.

In a meeting, when all parties involved in the communication are present at the same time, we have a synchronous communication. Examples include a telephone conversation, a company board meeting, and a chat room. Asynchronous

communication does not require that all parties involved need to be present and available at the same time. Examples of this include e-mail service (the receiver does not have to be logged on when the sender emits the message), discussion threads, which allow exchanges over some period of time.

Random events are events with unpredictable outcomes. Random events may come from undefined sources, but some random events come from well-known sources (mathematical random number generators) but, for those who ignore how sources create or generate events, they appear as random. Random generators are a very interesting research concern as they are useful in simulating complex systems, such as the spread of diseases, the evolution of financial markets, or the flow of Internet traffic, besides constituting the brain of Las Vegas slot machines.

An event is modeled as a fact without any time duration. From a theoretical viewpoint, this hypothesis is false as nothing is instantaneous (even the light needs 1 ns to travel 1 ft), but in modeling, sometimes, systems must be studied with a rough model, then, when dynamic studies are necessary, we reiterate through models several times to refine them. This attitude allows us to ignore, at first approximation, all transient states and consider only most important ones to have an approximate view of the project before starting more meticulous dynamic studies to reach final algorithms or a procedural way of solving problems.

In the UML, the term “event” means the type and the word “occurrence” means the instance. When necessary, “event type” and “event occurrence” will be used. It could be surprising but the UML has no graphical notations to represent events, but, as events are created by actions, in some tools, `AcceptEventAction` and `SendEventAction` may be used as sink or source of events (see Table 5.4.4.1). In fact, events are omnipresent in dynamic diagram of the Behavior suite, is part of the state transitions, lifeline interactions, and messages in timing diagram. Events are the main driving force of system change and world movement. For instance, in an interaction diagram, when a lifeline sends a message to another lifeline, the receiver interprets the message implicitly as an event. In a state diagram, most transitions occur as a result of an event. The event appears in the “trigger” part of the transition specification.

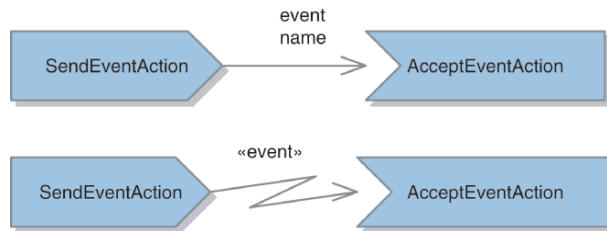
5.4.5 Condition

The UML defines “A constraint is a condition or restriction expressed in natural language text or in machine readable language for the purpose of declaring some of the semantics of the element”. In the UML, a condition is therefore some constraint put on a model element to shape its behavior. A condition must be evaluated to be true in order for the constraint to be satisfied. Preconditions and postconditions are used intensively in operation or protocol specifications. Condition nodes or test nodes are nodes used specifically to determine if a

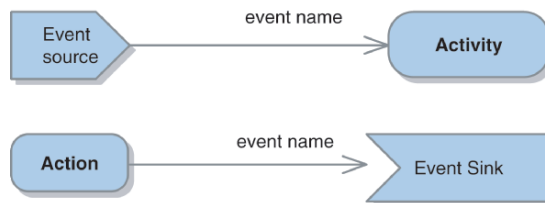
Table 5.4.4.1 Support for concept of event in the UML

Diagram	Comment on the mapping of event concept
State machine diagram	In a state machine diagram, events appear in the “trigger” part of the transition specification. Trigger [guard] / effect →

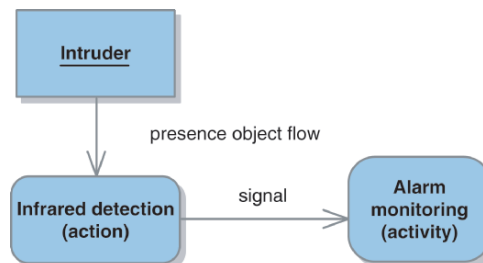
Activity Diagram	In an activity diagram, each action or activity, dependent on their nature, may potentially generate an event towards another action or activity. The following examples quote some possibilities:
------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



In the figure, SendEventAction acts as event source, and AcceptEventAction as event receptor. Control or object flows can be used as supports for events.



Regular activities and actions can replace specific graphical notations. Control or object flows can be used as supports for events.



Objects can send events to actions that can send other events to activities. Object or control flows may be used as supports for objects, signal, events, or messages.

Table 5.4.4.1 (Continued)

Diagram	Comment on the mapping of event concept
Sequence diagram	<p>When a lifeline sends a message to another lifeline, generally, the receiver interprets the message implicitly as an event. So, events are implicitly encoded in interactions. Please notice that, comparing with the example in the activity diagram, actions and activities are converted into objects (infrared detection to Infrared Sensor and Alarm monitoring into Alarm Detector).</p> <pre> sequenceDiagram participant Intruder participant Infrared Sensor participant Alarm Detector Intruder-->>Infrared Sensor: presence object flow Infrared Sensor->>Alarm Detector: signal note over Infrared Sensor: Infrared Sensor receives the object flow as an event note over Alarm Detector: Alarm Detector receives the signal as an event </pre>
Communication diagram	<p>Idem for communication diagram that is a replica of the sequence diagram.</p> <pre> sequenceDiagram participant Intruder participant Infrared Sensor participant Alarm Detector Intruder->>Infrared Sensor: 1: presence object flow Infrared Sensor->>Alarm Detector: 1.1: signal </pre>

condition is evaluated to be true. Conditions appear as guards that determine if transitions must occur or not in state machine diagram.

Condition in spoken language has a rich set of meanings. It can be assimilated to a state. “The Car is in a bad condition.” Or, when defining states, the UML declares “A state models a situation during which some invariant conditions hold.” Initial or final conditions are effectively states as conditions do not evolve. The slight difference is that conditions focus the description on some details of states but are not intended to completely describe the states.

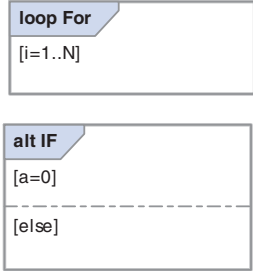
Some imprecise uses of “conditions” say for instance “When entering a state, *On Entry* actions must be executed or when exiting a state, *On Exit* actions must be executed.” In fact, some “mentioned conditions” are implicit to the definition of states and this imprecise prose seems to add supplementary conditions that must normally be pruned off.

If the Car must enter its deceleration state, the Driver must release the gas pedal and pushes on the brake pedal. So, the action on entry of the state is not a condition but an regular action of the state if from the beginning, states are designed with both On Entry and On Exit inside. However, if On Entry and On Exit actions are defined outside the state, their activation may be ruled by conditions, e.g. “Want to decelerate?” Yes, No, etc.

Table 5.4.5.1 This table explains how the UML deals with conditions at various levels

Diagram	Comment on the mapping of conditions
Any diagram	<p>Constraints are a kind of condition</p> <p><i>Constraints</i> are conditions or restrictions imposed on nearly any model element. Constraints may appear for instance on class, on sub elements of a class definition (attributes or operations). Constraints may be applied to a whole project, to parts of project, to several diagrams, to one diagram, etc. People talk of local or global constraints.</p> <p>Constraint is often identified with a pair of curly brackets {}</p>
Any diagram whenever applicable	<p>Relationships and stereotyped relationships</p> <p>All relationship may turn into condition ruling the association between two modeling elements as a relationship can tie a class to another class.</p>
Any diagram whenever applicable	<p>Pre and Post conditions of operations</p> <p>“The pre conditions for an operation define conditions that must be true when the operation is invoked.”</p> <p>“The post conditions for an operation define conditions that will be true when the invocation of the operation completes successfully, assuming that the preconditions were satisfied.”</p>
State Machine diagram	<p>Guard part of transition specification</p> <p>A guard imposes a condition that, if true, authorizes the transition to take effect.</p>
Activity diagram Interaction Overview Diagram	<p>Some control nodes of the activity diagram</p> <p>A <i>decision node</i> imposes a condition to the system, selecting the next activity to be executed. A <i>merge node</i> imposes that all flows must converge to a unique output. A <i>fork</i> imposes that a set of tokens is distributed simultaneously at all outputs (Petri Net logic). A <i>Join</i> imposes that all tokens must arrive before the next activity could be activated (Petri Net logic).</p>

Table 5.4.5.1 (Continued)

Diagram	Comment on the mapping of conditions	
Sequence and Interaction Overview diagrams		<p>Fragments with predefined conditions</p> <p>Fragments are predefined conditions put on units of interactions for definition of standard fragment types: alt (alternatives), opt (option), break, par (parallel), loop, critical, neg (negative), assert, strict, seq (sequence), ignore, consider.</p> <p><i>Example:</i> Fragment for implementing a FOR loop condition and an alt fragment for an IF condition.</p>
Use case diagram	<p>Include and Extend Relationships</p> <p>The two most important <<include>> and <<extend>> relationships are the basic mechanisms for decomposing a use case into more elementary use cases and are therefore a feature of complexity reduction. They are metaconditions imposed on the way processes are connected together.</p>	

When a sequence of actions (for instance, A1 then A2 then A3 must follow in this order), there is an implicit condition on the sequence order. Otherwise, we must add everywhere in the activity diagram constraints that A1 precedes A2 that in turn precedes A3. If we have an automatic code generator, it will translate the actions in this order. So, to avoid a very complex specification, it is recommendable to keep conditions specifications only in situations enumerated in Table 5.4.5.1.

Moreover, it would be useful to distinguish conditions and constraints imposed by users to their models at the application level from conditions ruling the interpretation of the UML diagrams themselves or *metaconditions* (conditions ruling the interpretation of the UML itself). On Entry and On Exit defined in states are metaconditions and some of them are inherent to the way diagrams are laid out and need to be combined to user conditions.

5.4.6 Messages

In object/agent architecture, the dynamics or behavior of the whole system is defined in terms of interactions between objects. Interactions can be described entirely with message exchange mechanism. When an object S (sender) sends a message to object R (receiver), S executes a *send message action* and R also executes a *receive message action*, so the two objects collaborate through this messaging process.

The real purpose of a message depends upon the semantics of applications and is theoretically limited only by our imagination. S can inform R, requests a service from R, wakes R up, fire a chain reaction in which R plays only an intermediate role, asks R to compute a mathematical formula and returns to S the final result, kills the object R with a delete object request, etc.

If there exists more than one typed message between S and R, R must interpret the message from S to analyze the sender request and performs the corresponding action or activity. If we model a message theoretically, we must have at least the following information:

1. *Sender identity*. This information may be used to inform R of the identity of the sender S.
2. *Receiver identity*. This information allows to a message dispatching system to direct the message to the qualified receiver.
3. *Address of the receiver*. If the receiver identity is redundant information, the address completes the identification of the receiver and make his identity unique in the system.
4. *Envelope*. The envelope is the *message wrapper* that allows the message to adapt its format to the format of the media used to convey and deliver the message (for instance, TCP/IP packets in the Internet).
5. *Content*. In programming context, $Content = Service + Parameters$. A service is called a method or an operation at programming level. This information of highest importance is generally embedded in the envelope, sometimes, encoded. The receiver is normally the only object that can read and interpret the content, unless the message is not encoded and broadcasted. In the context of object/agent architecture, this content is the service that R must execute and all the parameters help R to perform correctly its task. So, the content can be split into $Service + Parameters$.
6. *Timestamp*. This information is not intended to the receiver. It is often used to manage the envelope in the messaging media, to determine the lifetime of the message, and to avoid congestion of the messaging media in case of huge amount of nondelivered messages.
7. *Media*. The media used to deliver the message may have various speeds and real-time parameters.
8. *Returned message*. The return message could be everything (nothing, data, object, pointer or reference, etc.) depending on what the sender asks the receiver to do. In the general case, the return data is a new message with the same parameters.

So the format of the message M contains the following data:

$M = (\text{Sender, Receiver, Address, Envelope, Content, Timestamp, Media, Returned data})$
(formula 5.4.6.1)

If we take the US or Canada Post Office, an instantiated message is composed of:

$M = (\text{"Bob", "Mary", "1000, Ocean Drive, CA 92100", protecting envelope, letter inside the envelope, "1 July 2020", ground service, none})$
Constraints: The names of the Sender, Receiver, the address, the timestamp are put on the protecting envelope and the letter is inside the envelope.

Anyone who has programmed in C++, Java, or C# knows that if we want to invoke a function *compute()* declared in an object *Obj* (message receiver), we write for instance:

result = *Obj* . compute (a + b)

If we translate this implementation into the general messaging format, we have a reduced version of message *M* with numerous information removed from the initial format.

Returned message = Receiver . Service (Parameters) (formula 5.4.6.2)

Notice that Content = Service + Parameters

Here is the explanation:

1. The *Sender* is implicit. The call is made from the current program code of the current object, so, the compiler knows implicitly who is the sender so this information is absent from the format
2. *Obj* stands for the *Receiver*
3. *Address* is not necessary as the name of the object *Obj* identifies unequivocally the receiver object in the current small system. If the system is for instance the World Wide Web, an IP address like 132.203.26.21 would be necessary to complete the receiver specification
4. The *Envelope* is always opened in current object programming languages and there is no special protection for the receiver except that *compute()* can be made *public*, *private*, or *protected*. In the present case, *compute()* is public to allow the sender code to call the function
5. The *letter inside the envelope* contains the name of the Service *compute()* and the two parameters a and b necessary to satisfy the preconditions of the *compute()* function
6. *Timestamp* is not necessary in current programming environment unless a special debugging mode is required and a timestamp is asked systematically when entering the call stack

7. The *media* is automatically the Windows, Linux, etc. Operating System, so the media is also implicit and absent from the message specification
8. The *returned message* is now moved to the start of the formula and represents the data returned by the function compute()

Actually, the message model (formula 5.4.6.2) at programming level (implementation or platform specific level) used in this example lacks generality and cannot serve as generic messaging format.

To show the problem of this format, let us consider a citizen that wants to be served in a Ministry. He must emit a request like

Bob . Receive (my_application_form)

This model of requiring a service cannot stand for a Ministry. It supposes that the citizen, when going to the Ministry, can go through all the barriers, reach directly the person who is in charge of the application form, knows that his name is Bob and his role in this Ministry is to receive forms. The Receiver cannot select the appropriate operation to execute

One of the fundamental principles of modeling is to respect the full correspondence between the reality and its model. Thus, to model real world, it is recommended to start with the generic formula (5.4.6.1) and adapt it to specific usages for mapping real messages into the UML.

5.4.7 Relations between Action, Activity, Event, State, Condition, and Message

Through the mapping of fundamental dynamic concepts, we discovered some interesting results. Fundamental concepts are much closed to each other and they are all related. The difference of terminology used comes from the observer viewpoint, where we are on the *cause-effect chain*.

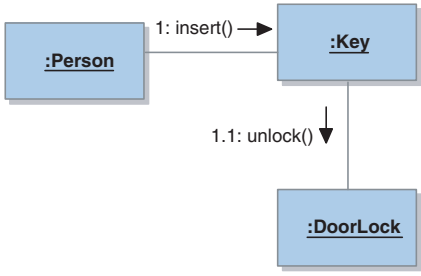
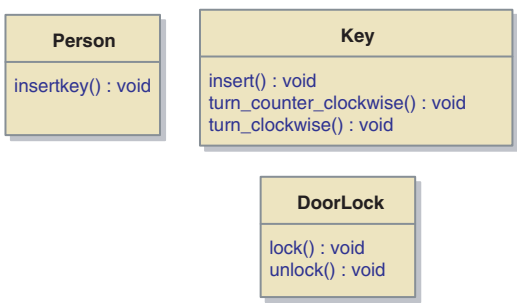
1. *Action and event depends on the observer viewpoint.* When an object S (source) executes an action, the receiver object R perceives it as an event. When a Car A hits a Car B, Car A has done an action and Car B receives it as an event. As action is generally of short time, the result takes the form of an event by its rapid and sudden character.

In the given modeling context, we can find that some actions do not lead systematically to an event.

When a person sneezes, apparently, if we are observing this person, we cannot perceive the “event” character of this act, but, if we are sleeping or if we are in another room, this fact may be perceived as an event. So, an action can give rise to an event or not according to the context.

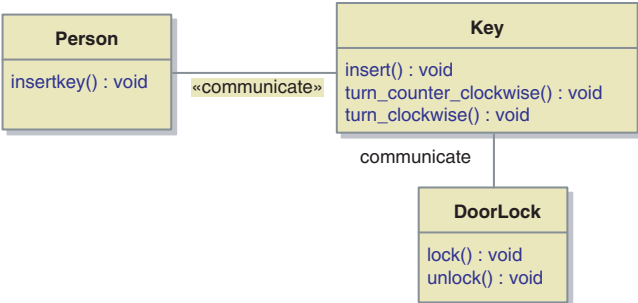
2. *Activities are basic ingredients for building states.* An Activity takes place across a time interval defined by its start and end points. Activities are more structured than actions and we can identify their goals.

Table 5.4.6.1 Support of Message concepts in the UML

Diagram	Comment on the mapping of messages
Object or Communication diagrams	<p>Messages on Link</p> <p>Links in an object diagram are pathways for exchanging messages between objects. <i>Messages</i> are based on some operations defined in classes.</p>  <pre> sequenceDiagram participant P as :Person participant K as :Key participant D as :DoorLock P->>K: 1: insert() K->>D: 1.1: unlock() </pre> <p>In this example, the object <i>Person</i> insert objects <i>Key</i> in the object <i>DoorLock</i> to unlock it. In object world, <i>:Person</i> sends a message <i>insert()</i> to <i>:Key</i> that in turn sends another message <i>unlock()</i> to <i>:DoorLock</i>.</p>
Class diagram	<p>Operations in classes</p> <p>Some operations in classes are designed for supporting communication (all communications between objects are realized through messages).</p> <p><i>Example:</i> Classes in previous example. When object <i>:Person</i> sends a message to <i>:Key</i>, it activates the <i>insert()</i> operation of object <i>:Key</i>.</p>  <pre> classDiagram class Person { insertkey() void } class Key { insert() void turn_counter_clockwise() void turn_clockwise() void } class DoorLock { lock() void unlock() void } </pre>
Class diagram	<p>Class associations</p> <p>Messages are exchanged between objects and objects derived from classes. To show that classes <i>Key</i> and <i>DoorLock</i> manufacture objects <i>:Key</i> and <i>:DoorLock</i> that can communicate together through messages, some designers draw an association between class <i>Key</i> and class <i>DoorLock</i> to mean that their objects may communicate together.</p>

(cont.)

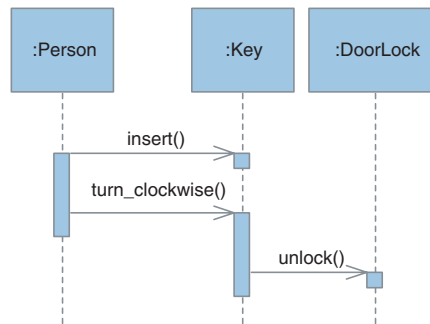
Table 5.4.6.1 (Continued)

Diagram	Comment on the mapping of messages
	<p>From a theoretical viewpoint, the UML accepts that objects are instantiated from classes and links are instantiated from associations, so this way of doing things is acceptable. However, as associations mean relationships between classes, they could create some difficulties for interpreting class diagrams. A stereotype <<com>> or <<communicate>> can be used to differentiate communication pathways from semantic associations.</p>  <pre> classDiagram class Person { insertkey() void } class Key { insert() void turn_counter_clockwise() void turn_clockwise() void } class DoorLock { lock() void unlock() void } Person --> Key : <<communicate>> Key --> DoorLock : communicate </pre>

Sequence diagram

Message between lifelines

In a sequence diagram, messages are represented by solid arrows sourcing from the sender lifeline and ending on the receiver lifeline.



State machine diagram

Actions in states, on entry, on exit, in state, on transition

All actions in a state machine diagram (on entry, on exit, in state, on transition) are potentially messages exchanged between objects.

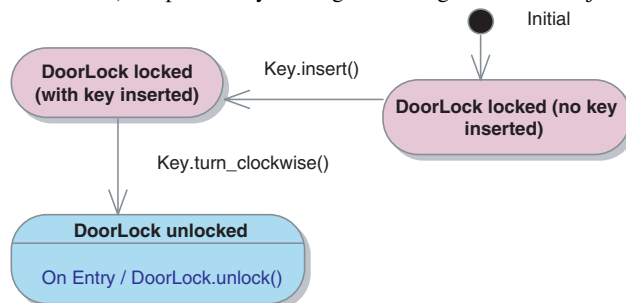


Diagram	Comment on the mapping of messages
	<p>In this example, only state of the DoorLock is represented. Events received by the DoorLock can be named with passive voice like “Key inserted” or “key turned clockwise.” We can make reference directly to actions responsible for generating those events <i>Key.insert()</i> or <i>Key.turn_clockwise()</i>.</p>
Activity diagram	<p>Actions or Activities may send messages</p> <p>Objects are behind “partitions” in an activity diagram. Control flows and object flows are responsible for conveying messages between objects.</p> <p><i>Example:</i> Person sends two successive messages to Key, one for inserting and the other for turning the key. DoorLock receives two messages from Key and performs two actions, <i>receive_key()</i> and <i>unlock()</i>.</p> <p>The representation of the concept of message is oversimplified in the UML as the complete chain is not represented. For instance, the action <i>turn_key_clockwise()</i> executed by the Person is absent from the model. Only, <i>turn_clockwise()</i> is represented as operation in the Key object. We will come back to this aspect in exposing the Uniform methodology).</p>

The time interval between its start and end points may be used to build a corresponding state. If many observable changes are evidenced between these two points, we can decompose activity in subactivities and consequently corresponding substates. An activity may contain several actions (“in state”, “on entry” and “on exit” actions of UML states). If the model needs more precision, transient and short states may be isolated from the main activity for a more accurate investigation.

3. *Actions and activities.* If actions and activities are well defined in the UML, in real world, the term “action” is a strong political term, is

overused so as to mean “reaction,” “position face a situation,” etc. So be careful for the mapping.

A phrase like “The action of the government in agricultural markets is translated into various activities in order to improve the productive efficiency of the whole agribusiness chain” for instance reverses apparently all what was said about action and activity.

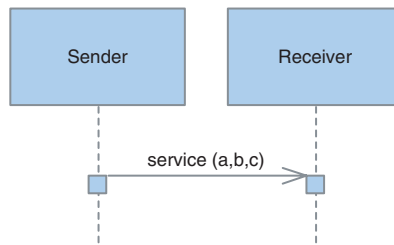
4. *Actions can be considered sometimes as transient states.* Some real time and safety-critical systems are very sensitive to short actions and transient states. In these cases, actions could be candidates for building short but important states.
5. *Activity, state, condition, event are related in a cause–effect chain.* For making tea, we need boiling water. The water receives its energy from the electric heaters. The Heater object performs an activity “Transmit heat to water.” The state of water changes with time and boils after 5 min. The temperature that starts from 20°C then reaches 100°C after 5 min is the most visible parameter. We look at the condition (or state) of the water and decide when the water is sufficiently hot for making tea and decide to stop the heating process.

So, in some systems, people select the most visible parameters to monitor a system and consider them as monitoring values to be tested against some limits. This process actually converts an activity (or action) to state, and then state to condition by the way visible parameters are processed by the observer. Dependent on WHERE we take the information and the way an observer samples and looks at the system, a same reality can be interpreted as activity, state, or condition in a long cause–effect chain.

Now, if the boiler has a built-in whistle, we perceive the condition of the boiling water as an event if we are not in the kitchen. From the object paradigm, we received a message sent by the boiler. A condition has thus been changed to event and message.

This example shows that the way we interpret a phenomenon or fact depends upon the nature of the dynamic interaction chain, the way information is captured and analyzed, where we are on the cause–effect chain, etc. In human organizations, things are more complex as cultural and instructional parameters are added to the batch and influence the observer conclusion.

6. *Actions and activities are executed by objects and messages are data/objects/signals* exchanged between them to coordinate their activities. If in an activity diagram, the object flow is present in the current standard, there could be some difficulties in the way message concept is implemented in sequence and communication diagrams.



For the moment, we cannot, in the sequence or communication diagram, define a message object between two lifelines, showing that two processes (one at the sender side, another at the receiver side) collaborate with this object exchange to evolve. The activity diagram has already implemented an object flow between two activities/actions and is more appropriate for handling this case.

From a theoretical viewpoint, it would be more flexible and natural to propose that the content of the message suggest (or not) to the receiver a given operation for answering his message but it is up to the receiver to choose the most appropriate operation to serve the sender's request. The way object programming languages implements message concept is really not universal and cause distortion with reality when modeling at high level.

5.4.8 Transitions: Relationship with Control Flow

This question very often comes back with students. A transition in a system is often named *system change*, *system movement*, *system evolution*, or any term that accounts for an observable dynamic change.

The UML has an equivalent metaclass bearing the same name that can serve as a direct mapping. *Transitions* are used in the state machine diagrams to show evolution of a system from state to state. In Figure 5.4.8.1, a transition is composed of three concepts:

1. *Event or combination of events* that is the cause of the state change specified by the transition
2. *Guard* that is the internal or an imposed condition that must be evaluated to true to authorize the state change
3. *Actions* that accompanied the transition (*actions on transition*). Those actions are specific to the transition and cannot be affected to any states

Transitions are drawn as arrows connecting boxes representing states. Arrows also indicate the direction of the transition. A state may be represented mathematically by a set of state attributes. A UML transition must therefore change at least one of the state values in order to have a state change.



Fig. 5.4.8.1 Transition characteristics in state machine diagram. Transitions are made of known concepts exposed in previous chapters

One more interesting fact is that states are decomposable. So, state attributes that characterize a system undergo a redefinition at each level of refinement. So, we may have many state changes in lower levels of decomposition and the state at higher level will stay unchanged.

If a Car is characterized only by two global states, “at rest” or “moving”, the “moving” state may be decomposed into “accelerating”, “cruising” and “decelerating”. State attribute at high level is a Boolean “Moving” attribute. At next level of decomposition, we must add another Enumeration “Speed” attribute to make sub state description. When the car goes through three states “accelerating”, “cruising” or “decelerating”, it is in the same high level state “moving”, so at high level, it appears as the system does not change its state.

So, some transitions that occur at low level are hidden from the high-level view of a system, in submachine or composite states. Without entering into philosophical considerations, in modeling, sometimes, “systems seem stable at high level when scrutinizing in more details, variations could be important.”

As said earlier, standard states defined in the UML may contain activities and actions (on entry, on state, on exit). Transitions between states are complex model elements (*trigger*, *guard*, and *effect*). At the difference of transition, a control flow in an activity diagram is a more elementary concept. It is used to make up control rules (fork, join, merge, etc.), to indicate the sequencing order of processes. So generally, transitions are not equivalent to control flows.

Table 5.4.8.1 Mapping of system change, system evolution, system transition, or system movement into a UML transition and UML control flow

Diagram	Comment on the mapping of transition or system change to UML
State machine diagram	<p>State transition</p> <p>System change, system evolution, system transition, or system movement in real world are mapped into UML transitions in a state machine diagram. <i>Trigger</i> explains the cause of change. <i>Guard</i> explains the condition of the system that allows this change. <i>Effect</i> describes action that results directly from this change.</p>
Activity diagram	<p>Control flow</p> <p>Control flows in activity diagrams are used to indicate the movement of tokens (activity marker) between activity nodes. Activity nodes can be actions, activities, and control nodes (fork, join, etc.). So, control flows in an activity diagram can show system change or evolution as well.</p>

5.5 Fundamental Concepts of the Structural View

Structural view describes systems as a collection of objects, their characteristics and relationships. The structural view was qualified in the past as *static view*, by opposition with the dynamic view that describes essentially the evolutionary aspect of a system.

In the domain of real-time systems, the vast majority of software developments are PSM. Notions like *reuse*, *logical model*, and *PIM* are not widespread; we often found in a structural description objects specific to applications, disseminated among objects like computer, operating system, software components, deployment nodes, and even hardware mounting devices, were all mixed together in an inextricable manner.

In the domain of database development, things were better organized. Conceptual data models made with E-R tools tend to behave like PIM (however, not all PIM are conceptual data models). Conceptual data modeling is the first stage in the process of database design. Traditionally, it identifies *entities* (persons, objects, data, etc), their attributes (information that describe these entities), and relationships that link entities together into a network that gives guidelines for developers to interpret the model. If we compare the conceptual data models with the object structural view, the only difference seems to emerge only from the replacement of the terms *entities* by *objects*.

However, there exists a deep difference between the E-R model (Chen, 1976) and the object paradigm. Maybe, the semantic models (Smith and Smith, 1977; Hammer and McLeod, 1981), which provided a modular and hierarchical view of data, can be considered as precursors of object data model. The primary components of semantic models were the introduction of the *ISA* (“is a”) and *aggregation relationships*. *ISA* relationship (e.g. a square or a triangle *is a* polygon) is equivalent to *object inheritance* and the *aggregation relationship* has its equivalent *aggregation/composition* in object model. The composition relationship is the basic architectural concept that supports complexity reduction mechanism and allows a system to manufacture composite objects or components from smaller objects and thus makes all systems manageable at any level.

However, semantic models are not object models. Essentially, semantic models encapsulate only structural aspects of objects, whereas *object paradigm encapsulates both structural and behavioral aspects of objects*. Moreover, E-R models lead to relational models that, in turn, are implemented with *tables of values*. Tables are value based or value oriented (Khoshafian and Copeland, 1986; Ullman, 1987). Objects are identified by their *surrogates* (system-generated globally unique identifiers) rather than their values, objects have OID (Object Identity) and do not need to create value based strings to serve as primary keys for indexing.

To explain the value based problem for identity, in E-R modeling, the class Person is for instance identified with a specific string (its NAS or National Assurance Number). Codd (Codd, 1970) called this attribute “user-defined identifier key”. There are several problems with identifier keys as the concepts of data value and identity are mixed together. If a value is served to identify an entity, people cannot change this value easily as this change may affect all dependent applications based on this identification. For instance, if a database encodes the department name “ABC” in the product identification “ABC12345678”, if we change the department name “ABC” for “DEF”, all products that have an encoded identity must be updated to reflect the new name change. So, values interfere with identities and add substantial data manipulation operations.

In an object system, it is not necessary to have a special attribute for distinguishing one object from another. A surrogate is automatically affected by the system, doubled by a more easily reminded identity proposed by the user for local and short term use. If surrogates and their synonyms are unique at object creation, they may serve at runtime as object identity. If this identification method based on surrogates avoid the previous problem of value-based identity, there could be some difficulties in a distributed environment (Leach et al., 1982), for instance in the Internet. Many solutions are available to handle distributed databases on the Internet and we will come back later in the “object identity” chapter.

Object technology places the *encapsulation* of attributes and operations and *class inheritance* among its natural and basic characteristics. When digging inside inheritance, once more, inheritance in object technology is deeply different from inheritance in semantic models as it encompasses behavioral inheritance (presence of object operations) instead of inheritance of attributes only as observed in semantic models. For detailed information and a survey paper about semantic models, please consult Hull and King (1987).

To build the structure of an object system and later provide a description of its structural view, we need to consider the following points:

1. *The adhesion to the reuse concepts* (see Section 2.8) will structure differently the application.
2. *The identification of application domains* being developed and the adhesion to ontologies defined on these domains give us a rigorous description and a coherent vocabulary. If ontologies do not exist, a small glossary with precise definitions would be an advisable minimum.
3. *Developers must be convinced of the importance of models and MDA guidelines* that partition the development process into activities of building PIMs and PSMs. The development is therefore a multistep or multidimensional process. When decomposing a PIM, complexity reduction and decomposition still respect PIM principles and must not end up with platform-specific components. A description of a structural view is said to be model-based in this case.
4. *It is advisable to model even the PSM in the form of the UML model* instead of the running code, despite the fact that the PSM may most of

the time contain the same information as its implementation counterpart. So doing, automatic code generators can take UML models and translate them into code with interface definition files, configuration files, make-files, and other file types. If generators are not available at the developing time, we must handcraft a code, however, we still have a good model for the future, so the development is said to be *future proof*. The major trend in software engineering adds one or several layers of middleware, so possibly there could be more than one PSM layer.

Middleware is the “glue” between software components or between software and the network (Internet or Local Area Network), and roughly the “slash” in a Client/Server architecture. In today’s corporate computing environment, many applications have to share data. Putting middleware in the middle means each application needs only one interface to the middleware instead of separate interfaces for each connection. Middleware can keep captured data and hold them until they are processed by all applications or databases. This capacity of “buffering” adds another dimension to middleware. Middleware add several services needed nowadays in most distributed environments (checking data for integrity, printing out, converting data, reformatting, etc.). From a functional viewpoint, middleware allows multiple computers to do multiple things across a network and allow one computer to do many things or one complicated thing across a network (link a database system to a web server across the network, allow users to access the database via a web browser, link multiple databases in multiple servers, online games, etc.) Middleware can be a single application, or it can be an entire server. Middleware are often invisible or transparent to some categories of developers. Well-known middleware packages in the Internet Domain include the Distributed Computing Environment (DCE) and the Common Object Request Broker Architecture (CORBA). There is another marginal trend that considers middleware as an adapter device (for instance connecting a new printer to an older computer. In this case, middleware are not reserved only for web or distributed applications.

5. *Nested classifiers* are an extremely powerful concept for structuring object system. In the UML, almost every model building block (classes, objects, components, behaviors such as activities and state machines, and more) is a classifier. So, we can nest a set of classes inside the component that manages them, or embed a behavior (such as a state machine) inside the class or component that implements it. This capability also lets us build up complex behaviors from simpler ones. For example, we can build a model of an Enterprise object, zoom in to embedded site views, to departmental views within the site, and then to applications within a department.

6. *Use of inheritance relationship when appropriate.* The major advantage of object-oriented programming is the ability of classes to inherit the properties and methods of their parents. Inheritance enables to create objects that already have built-in properties and functionality.

5.5.1 Class, Type, Object, and Set of Objects

5.5.1.1 Object. In an article dated from 1987 Wegner quoted:

An object has a set of operations and a state that remembers the effect of operations” Objects may be contrasted to functions which have no memory. Function values are completely determined by their arguments, being precisely the same for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience, its reaction to an operation being determined by its invocation history.

This definition reinforces concepts used in pioneer languages like SIMULA and SMALTALK. Object is implicitly an entity, has operations but cannot be assimilated to them (operations and functions are synonyms in this definition). Objects have states through the execution of their operations, may learn from experience (this fact draws objects towards common characteristics of “agents”) and actions deployed by objects are not of combinatorial type but have a “historical” component. So, the reaction of objects depends upon current inputs and experience learnt from the past. Objects therefore own some kind of memory and can use them to elaborate new behavior.

Passing in review all the definition of the term “object” in the literature will be a very annoying task, if not controversial. In fact, the definition of a concept is a complex task that is dependent of many factors:

1. The intended *audience*. Cultural and education level may influence the definition
2. The *domain* of interest
3. The *view* adopted inside this domain.

For instance, while modeling, an activity in a dynamic view may correspond to an operation in the structural view. So, terms used will change accordingly

4. *Where the concept is located* in the ontological architecture built on the current domain. Surprisingly, an elementary concept is more difficult to define than a complex one. Very often, the definition of elementary concepts leads to circular use of elementary terms.
5. The *purpose* of the definition. More insidiously, if a definition must fit into an existing system, sometimes, the definition must be adapted to the context where this definition must be inserted, and in this respect, the result can depend upon the purpose of the exercise. The problem is currently referred as the contextual interpretation of terms.

6. The *validity range* of the term to be defined

The definition of the term “object” is very difficult as an object can be very elementary object as a passive “piece of chalk” used by the instructor and at the opposite end, it can be the whole earth seen from the moon, passing through social object like agents, or abstract objects like our mind or human memory system.

The term “object” already defined in the Section 2.4.1 is closed to the definition given by Wegner. It includes agents, humans, organizations, etc.

5.5.1.2 Classes, Attributes, and Operations. The notion of class is ubiquitous in science and is central to the representation of structured knowledge and database development. Wegner (1987) defined class as:

A class is a template (cookie cutter) from which objects may be created by “create” or “new” operations. Objects of the same class have common operations and therefore uniform behavior. Classes have one or more “interfaces” that specify the operations accessible to clients through that interface. A class body specifies code for implementing operation in the class interface.

Even this definition is closer to the implementation model; the main idea is that a class is a kind of mold for creating objects that have common characteristics and behavior (objects do not have all identical states at run-time). Class is not a “set of objects” but just a *mold of creation*.

We can have two switches *SW1:Switch* and *SW2:Switch* issued from the same class *Switch* in a same system and SW1 is On and SW2 is Off.

But, the notion of class and classification in real world goes beyond the concept of mold for object creation and code sharing in object-oriented programming languages. Classes are often considered as results of the classification process. To simplify object description, we can see the *classification* as the process of mapping objects into categories, using some *classification rules*. Rules are based on *classification attributes* values.

There are several kinds of attributes in a class that need comments:

1. *Classification attributes* or *global attributes*. When we make a class Person, a class Sensor, a class Alarm_Detector, etc. to solve a real-time problem of home security system, we make our own “user classes” or “application classes” in the UML and suppose that all classes of our application are already there and magically “classified” by our imagination or design skill. In fact, it appears as the whole world has all its real classes already made, each corresponds to a real object or a conceptualized object. If somebody asks us the question “How are classes made and on what basis?” we could be very embarrassed. At this time, we look for some criteria that may help us to justify the way we make our own classes at the

application level. The most obvious solution is “taking an enumeration variable and affecting values as Person, Sensor, and Alarm_Detector to it.” Even if concepts are not well explained (because we do not have to explain evidence), classes are there, differentiated by value of an enumeration type. If nobody asks us embarrassed questions, the mentioned attribute is “invisible.” It is a *classification attribute* or a *global attribute*. The term “global” comes from the fact that we suppose that the whole humanity agrees with us on our classification.

In applications like pattern recognition, text recognition, biology, oceanography, biometric parameter matching, ontological development, etc. where we must take objects, analyze all their characteristics, classify them into categories, classification attributes are not “invisible” but must be defined and justified by the classification process.

So, classification attributes or global attributes are used to classify objects of the world or more restrictively in a given domain of interest. Classification attributes are “invisible” for people who develop from scratch systems with the UML.

2. *Local attributes or application specific attributes.* Local attributes are defined inside an application. If a class Person is defined in a database application with attributes like *firstname*, *lastname*, *age*, etc., those attributes are local to the application. Local attributes are partitioned into structural, functional, or dynamic attributes and can be defined freely for modeling the application.

Firstname, lastname of a class Person is a structural attributes. Functional attribute may affect for instance the availability of operations defined in this class and condition the way they must be activated or executed. For instance, some mobility functions cannot be activated for a handicapped person (handicap attribute). Dynamic attributes are defined in the context of the application to store states of the system (e.g. a home alarm system may be found, when powered, in the following dynamic states: *idle*, *intruder_detected*, *alarm_on*, *central_calling*, *phone-line_occupied*, etc.). Some attributes are hybrid, for instance, the age of a person is naturally structural, but, at some high values, can inhibit some of his functions, for instance *drive_car()*.

The UML defines itself meta-attributes (for instance *isComposite* for a state). Meta-attributes dictate how model elements are represented or interpreted in a diagram and is not concerned directly with the semantics of concepts we want to model.

Objects works through their *operations*. Operations are called *methods* or *functions* in object language programming. Operations are modeling concepts that represent processes or tasks in the real world. Hereafter are some characteristics of operations:

1. *Visibility.* The visibility of the operation or its access rights show who can invoke operations.
 - (a) The UML defines four default visibilities:
 - Private (“-”): Only the current object can invoke the operation
 - Protected (“#”): Only object of the current class or inherited classes can invoke the operation
 - Public (“+”): Any object can invoke the operation
 - Package (“~”): Only classes within the same package (namespace, container) can invoke the operation
 - (b) As visibility is defined as an enumeration, we can go beyond the default number and define another visibility. If more complex visibility is required, we can always put a “public visibility” frontal service desk, make verification and redirect or reject the request and, in doing so, achieve any form of visibility.
2. *Operations may be of any complexity.* The complexity of operations derives from the complexity of the wide range of objects.
 - (a) Starting from a most simple private operation that changes only the value of an internal attribute (e.g. a clock that changes at each second its internal time), we can activate a complex operation of an agent who must realize a mission. During this mission, he interacts with other agents by activating their operations and so on.
3. *Operations parameters.* To execute any operation, an object needs data obtained from local attributes, data coming from external sources, or in complex situation, it must request data from another objects, different from the invoking object
 - (a) If we ask an agent to compute the conversion of 100 CAD into US currency, we must give, with our request, the value 100, and invoke the operation *convert()* of this agent. This agent must eventually request data from another agent to get the most up-to-date conversion ratio at the moment where our request arrives.
4. *Operation returned results.* The result returned by an operation can be of any complexity.
 - (a) The model of programming languages let us return only a simple value (integer, real, string, etc.). In a complex case, we can return a pointer to an object or a pointer to any arbitrary complex structure.
5. The call to an object can be synchronous or asynchronous. Generally, when invoking a service from another object, the source or caller object

will lock itself and wait for the returned result. This kind of behavior is qualified as “synchronous” in the UML.

- (a) In an “asynchronous” call, after asking the service of other objects, the caller will continue with its own tasks without waiting for the returned value. As the caller may make many calls and the returned results may come back to the caller in any order, the returned results are therefore considered as new messages and must be identified correctly with a full message content and envelope. Effectively, in stateless protocols like HTML, the second scenario prevails.
- (b) The previous example describes the software implementation model. In real world, we can have more complex situation to model. There can be no bound on the amount of time that can elapse between process steps, no bound on the time it can take for a message to be delivered. Semi-synchronous model of communication is a synchronous model that set up the upper and the lower bounds for time. This model serves often as a basis for real-time reasoning.

The number of slots defined in a class is usually 3. The first slot is reserved for storing the *class name*. The second slot groups *local attributes* defined by designers and the third slot gathers all *operations*. The metamodel does not fix any number of slots, so user-defined slots are expectable feature in UML tools for organizing various categories of attributes and operations.

5.5.1.3 Type. Type has been extensively studied in Computer Science Domain (Cardelli and Wegner, 1985) (Nierstrasz, 1993). Type is a complex issue and the meaning may vary considerably with domains. Common meaning defines type as a “category of being”. In this sense, typing is classification.

Human is a type of things and Animal is another type of thing. When basic types are defined, subtypes can be added to enrich the tree structure commonly used in classification.

The type verification with this common meaning needs a more precise definition with multiple criteria that encompasses *both structural and behavioral aspects* of objects. If everything is well defined, normally we can make type verification.

In computer language domain, when programming, we manipulate datatypes like integer, real, string, double, etc. They are primitive datatypes and their values ranges are well defined. Linear data structures (arrays, lists, etc.), graph data structures (B-tree), heterogeneous structures (structure, union) can be built from primitive datatypes.

Starting from the simplest definition, a *type A* is a set of all values that has that type A. *Int* is a set of integers. *Float* is a set of IEEE floating point values coded with 32 bits. To say that a particular variable is of type A, we write

a : A

By extension to record types, we write

$$\{a_1, a_2, \dots, a_n\} : \{A_1, A_2, \dots, A_n\}$$

Functions are also variables but functions need arguments, so the type of a function f is its extended signature (see Section 4.6.4) that comprises the type of all arguments P plus the type of the return value V .

$$f : P_1 \times P_2 \times \dots \times P_p \rightarrow V$$

Languages in which all expressions are type consistent are called *strongly typed* and a compiler can guarantee that programs accepted by it will execute without type errors since verifications at compile time can discover inconsistencies. This verification at compile time is called *static typing*. Static typing may lead to loss of flexibility by constraining objects to be associated to a particular type. Dynamic typing allows datatypes to be changed at run-time.

Now, what is the type of an object?

An object type is that which can be seen from the outside, so we must take into account only the attributes and operations that are declared *public*. If A is used as generic letter for attribute type and O generic letter for operations, the type of an object comprises n attributes and m operations (each operation type is itself another record type of its parameters and return value).

$$\{A_1, A_2, \dots, A_n, O_1, O_2, \dots, O_m\}$$

The static type verification so defined and based on the type of all public local attributes and the extended signature of all public operations is a low-level process. If two objects passed the type verification, we can ascertain that they belong to the same class if in our local system. If there is no collision (declared objects that bear the same appearance but their interfaces are not built with the same classes). Moreover, this type verification does not verify anything about the behavioral aspect of objects. We can have two operations that take no parameters, which return *void* (nothing) value, but behave differently and finally do not come from the same class. In a global or distributed system, the collision of two objects that have the same appearance is not negligible.

To summarize, an object type is a record type of its public attributes and operations and the object type verification cannot say in all cases that they are necessarily issued from the same molding process. This verification is nevertheless the first step for any identity checking.

An object-oriented system is not necessarily class based. We can work with objects in a system that does not contain any definition of class concept. Systems can be built just by composition of objects. A car is made of a composition of tens of thousands of mechanical pieces. We can make a new object by designing it from scratch or by taking an old object and then adding by composition some new attributes and operations.

To summarize, the way the notion of types was defined in programming languages is not universal. With systems having the class concept, classes play the role of molds (of creation). *Object type* can be seen from class interface (the term “interface” used here is more general and does not mean only UML interface) with its overall signature or record type. A class contains private attributes and private operations, so *a class is not always a type*. They are fundamentally different concepts as a mold is not the appearance of the object it molds. When discussing about inheritance, inheritance or subclassing is not subtyping.

5.5.1.4 Set of Objects. A set is a collection of objects, usually with some common properties. For instance, we can divide a class of students into the set G of all girls and the set B of all boys. We can also divide the same class of students into three sets A-F, G-N, and O-Z, consisting of students whose first letters of names are, respectively, in those intervals. We can combine sets together and find all girls with names beginning from O to Z.

A set of objects is therefore a collection that shares some attribute values in common. A set is not a class, because a set is a collection of instantiated objects and a class is a mold. The mold is unique but the cardinality of the set is different from 1. However, there is an interdependence of class and set of objects in the way classes are defined in an application as illustrated in the following example.

If we make a class Car as shows in Figure 5.5.1.1, we can instantiate a set (or group) of cars manufactured in USA, other sets from Japan, Korea, Germany, etc. The class Car, as defined, can be used for registering all cars that are in circulation in California State.

Now, let us consider the Car class created by a manufacturer of US cars. He is not interested to make a class with local attributes *origin_country* and *constructor* as all values would be initiated by default to “USA” and “GM” (if the constructor is e. g. GM). Normally, he must name his class “US_Car_from_GM” but he decides to call his class Car to simplify things.

Now, let us consider a vendor of “made in US only” used cars. He must declare a class Car with *constructor* attribute as he must differentiate between GM, Ford, Chrysler, etc. If he changes his mind and decides to sell Japanese cars together with US cars, he must reengineer his database and insert *origin_country* as local attribute in the definition of his Car class.

The possible ambiguity is the fact that the dealer does not call his class “made_in_US_Car”, and the manufacturer does not call his class “US_Car_from_GM” (This example is hypothetical only since GM may decide to create all attributes, for instance to record a GM car made in Japan). In fact, there is implicitly a transfer of local attributes to the block of global or classification attributes.

“US_Car_from_GM” is a subset of “made_in_US_Car” that is itself a subset of “State_Registration_Car” that is itself a subset of “Generic_Car”. Subsets based on values of attribute are often isolated to create independent classes. This fact is

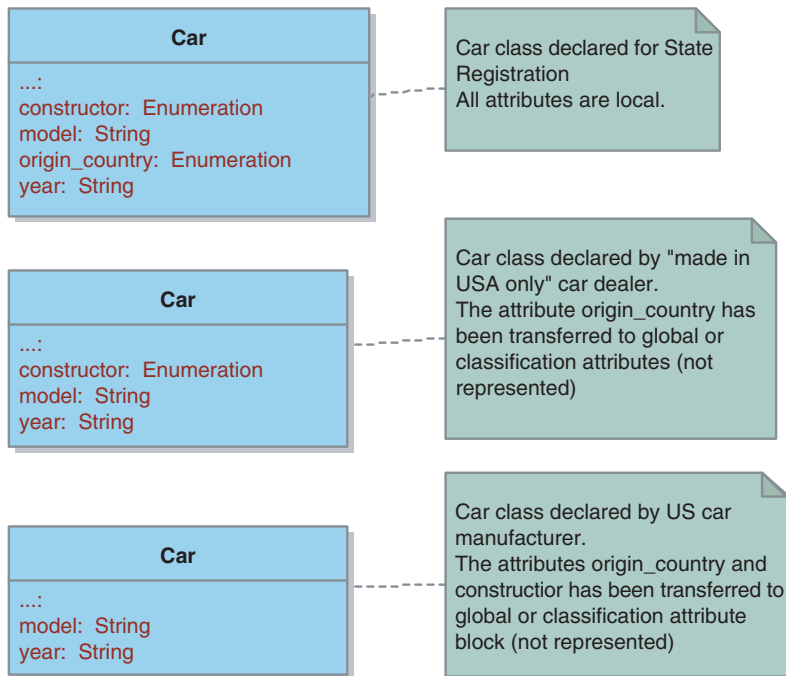


Fig. 5.5.1.1 Car class declared in three different contexts

often observed in real systems. Dependent on the number of enumeration values, this fact could create a very rich set of classes disorderly named if we look at the problem globally. But each local database is still consistent if no data merging, no data fusion, no data cross referencing are necessary.

If we consider global attributes and local attributes as a whole, the number of attributes has not changed but according to the context or the domain, there is a *bilateral transfer of attributes between the global and the local blocks*. This transfer has an impact in the classification mechanism as we can make theoretically any class based on any combination of values of attributes. This fact explains the subtle relationship between sets of objects and classes.

5.5.1.5 Abstract Class: Difference with Interface. Abstract class is defined as a class that cannot be instantiated to create objects. Why do we need them?

Food? What is an instance of food? So, food is an abstract thing that we can eat. But we can derive salad, orange, apple from food. So, abstract class Food is useful as an abstract element in the hierarchy of food for structuring a global description.

Abstract classes are very useful in defining domains and ontologies, as they are semantic nodes or semantic reference points. A class becomes abstract if it

has at least an abstract attribute or an abstract operation. An abstract operation is an operation declared only with its extended signature without any implementation. Many concrete classes can be derived from an abstract class and each concrete class may implement a different version of an abstract operation.

There is traditionally a difficulty between the notion of abstract class used mostly for classification and the concept of *interface*. The UML defines interface as:

An interface is a kind of classifier that represents a declaration of a set of coherent public features and obligations. In a sense, an interface specifies a kind of contract which must be fulfilled by any instance of a classifier that realizes the interface. The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and post-conditions) or protocol specifications, which may impose ordering restrictions on interactions through the interface. Since interfaces are declarations, they are not directly instantiable. Instead, an interface specification is realized by an instance of a classifier, such as a class, which means that it presents a public facade that conforms to the interface specification. Note that a given classifier may realize more than one interface and that an interface may be realized by a number of different classifiers.

“Public features and obligations” are operations. An interface is a named collection of operation definitions, without implementations. The differences between an abstract class and an interface are very significant:

1. An interface cannot implement any operation, whereas an abstract class can implement some common operations for all derived classes
2. An interface is not part of the class hierarchy
3. Classes derived from an abstract class share commonalities whereas interfaces can be built from completely unrelated classes.

5.5.1.6 OID, GUID, and UUID. OID has been investigated in the past (Khoshafian and Copeland, 1986). Identity is a property that distinguishes an object from all others. Object identification is a general problem in real life (complicated names, collisions, etc.) and is not specific to computer sciences. If in a local system, we can adopt a strategy to name unequivocally things, modern communication networks need a well-designed identification in order to have a global and unique ID (GUID) and at the same time a shortest ID, avoiding communication delay.

When we name a directory of our file system, we are facing the problem of object (directory) identification. Every path taken from a computer must be unique, for instance *C:\data\book\chapter1*. If we have a second computer connected to the first one and the file directory contains the same name *C:\data\book\chapter1*, we are facing a collision problem of names. Happily, standard operating system has resolved this difficulty by adding, for instance, the name of the second computer to the directory name, e.g. *\\forgetmenot\ DiskC\data\book\chapter1* (*forgetmenot* is the name of the second computer and *DiskC* is mapped to *C*

on the local computer). So doing, we don't have to rename all directories of the second computer to be able to see those two file systems at the same time.

Internet is a global communication medium. Any server in the net is located with an IP address composed of four fields starting from 000.000.000.000 and ending at 255.255.255.255 allowing more than 4 billions servers to be identified in the Internet (some addresses are however reserved). As this number is hard to remember, an equivalent textual address is used to map a server address to a more easy retained character string. For instance, `www.omg.org` is translated into 192.67.184.5 (just emit with the DOS shell a "ping `www.omg.org`"). `www.omg.org` is a reserved and protected string (as a name of a company), but the server underlying this string address may change every time. If OMG wants to shut down their server for maintenance, technicians can map temporarily `www.omg.org` to another IP address.

OID has been investigated independently in object programming languages and in database management, either relational or object. In the past, object practitioners claimed that they do not need to create an artificial key to identify any record since the object system maintains itself an OID, so their system was automatically indexed. Before discussing this "advantage" let us come back to the original problem.

Identity is normally invariable and is independent of whatever can change with the object (value, properties, location, etc.). The trend towards giving a global identity to every object come from the fact that all networks in our modern world tend to be fully interconnected and there is a real need to identify objects globally in a distributed environment. We thus need a kind of universal ID called GUID (pronounced "gu-ID") by Microsoft. The GUID is also known as Universally Unique Identifier or UUID. GUID is ID that is guaranteed to be unique in space and time. An object with a GUID can be replicated in many servers for optimizing the access time but remains invariably the same object (the object coherence problem between copies must be solved in this case but it is another problem).

Object systems and object databases, by their nature, have a low-level identity generator designed for handling the object system. They can handle true clones (in an object system, we can have two "John Smith," same firstname, same lastname, same age, etc. but recognized as two different persons without any artificial primary key). But, this identity is not always accessible and the way it is encoded shows that it is a "machine" identity, not really human readable. So, from an application viewpoint, sometimes, a given identity, easily remembered (phone number, NAS, etc.) is still very useful in everyday life systems. So, the presence of the two identity systems, one low and one high level, is probably an optimized solution.

Some relational databases make use of surrogate keys that have the same property of a low-level ID in object context. The two worlds are connected through this initiative. Surrogate keys provide a number of advantages. The most significant one is that this key needs never to change. We can now easily

change the NAS attribute of a Person in such a context. The surrogate key in relational world has its equivalent OID in object world. They are system-generated keys. At higher level, the “business key” or “user-defined key” in both worlds bears some semantic for human beings.

5.5.2 Database Objects, Real-Time Objects, and Their Relationships

Real-time objects are typically full-featured objects with:

1. *Identity*
2. *Structural attributes* that describe them
3. *Behavioral attributes*, so objects have states
4. *Operations* that make objects dynamic workers

The construction of a real-time object system is typically based on two important relationships:

1. *Ownership or aggregation/composition*. Objects are hierarchically structured in an object system, Objects and composite objects collaborate together to accomplish complex tasks.
2. *Inheritance*. Objects are classified and new classes can be defined from existing classes, in other words, it is possible to incorporate structural and behavioral features of existing objects into newly created objects.

Unless a database is used, for instance, to record the evolution of a scientific experience (in this case, behavioral attributes are relevant), records in relational databases describes only structural attributes. Each record is a list of values. To make a correspondence with real entities, a given identity is fabricated to mark each record unequivocally. So, relational database is concerned only with points 1 and 2 of the preceding list of four points.

If our system describes for instance all the workers of a very large hospital that include patients, medical and administrative personnel, we can study how fast this hospital provides services or some targeted operations of this hospital by defining and simulating systems with patients, medical doctors, nurses, and so on. At the same time, this hospital must keep records of thousands of patients, data of its personnel. We have therefore a real time system and simultaneously a database system that may share common structural attribute values.

Traditionally, for the database, the data model can be established with an E-R (entity-relationship) diagram and implemented with a RDBMS. User can access to the database by using SQL (Structured Query Language) in two steps, the first step is to define the database structure based on E-R schemas, and the next step is to query this database to obtain information. Between the two, the database must be populated with data.

The real time simulation can be developed with an object programming language (C++, C#, Java, object Basic, etc.). Objects will be created in the simulation program with classes. Some classes bear striking resemblances with entities in E-R diagram if we consider structural attributes.

Now, how to make those two systems developed in parallel, to communicate together, i.e. allowing object programming to access relational datastores (for instance, if an object in this context has a long list of structural attributes, it would be natural to desire to store those attributes in the database, and use an object with the same identity in the real time system populated with behavioral attributes, having some kind of connection with the object structurally described in the database). There are known modern methods as JDBC (Java Database Connectivity) or ADO (ActiveX Database Objects) that make the glue between two independent systems. From the programming platform, SQL requests can be sent to RDBMS and answers are wrapped then returned to the requester. The method is slow but direct access implies some form of mapping between an object system and a system based on tables of values (O/R or Object Relational mapping). Some tools in the market are targeted to solve this issue but generally programmers prefer to hand code database access through JDBC or ADO.

One of the crucial questions is the opportunity to use objects as the front end, at the design phase, for developing relational databases, replacing the E-R diagram by the UML class diagram. The mapping is nearly straightforward as each entity can be replaced by a class and all attributes or fields in the E-R diagram are mapped into class attributes. Relationships in E-R diagrams are converted into associations in a class diagram that fully support all concepts (roles, multiplicities, navigability). Attributes of relation are converted to class associations. Inheritance or aggregation/composition is translated into tables as well. If the object model are well structured, the resulting structure issued from tables is blurred and information are got by an expensive table joining process to find things that need merely some navigation hops in the corresponding object implementation model. Nevertheless, the mapping is possible, the implementation is rather awkward but it works. So, in waiting for a better proposal widely accepted, we can use the object model as the front end, even if the implementation is purely relational. For more information on the mapping, please consult (<http://www.service-architecture.com/>).

The last effort made by the database community to adhere to object paradigm was the famous Object RDBMS (ORDBMS) that extend relational database systems with object functionalities to support a broader class of applications. This concept bridges the relational and object-oriented paradigms. SQL 1999 was an enhanced version of SQL intended to support this extension. ORDBMS was created to handle new types of data such as audio, video, and image files. One advantage of ORDBMS is its compatibility with core RDBMS, so organizations can continue using their existing R-systems while trying new O-systems in parallel.

In summary, it would be unwise to base everything on the small market share of the ODBMS (about 1% in 2005 of the whole database market) and hastily conclude that the object database has marked its end. It is still, at the start of the new century, an immature technology but the object paradigm is omnipresent and has let its marks everywhere in the database industry during the last two decades. Today relational databases are able to store complex data structures (BLOB: Binary Large Object; CLOB: Character Large Object) and the ORDBMS is present as a historical witness of the object footprint.

5.5.3 Dependencies, Relationships, Associations, and Links

Terms like “dependency, relationship and association” are defined in the UML with their metaclasses. They are compared to more common meanings.

Relationship

UML: Relationship is an abstract concept that specifies some kind of relationship between elements (Relationship metaclass from Kernel package)

AskOxford: 1. The way in which two or more people or things are connected, or the state of being connected. 2. The way in which two or more people or groups regard and behave towards each other.

Dependency

A dependency signifies a supplier/client relationship between model elements where the modification of the supplier may impact the client model elements. A dependency implies that the semantics of the client is not complete without the supplier. The presence of dependency relationship in a model does not have any runtime semantics implications; it is all given in terms of the model elements that participate in the relationship, not in terms of their instances.

Cambridge Dictionary: a state of needing something or someone, especially in order to continue existing or operating.

Association

An association specifies a semantic relationship that can occur between typed instances. AskOxford: “a mental connection between ideas”.

Link

UML: A link is defined in UML as an instance of an association.

Connector

UML: Specifies a link that enables communication between two or more instances. This link may be an instance of an association . . .

As graphical notation, we read A connector is drawn using the notation for association.

In the current language, *relationship* and *link* are often taken as synonyms. *Connector* sounds hardware and is mainly used for electrical connections or graphical link between graphical objects. *Association* is used to relate more

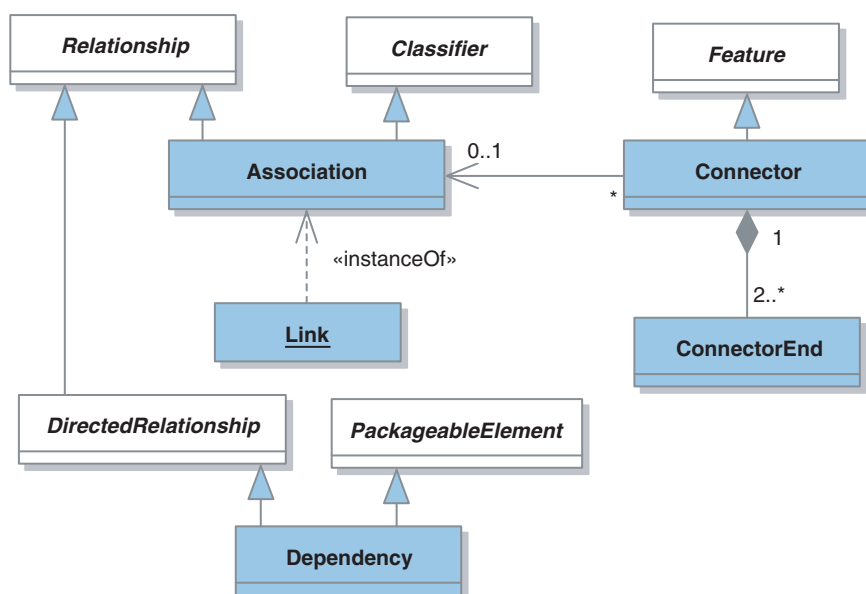


Fig. 5.5.3.1 Relationship between Relationship, Association, Link, Connector, and Dependency

abstract concepts. *Dependency* depicts an asymmetrical relationship between humans or things.

In the UML, the classification is more precise so the mapping will depend upon the meaning we can deduce from real contexts. The Figure 5.5.3.1 summarizes the relationship between those concepts in the UML. In this figure, *relationship* can be used as a general term as there is an abstract (noninstantiable) metaclass attached to it. So, *relationship* is the most elementary brick in this hierarchy. *Connector* is just a feature but it can be related to something more important as an association.

Association is a richer concept as it derives from both *Relationship* and *Classifier*. This view complies with the common meaning of this term in real world except that association is used at the class level in the UML. A link is an instance of an association and links connects objects (instances of classes) together. As, there is no link metaclass, association may be used both to link classes or objects, even objects to classes.

Primarily, *connectors* represent interactions among objects and provide communication paths between two or more instances (parts, ports, etc., see Section 4.9.5). It is used mainly in a composite structure diagram.

Dependency is a directed relationship and can be used to enslave a model element to another element. The exact definition of this dependency must be specified by a constraint or a significant name of this dependency.

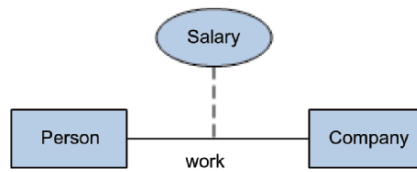


Fig. 5.5.4.1 Attribute of relation in E-R diagram

5.5.4 Association Class: N-Ary Association

Most relationships connect two entities together in an E-R diagram. The classic example in the literature is the attribute *salary* that connects an entity *Person* to another entity *Company* (Fig. 5.5.4.1). As the person works for the company, he earns a salary through this employment. The attribute salary cannot be an attribute of a *Person* or a proper attribute of the company. According to the qualification of the person, the salary may vary, so this attribute will depend upon the nature of this relationship.

If we map now this E-R diagram into a class diagram, orphan attributes are not tolerated in object world (each attribute must have a host object), so a class association replaces the attribute of the relation in this context. This class is instantiable, so it must be qualified for an independent existence. Actually, when we think of how human relationships are interweaved, very often, we run business by contracts (Andrade and Fiadeiro, 1999), the *Person* and the *Company* agreed on a *Job* contract. The job has a proper existence as independent class as the company may publicize the job to find a candidate. Later, when a person accepts the job, the job contract consolidates the relationship between the person

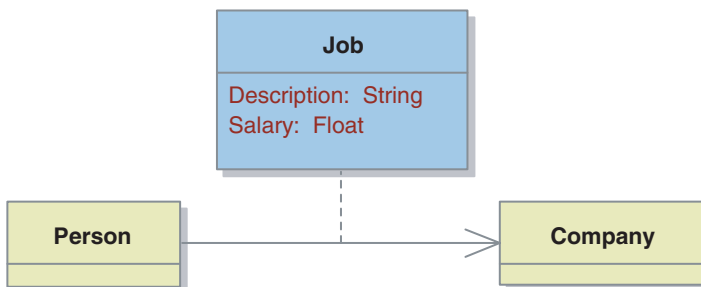


Fig. 5.5.4.2 Association class. When a person takes a job in a company, the two sides are tied by a contract that is represented as a class association. We can instantiate the *Person* alone, the company alone, and the *Job* alone. If the person decides to take the job, a contract must be signed with the company

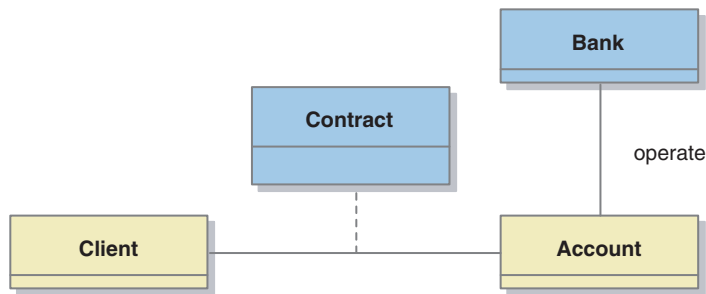


Fig. 5.5.4.3 Association Class. Association class represents a contract in business domain

and his company. The contract has the same importance and a proper existence as the classes Person and Company themselves (Figs 5.5.4.2 and 5.5.4.3).

An n-ary association is an association between three or more classes (a single class may appear more than once). The multiplicity for n-ary associations may be specified but is less obvious to interpret than a binary association. In the absence of a special constraint, it is considered as a potential number of instances in the association when other (n-1) values are fixed.

If we follow the logic of business, when x persons are implied in a contract or association with nearly the same responsibilities, an n-ary association can be conceived to represent this collaboration. If we suppress a member of this association, we can threaten its existence, weaken or destroy the n-ary relationship. This fact could be used to test the “solid” foundation for an n-ary association.

N-ary associations may be misleading and sometimes overused. The presence of multiple n-ary associations in a project may be a hint showing that this project is not correctly analyzed or not analyzed at all, as the analyst announces the problem in its most primary form. The following example demonstrates through an example. In the context of very large trade shows that must be held at several locations, the 4-ary association announces at the beginning that a participant may attend several trade shows; each of them can be held at more than one location at various dates. We therefore have the most complex association of type 4-ary with multiplicities N (or *) at each role end.

After the first examination of the 4-ary relationship, most analysts will correct and consider that FromTo.Date cannot be considered as class (or entity in E-R analysis), since each tradeshow has its proper set of *from* and *to* dates (notion of *weak entities* in E-R modeling). So, they are better considered as attributes of TradeShow class than as a standalone class.

Hereafter, we adopt this argument as well, but point out that transformations that results from hunting systematically “weak relationships” may change the semantics and the context in which this relationship has been designed. For instance, FromTo.Date may have a full existence as independent objects (so as

class) if the tradeshow manager extracts from his calendar a set of *from* and *to* date intervals that he considers available to accept all tradeshow proposals. Later, he can decide to match those dates with accepted events. So, a choice of representation implies underlying meaning and this choice was made in a specific context. Hasty conclusions based on some pattern of reasoning may constitute a barrier for understanding subtleties.

Another aspect that may impact on the interpretation comes from the possible Pavlov reflex when converting E-R schemas into object model, regarding the verification of Codd normal forms (NF). When the 4-ary relationship was presented, it was a high-level model and not an implementation one so there is no need to verify all Codd *NFs* when modeling at a high level (even if at the implementation level, it must be transformed for evident reason of efficiency).

As the context is not specified, we may erroneously feel, in presence of n-ary relationships and weak entities, that the problem has not received a thorough investigation. This feeling is confirmed in the case of nonuniqueness of interpretation in the presence of n-ary association with high multiplicity values. The first correction reduces a 4-ary association in a ternary association between Participant, TradeShow, and Location. FromTo_Date class is suppressed and replaced by two separate attributes of TradeShow. Actually, the context has changed: we are not in the model of the tradeshow manager but we are managing all subscriptions (another model and another concern). Within this ternary association, we can track what participant is subscribed to what tradeshow and each location this participant may visit.

In E-R modeling and relational implementation model, if we make a table to display information stored by this ternary relationship (Table 5.5.4.1), some lines show violations of Codd NF.

In relational databases, as table is the only data structure used to store information, NFs previously help us detect some implementation problems. Pure object database (not *object relational*) store objects as they appears at high modeling level and use intensively a network of pointers to implement associations. Information retrieval in an object store is fundamentally *navigational*; there is no table to be joined. So, object databases are not really concerned with Codd NFs. In fact, it violates even the first 1NF.

1NF specifies that all attribute values must be atomic (indivisible). By the virtue of encapsulation, object concepts accepts at the starting point any complex structure (audio, video, image files etc.) so it violates the 1NF as the structure of a component is decomposable and not atomic.

Other NFs are exposed in (Codd, 1970, 1990). The ternary relationship in Figure 5.5.4.4 violates NFs defined by Codd and will be transformed by E-R analysts. The last version of the same Figure 5.5.4.4 eliminates the direct connection of Participant class to Location Class. Participants subscribe to multiple trade

Table 5.5.4.1 Table containing all possible entries of a ternary association between Participant, TradeShow, and Location

<i>Line number</i>	<i>Participant</i>	<i>TradeShow</i>	<i>Location</i>	<i>Comment</i>
1	Adams	Computer Peripherals	North Tower	Computer Peripherals and Location form an independent multivalued association (4NF violation)
2	Adams	Computer Peripherals	South Tower	
3	Bob	Computer Peripherals	North Tower	
4	Bob	Computer Peripherals	South Tower	
	
101	Mary	Software	South Tower	idem
102	Mary	Software	Central Tower	
103	Noemi	Software	South Tower	idem
104	Noemi	Software	Central Tower	
105	Bob	Software	South Tower	idem
106	Bob	Software	Central Tower	
	

shows (1st binary relation) and trade shows are held in multiple locations (2nd binary relation). We can eventually make a search through multiple tables and find out where a participant is loitering at a given time.

Object models are not really concerned with E-R schema transformation techniques, or, in other words, with relational Codd normalization forms. One of the strengths of the object implementation model is the ability to define references that concretize relationships between objects at high level. At low level, references are pointers or memory addresses. When storing an object with references to other objects, OID is used. Once reloaded in memory, the OIDs are reconverted into pointers. Navigating with OID in the storage structure or with pointers in memory is the natural way used by objects to retrieve information on their relationships.

The normalization in object model is by nature iterative, in which objects are restructured until their models reflect user's interpretation (Tari et al., 1997). Object normalization matches interpretation in the UoD (Universe of Discourse) to that of objects in the schema. An interpretation is a set of dependency constraints that provides a given “understanding” of the semantics of the underlying application.

So, when working with object models, we must systematically analyze high-order association (ternary to n-ary) to make sure that we really need them, or there exists a simpler way of modeling things and interpreting them, Secondly, the user's interpretation must be straightforward and fast.

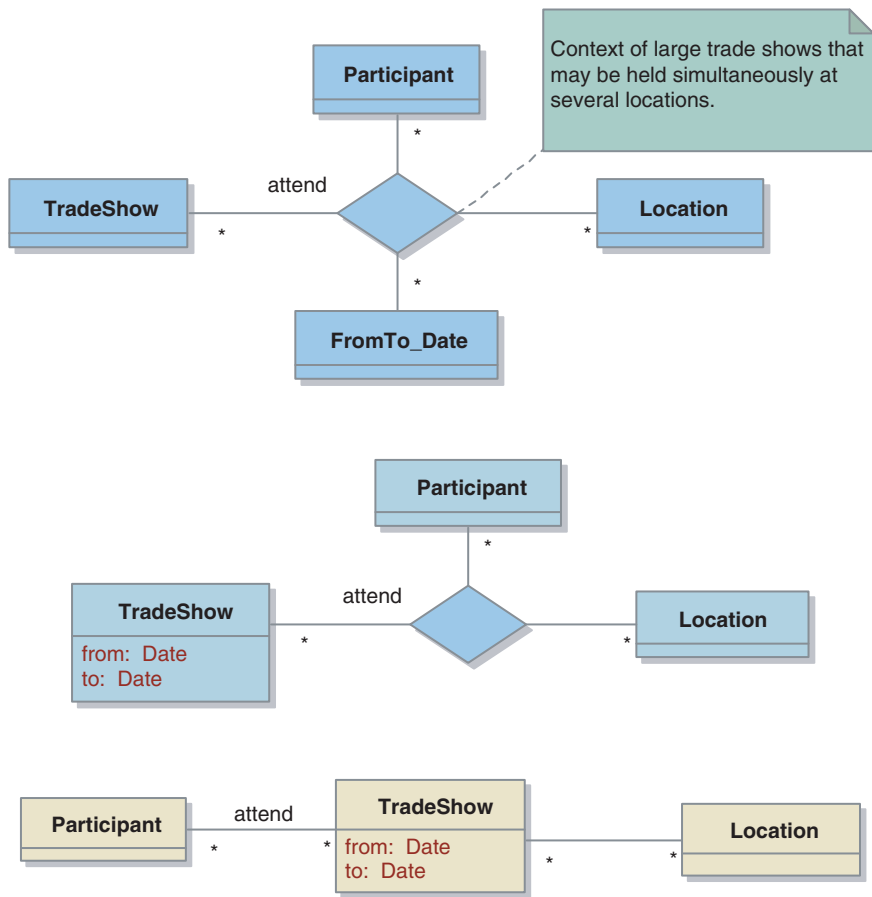



Fig. 5.5.4.4 The first example shows a misuse of 4-ary relationship, corrected in the first step to a 3-ary relationship, then corrected in a second step to transform the ternary association into two binary associations (see text for discussion)

5.5.5 Aggregation/Composition in the Object/Component Hierarchy

Aggregation and inheritance, angular constructions and among the most important features of object paradigm, are often problematic. This fact can be confirmed by any instructors teaching object modeling. Moreover, aggregation has been differentiated from composition in the UML.

An association may represent a composite aggregation (i.e. whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with its. Note that a part can (where allowed) be removed from a composite before

Table 5.5.5.1 Aggregation is considered as the loosely coupled form and composition of the highly coupled form between parts and whole

Aggregation		Composition
		
<i>Cataloguing purpose.</i> Parts are grouped by convenience or for simplifying a description. Description of the whole needs description of parts as well	Treating collections of concepts as higher concepts	<i>Monolithic Component definition.</i> Suppressing details of parts to emphasize details of the whole
No emergent properties and functionalities	With <i>emergent properties</i> and functionalities at various degrees	Individual properties and functionalities are invisible
No lifetime constraint	With loosely coupled lifetime constraint	Lifetime constraint
No propagation of operations		Propagation of operations (whole to parts or parts to whole or mixed scheme)

the composite is deleted, and thus not be deleted as part of the composite. Compositions define transitive asymmetric relationships (their links forms a directed, acyclic graph). Composition is represented by the *isComposite* attribute on the part end of the association being set to true.

We must choose one alternative in the UML (between aggregation or composition) when representing this relationship. As the modeling world is richer and cannot be partitioned into two blocks, voluntarily, we call this relationship all over this book as aggregation/composition (or whole/part) and let the designer specify the correct constraint that accompanies each relation. In doing so, we class this relation into one unique category and let the constraint decide on the nature and the contextual interpretation of this relationship. More details can be found in Smith and Smith (1977), and Henderson-Sellers and Barbier (1999) that redirects to a rich literature.

Aggregation/Composition has some common characteristics: *binary association*, *asymmetrical* (if p is part of a whole w , then w cannot be part of p), *transitive* (if p is part of w , and q is part of p , then q is part of w).

Transitivity may cause some funny semantic problems in the loosely coupled side of this relation. A professor is part of a university. His brain is part of his body so his brain is part of his university.

Figure 5.5.5.1 represents this concept as a *continuous segment* with aggregation as the lower end of the concept and composition in its higher. At the lower end, this relationship can be served for cataloguing and at the higher end as a basic mechanism to build monolithic components.

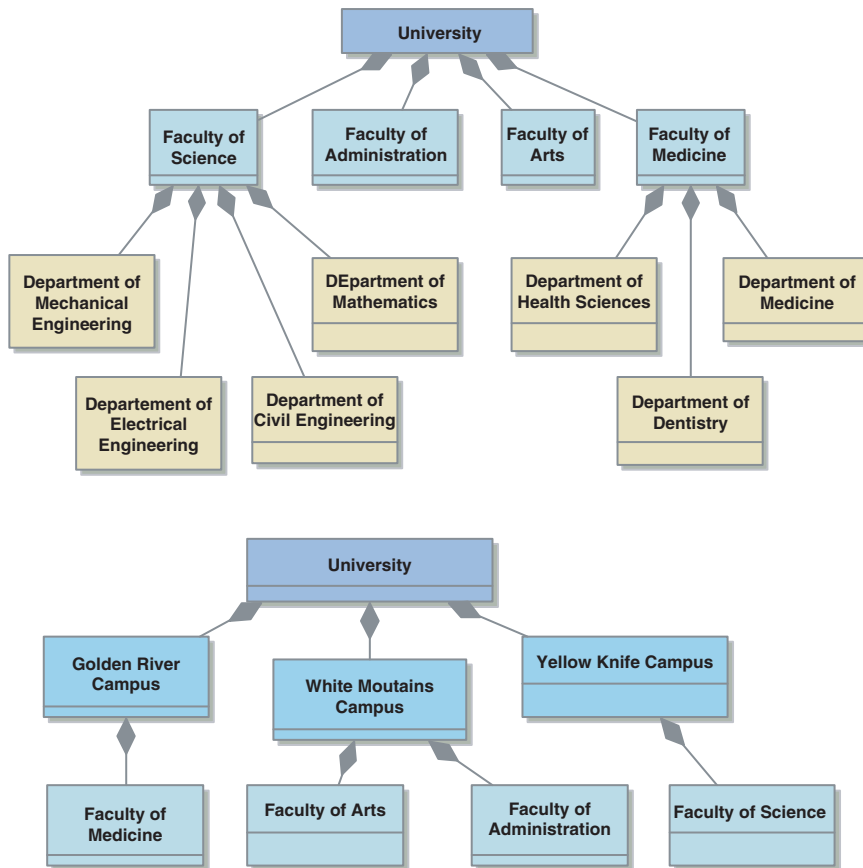


Fig. 5.5.5.1 Same objects but two different aggregation criteria that are “Administrative Structure” and “Geographical Localization”

Components are for composition and components are made of parts. Components have contractually well-specified interfaces and are made of other components, objects. A business component may contain humans or organizations. Aggregation/Composition is therefore a relationship that makes whole ensembles from parts of very different types. The result can be typed itself. Composition is seen as a strongest form that involves consideration on the lifetime of part objects in the UML.

Considering a main window that spawns several underlying windows. If, in an application, we want, with a single click on the “X” of the main windows, delete all spawned windows, the main window is considered as owning all other spawned windows. In this case, the delete operation, activated in the main windows, has started a tree structured delete operation starting from the leaves of this tree structure. We have therefore an operation that is instantiated in the “whole” object and its execution begins from the lowest parts in the tree structure.

At aggregation side, an organization may suppress or reorganize an administrative Unit without any impact on other Units. In this case, aggregation is used as a grouping concept. Parts are grouped by convenience or for description. But, in the same organization, we can have a sub component that suppression may impact on the existence of this organization. In this case, composition is more appropriate. Inside a same application domain, a same relationship has different “degree of composition” or different “degree of aggregation”. Only specific constraints can make things clearer.

Some system is able to work with reduced functionalities even if some components are declared as flawed. Even a flawed component may continue to offer some services. So, it would be very difficult to decide, by a binary decision, whether a system is still usable or not by examining only its lifetime criteria.

A human may lose a limb (an important part of a whole) and is still considered as a human, unless the relationship is not really a composition.

Giving a set of objects, we can have more than one aggregates built on this set of objects, according to the criteria used to establish the aggregates. Figure 5.5.5.1 shows that the tree structure changes completely with the criteria.

From an administrative viewpoint, a university is composed of faculties that are composed in turn of departments. If the criterion is the geographical localization, we have another aggregate since the number of campuses is not always equal to the number of faculties. Many faculties can coexist inside a same campus. So aggregations must be accompanied by their criteria. These criteria must be explicated to avoid non unique interpretation.

When decomposing a system with an aggregation tree or forest of trees, the rule of *level semantic coherence* must be preserved so as, we can stop at any level in the decomposition tree and still find coherence in the description at this level.

Saying, for instance, that a computer is composed of memory, processor P4000, a graphic card, an operating system, a mouse and a printer, etc. is probably the worse description since we find several sub domains at the same level of decomposition. The correct description that respects the level semantic coherence is depicted by the Figure 5.5.5.2.

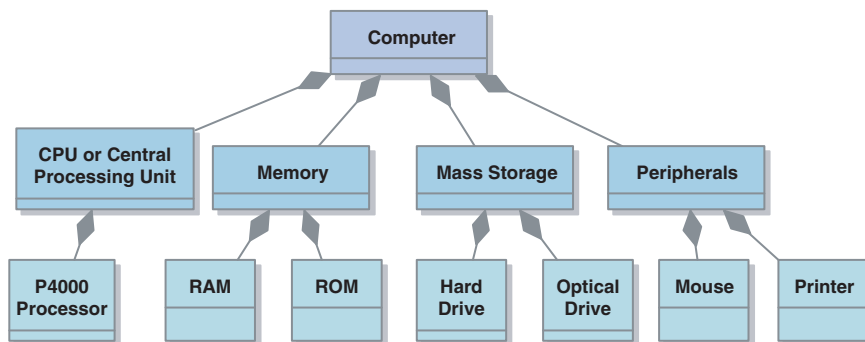


Fig. 5.5.5.2 Decomposition that respects the level semantic coherence

5.5.6 Inheritance

Inheritance is one of the topmost important features of object technology that impacts the reuse mechanism along the whole development process, starting from the inheritance in a UC diagram between actors towards code sharing at the implementation phase. Moreover, it simplifies considerably the way we catch the structural view of all systems. In semantic networks, its name was “is-a” relationship.

Inheritance (Snyder, 1986) works at the class level and factorizes properties and operations in parent classes. *Child (sub, derived, specialized)* classes must define only differential properties and supplemental operations that will be added to characteristics of parent classes. Qualifications as “parent” or “child” refer to the naming of two adjacent classes at a given point in the long chain of inheritance hierarchy. An *ancestor class* is a class that is far away from the current definition, starting from the parent, then the parent of the parent, and so on.

Operation in an inheritance hierarchy can be *surcharged*, i.e. the content of the operation in the child class can be changed but the same operation name is preserved through the whole hierarchy. For the signature or extended signature, local rules may apply.

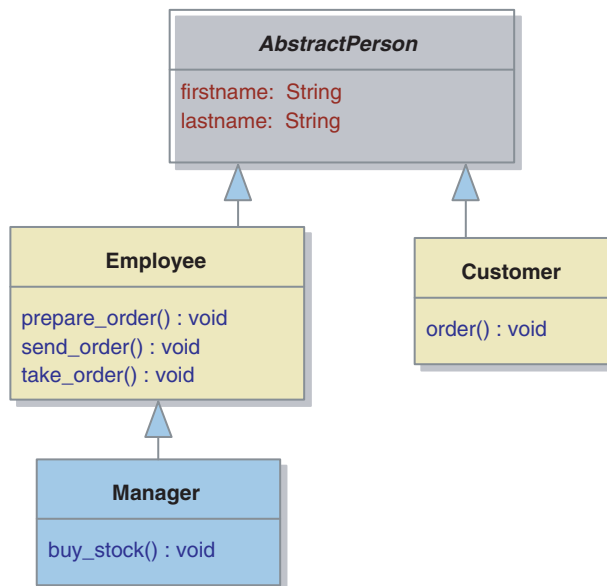


Fig. 5.5.6.1 Example of inheritance hierarchy. Abstract classes are often useful as semantic nodes. Employee, Manager and Customer has all the attributes firstname and lastname. Manager can execute buy_stock() and all operations defined for Employee

Most implementation language does not permit a derived class to “exclude” an inherited operation from its parent. If the operation of the child is redefined to signal an error or to enter an exception treatment if invoked, we realize the equivalent of operation exclusion. This artifice will subtract an operation from the parent.

Excluding operations is an attractive feature. It corresponds to some situations in real world as well. Children cannot always inherit full properties of their parents. But, a hierarchy with some properties added, others subtracted is very difficult to work with as we must keep an eye on every step of the inheritance tree and keep track of all transformations.

When operations are redefined in the child, an interesting implementation feature would allow us to make use of any operation version of any ancestor by explicitly naming this ancestor. This possibility of surcharging an operation is interpreted mostly as a quality as it allows two or more versions of a same program to coexist in a same system (one for supporting older applications and a newer for new applications), compatibility issues are stringent and must be handled in some way. If operations are surcharged many times, it would be very difficult to recognize a class from its faraway ancestor. Structural clarity is decreasing because so much less can be inferred about the properties of descendants. In extreme cases, it would be questionable about keeping a hierarchy of inheritance if classes share only common names for their operations. In general, if a system accepts operation surcharge, there is no guarantee that the objects of the subclass behave like the objects of the superclass.

Inheritance is *subclassing*. Subtyping is theoretically different from subclassing.

Subtyping is a relation of inclusion between types. If

$B \leq A$

B is a subtype of A. We can supply a value of B whenever a value of type A is required. For instance

$\text{Square} \leq \text{Rectangle} \leq \text{Polygon}$

In computer science, subtyping is related to the notion of *substitutability*, meaning that computer programs written to operate with a given *supertype* can also operate with its *subtype*. Any given programming language may define its own notion of subtyping. In most class-based object-oriented languages, subclasses give rise to subtypes. If B is a subclass of A, then an instance of class B may be used in any context where an instance of type A is expected. In this case, subtyping results from subclassing.

5.5.7 Multiple Inheritance

Inheritance cuts down the description of child classes to the strict minimum and allows defining new classes from existing ones. When a system allows a unique parent class, it supports *simple inheritance*. If more than one parent class can be used to define the child class, the system supports *multiple inheritance*.

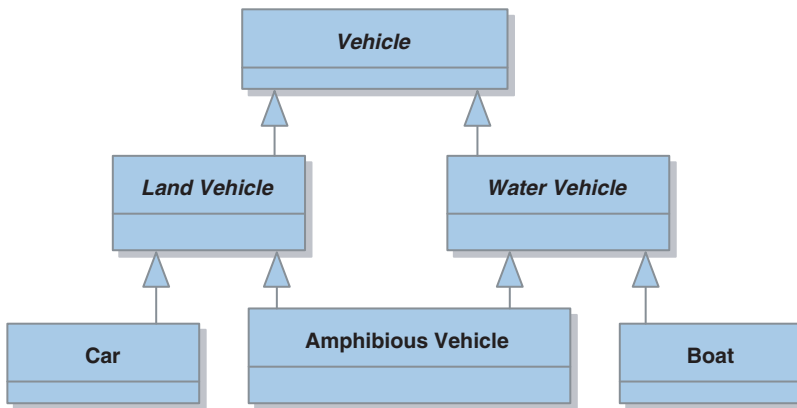


Fig. 5.5.7.1 In object technology, multiple inheritance use union of attributes and operations (not intersection), so the child class is essentially the union of the behavior of the two parent classes

(Cardelli, 1988; Scharli et al., 2003) that has its specific batch of conceptual and implementation problems.

For instance, the class *Amphibious Vehicle* has simultaneously attributes of *Land Vehicle* and *Water Vehicle* (e.g. it has four wheels and helix propeller, it has two maximum speeds, one speed as a car, and another speed as a boat), has both operations of the two parent classes (e.g. it can move both on land and on water). Figure 5.5.7.1 gives its UML representation.

Multiple inheritance is a difficult concept to apply and people can easily confuse it with other relationships, specifically composition. Designers of modern languages as Java and C# have decided that the complexities of multiple inheritance far outweighed its utility. C# supports only multiple inheritance of interfaces but not at class level. The reasons for omitting multiple inheritance from the Java language mostly stem from the fact that Java's creators wanted a language that most developers could grasp without extensive training and multiple inheritance just was not worth the headache.

Multiple inheritance (Fig. 5.5.7.2) additionally poses a classification difficulty when a class derives from two classes coming from different ancestor trees in a system having a forest structure. The easiest way to solve this problem is to accept that a class can belong to multiple categories. Another direction of solution is a relaxation of the famous *is-a* relationship. If D derives from B and C, then D is *like* B and D is *like* C where the form and the degree of similarity (*like* relationship) will depend upon local constraints.

As inheritance is a complex issue, it is often defined in a specified context. In an MDA, it will depend upon the model. So, at high level, inheritance and multiple inheritance can be used to structure and to provide an ontological

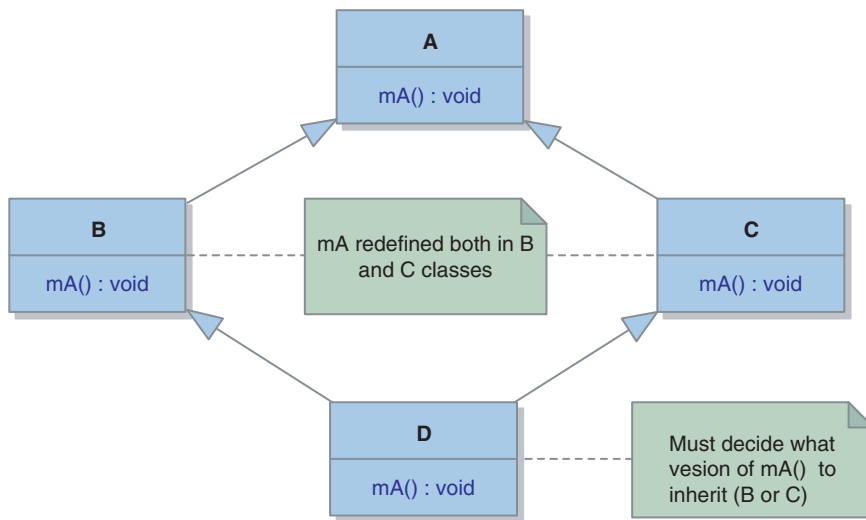


Fig. 5.5.7.2 Classical problem known as the “diamond” problem (Snyder, 1986) in multiple inheritance. Class A has `ma()` [A] operation implemented in A (“[]” indicates in what class, the implementation has been realized). Class B and C inherit from A and redefine both `ma()`. So `ma()` has two new versions under the same name `ma()` [B] and `ma()` [C]. If class D inherits from B and C, what version of `ma()`, from B or from C, the system must choose while deriving a multiple inheritance

description of a domain. In lower level, inheritance supports the reuse of existing classes for the definition of new classes. The mapping between levels or between models needs the engineering of intermediate models to bridge the two worlds. At any level of the MDA, we must observe behavior patterns of reasoning at this level and avoid distorting the reality perceived at this level. So, if a model needs a multiple inheritance, avoiding to model as such arguing that the implementation model does not support multiple inheritance is likely to distort high level models with low level model considerations.

To afford some guidelines for intermediate models, it would be interesting to notice that inheritance can be realized without classes (*classless inheritance*). Classless inheritance makes use of the mechanism of delegation. This mechanism is used to describe the situation wherein one object defers a task to another object, known as the delegate.

Single inheritance is known as a form of “class-class inheritance”. Class instantiation is sometimes considered as “class-instance inheritance” since instances of a class inherits a common interface and an initial state from the class. Delegation is considered in the literature as “instance-instance inheritance” in which an object requests another object to execute the task for it (so it looks like as if there is a transfer of behavior and state attributes to another object).

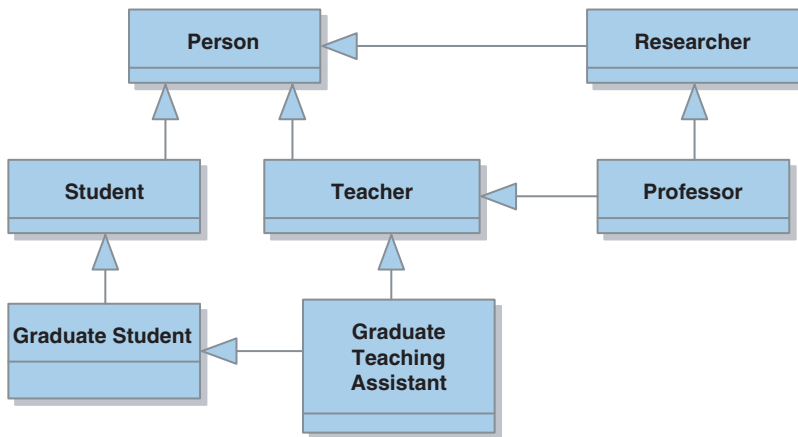


Fig. 5.5.7.3 Inheritance hierarchy containing simple and multiple inheritance

When using inheritance or multiple inheritance, the main thing to remember is the *principle of substitutability*, which essentially says that an object of a derived class should always be able to be used in place of (i.e., substituted for) an object of any ancestor class, whenever an object of that ancestor class was called for. To illustrate this idea, let us consider the example of Figure 5.5.7.3.

In this example, the *Graduate Teaching Assistant*, by the principle of substitutability, may replace anytime, *Graduate Student*, *Student* or *Teacher*. A *Professor* may replace a *Researcher* or a *Teacher*.

The substitutability rule is necessary but not sufficient to detect early incorrect structural constructs, as shown in Figure 5.5.7.4. A plasma display may be substituted to any of the three classes *Luxury Item*, *Electrically Powered Item*, and *Breakable Item*, but deriving immediately any final item from those three classes may create an awkward and bulky structure.

Another example of Figure 5.5.7.5 shows the difficulty of deciding between a composition and a multiple inheritance structure. The choice of a solution is often contextual.

A company wants to have a person for a multidisciplinary project (for instance robotics) that needs the skill in the following fields: Electrical Engineering, Mechanical Engineering and Computer Science. By deriving a *superman Multidisciplinary Engineer* class and solving some minor conflicting parameters, we choose a multiple inheritance approach. By hiring three different engineers and building a team, we choose the composition approach. Doubtlessly, the first solution is more economical for this company as it has only one salary instead of three, no team work and potential communication or human problems to solve, but it would be difficult to find such a perfect candidate.

If we transpose the same problem in the context of an expert system, if the computing resource is not a limitation, the choice between the two solutions will

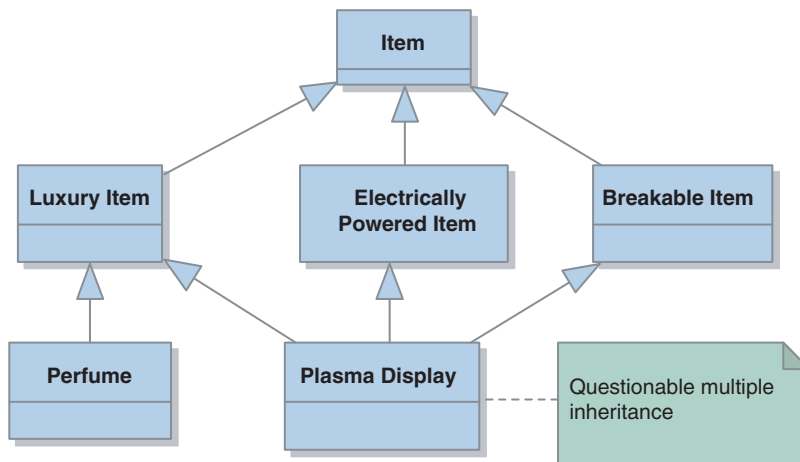


Fig. 5.5.7.4 Example of questionable multiple inheritance hierarchy. It would be advisable to create an intermediate class *Luxury and Electrically Powered and Breakable Item* class either by multiple inheritance or directly from *Item* if not all attributes and operations of *Luxury*, *Electrically Powered* or *Breakable* classes are needed

be less evident. We can examine also a hybrid solution that considers a conceptual model (PIM) different from the implementation model (PSM).

5.5.8 Objects and Roles

A role is a part played by an object in a relationship. A given object may participate in a system fulfilling any arbitrary number of roles. In the absence of specific constraints, there is no upper bound for the number of roles an object can play. Roles can be *dynamic*, so objects may change their roles in run-time (during their lifetime) and this fact could be modeled appropriately if we dissociate the role and the object that fulfill this role. If all objects keep the same roles during their lifetime, the system assigns *static roles*. The role concept has been used early in data modeling (Bachman and Daya 1977). The notion of role proposed in Object Role Modeling (ORM) (Halpin, available at: www.orm.net) is a different concept and methodology. (Steimann, 2000) has recently reviewed the notion of role. The notion of role used in this text simply means a part played by an object in a relationship.

Student is a role played by a human at a given period of his existence. *Electrical Engineer* is both a title (value) and a role. A person with this title may never exercise the corresponding role and may be doing something completely outside its role, for instance teaching music. A teacher who takes complementary courses plays both the role of a teacher and a student at the same time.

A company may be customer and supplier at the same time.

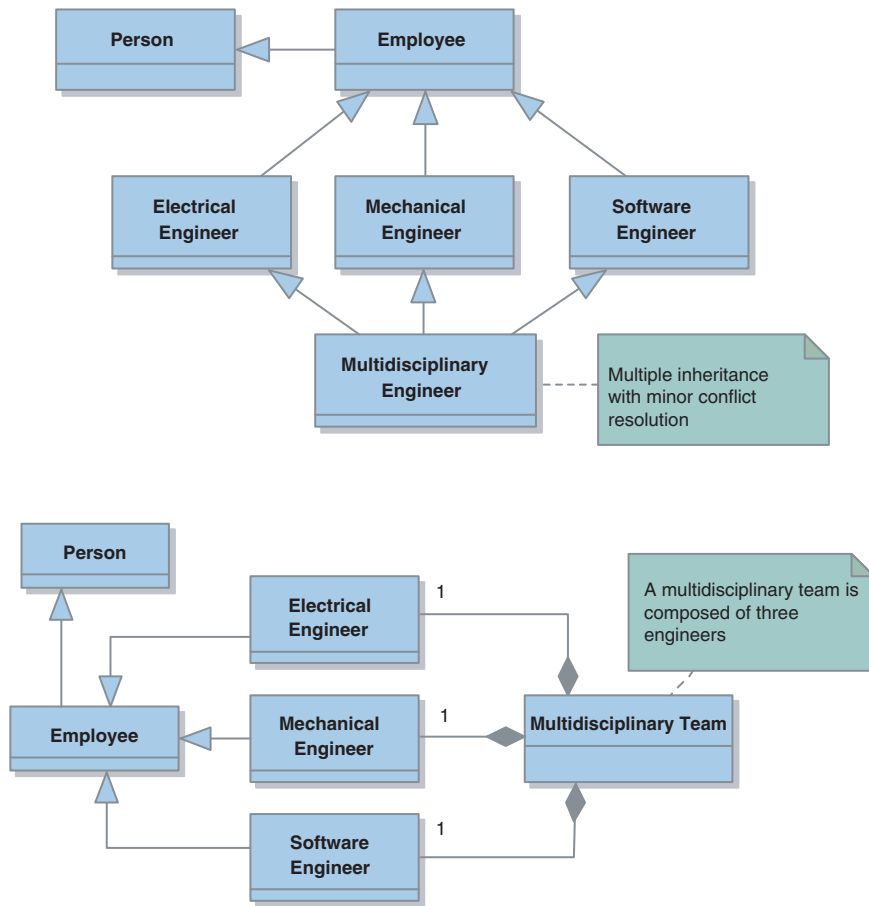


Fig. 5.5.7.5 Composition or multiple inheritance? The first solution is a multiple inheritance with some minor conflict resolution and the second is a team of three engineers

A job in a company is also a role. Once instantiated, the state of this job can be suspended until there is an employee that accepts the job. If this person is laid out, the job will be suspended again until another candidate accepts the job.

The role concept describes a component of an organization, specifies skills, functions, tasks, and all features (a kind of contract) needed to perform any activity defined within the role. Even if, behind a role, there is an object, a role has its own set of state variables. Normally, an object that executes a task defined in a role keeps an identical copy of state values of the role it fulfills (more generally, it can access or keep an *image* that may reflect the reality or not). If it does not have enough resource to keep a copy, at least, when reentering the role, it can reload this copy in order to be able to continue to play its role.

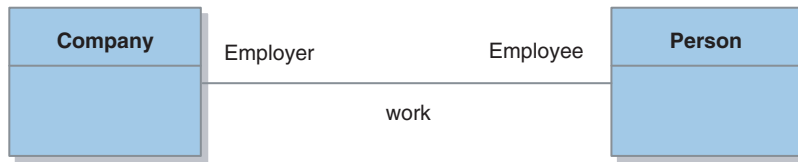


Fig. 5.5.8.1 Representation of role in the UML as name of an association end

Static roles are applicable to a small category of problems. Dynamic role attribution is the general model. Roles can be instantiated, have states, behaviors, and identities. An object and its roles are then related by a special relation *play* or *played by*. At the outside, the object and its roles are aggregated (Kristensen and Osterbye, 1996) inside a unique and indivisible *subject*. If an object drops a role, this does not necessarily mean in all cases that the role must disappear.

As discussed previously for a job, a role can be suspended. It is not necessary to suppress a role even if there is no object that takes over temporarily this role.

As for the identity of the role, some authors admit that only objects may have identities, not roles. Once more, this restriction works for some classes of problems. A role can have a temporary identity. A student in a university may have a registration number for this university. The proper identity of the person is different from that of a role. We can eventually make a search in a database with the identity of the person or the identity of his role.

If we dissociate roles from individuals that fulfill roles, the model would be more complex but very precise. Any real or abstract entity may, during its

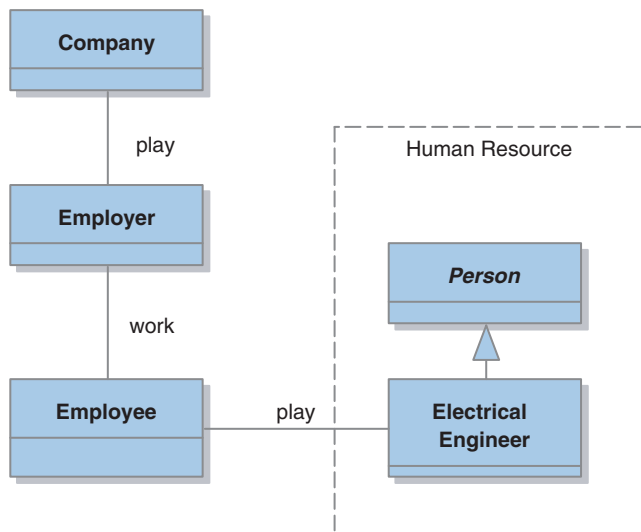


Fig. 5.5.8.2 Roles are modeled as separate classes

lifetime, acquire or lose any arbitrary number of roles without changing its proper identity. When we model an organization with all its business processes, it will appear as an idealized model in which we project to reach some objectives. Now, humans with their skills appear as resources that activate those roles and bring the system from state to state. Roles will be treated as objects and they have their own classes. Every system will now appear as been duplicated into two hierarchies. When we say for instance “somebody has not fulfilled its role,” we now have a way to encode how we expect the system to behave and what is the result observed with objects we assign to some roles.

The UML proposes a way of representing roles as names of the association ends, so there is no duplicate hierarchy. It appears as the role delimits a subset of objects that participate into the collaboration and there is no “aggregate.” This notation is heritage from the E-R diagram and the OMT (Object Modeling Technique) methodology.