Chapter 2

Coping with Complexity

2.1 Visual Formalisms for Handling Complexity: A Picture is Worth a Thousand Words

2.1.1 Free Diagrams

"Seeing is believing" or "A picture is worth a thousand words" stress the importance of visual formalisms in learning and communication. Every time, we want to draw a system, unconsciously, we draw a circle or a rectangle. A closed contour delimits a small surface that materializes an abstraction, in our case, a system, an object or a component. Children draw forms before language. Primates such as chimpanzees produce paintings with a striking similarity to drawings made by children at the early stages. The world seems to have an iconic representation before we can abstract it with language. Six recurring "diagrams" were identified by psychologists with children: circles/ovals, squares, rectangles, triangles, crosses, and irregular odd shapes.

Irregular odd shapes are difficult to draw (Fig. 2.1.1.1). In the early stage of object technology, Booch (1993) used a cloud or blob notation to emphasize abstractions. Venn diagrams use a very simple form of blob. Odd shape is dangerous because people draw it differently. Since it conveys artistic flavor, it may be appealing to someone and may create an opposite reaction to another and this fact can impair the message we want to pass. Moreover, the natural tendency when drawing odd shape is to include technological details in shapes and then suggest precociously an implementation detail in a logical model. For instance, when drawing a security lamp, if we represent it as a rectangle, we let to the implementer enough margin to decide and operate a technically optimized choice, but if we represent it with a pear form, it suggests an early incandescent lamp, so we have unconsciously passed a message to the implementer. Unless, this choice is an initial requirement of the contract, this design "bug" should be avoided while modeling systems. So, to avoid cloudy and hazy messages, irregular odd shapes must be banned from technical diagrams.

D. M. Bui, Real Time Object Uniform Design Methodology with UML, 19–45.

Fig. 2.1.1.1 Odd shapes and free forms. Despite its artistic flavor, the cloud representation of object is difficult to draw. The pear form of the security lamp passes an unclear and unattended message to the implementer. A simpler and neutral shape such as a rectangle is more appropriate to represent object at the logical phase

However, free diagrams or nontechnical diagrams that contain odd shapes are currently used by many people, incidentally those who give contracts. In fact, we cannot command everybody to know how to interpret technical diagrams. These persons come from disciplines distinct from Sciences or Computer Sciences (that is why they need us), they try to communicate with us and express their vision of things differently with nontechnical diagrams, sometimes with an artistic flavor. We must clarify things and identify true requirements with them in the analysis phase. Sometimes, it would be advisable to insert nontechnical and artistic diagrams accidentally in a conference or a meeting to capture the attention of the audience. So, do not underestimate the accidental power of unconsciousness.

2.1.2 Technical Diagrams

Technical diagrams are iconic by nature. Icons accelerate the recognition of the abstract concept. They must be self-explaining, suggestive, and should not, as said before, contain any detail that could be interpreted as implementation constraint. Each profession has its own set of icons. Electrical engineering for instance has a large set of icons to represent real objects at various scales (op-amp, resistor, capacitor, transistor, etc.). Computer analysis and logical models make use of standard and neutral forms like circles/ovals, squares, rectangles, and triangles which are connected by lines and arrows (single or double oriented lines). Except for circle/oval, these forms are made from a succession of straight lines (edges) and vertices forming a close contour, colored or not, delimiting an abstraction. They are *objects with volume*, meaning that they fill up a given space on a drawing area. Lines and arrows that are lightweight *objects without volume* may then connect objects with volume to establish relationships or communication channels between them.

The number of objects with volume laid on an $8.5'' \times 11''$ document must be limited to avoid *visual surcharge*. Psychologists think that visual

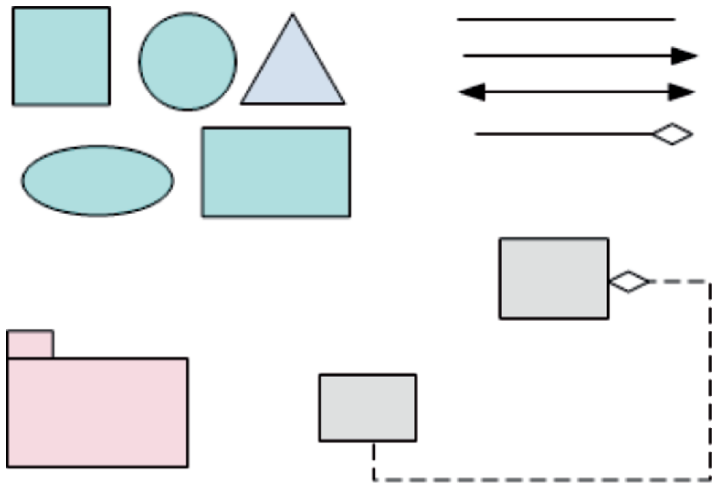

Fig. 2.1.2.1 Objects with/without volume, decorations, and adornments. Primary shapes with volume occupy space in an area. Arrows, connectors are object without volume. If connector makes a roundabout way to reach two objects, it may be equivalent to an object with volume and may create an unattended visual surcharge. A rectangle with a smaller rectangle as adornment is considered as a unique object as long as the adornment volume stays reasonable

short-term memory is able to store 7 ± 2 objects only (Miller, 1956) as a population average. Even if the theory of short-term memory and their empirical result are controversial, it would be common sense that there must be some limit for the number of objects with volume we can put on a diagram, by the simple fact that identification lettering of objects must remain visible. Objects without volume, such as lines and arrows, do not contribute to the visual surcharge at first approximation but, if a line follows a complex way to join two graphical objects with volume, it mimics a visual object with volume and by the way, may be assimilated to a pseudoobject with volume and could increase the visual overload as an object with volume does. Figure 2.1.2.1 summarizes these notions.

Must we fill objects with volume with color? The theory and treatment of color is a huge topic that has spawned massive discussions with conflicting opinions. Color sends a very strong signal and bear cultural values. If color can be used effectively to differentiate efficiently model elements, there is no single accepted theory about what color combinations are most effective. Color can make a dramatic difference in the aesthetic qualities of a diagram and as consequence can influence our desire to go further with its content. Ill-colored diagram can disturb, create confusion, and discomfort the viewer. So, be careful. A diagram without color (or white colored) can be more interesting than a diagram colored with a doubtful combination. Heavily saturated or primary colors should be used sparingly because they draw attention away from other

elements without mentioning the viewer fatigue and blurring phenomenon of accompanying text. Pastel colors are usually a good choice. As a thumb rule, never use more than six colors for a diagram (white paper background and black ink count already for two).

As humans tend to seek consistencies among visual elements, each of us can adopt a personal standard (if it is not already imposed by our organization). So, for instance, the human group will be colored in the same way at any level of the system. A package that would be decomposed further must create a kind of anchor in the way it is colored. At any stage of the visual process, remember that unity, harmony, good balance, and consistency are master words for guiding the modeler through his presentation and the way he communicates his vision of the system to other persons.

2.2 Object/Function Decomposition: Layering, Hierarchy, Collaboration

In the past, when modeling was in process, designs are conducted in a *top-down* manner, so the methodology of complexity reduction was called functional *decomposition*. Combining elementary objects to form a larger object is the *bottom-up* method and the process is called *composition*. Although composition/decomposition has been used for a long time in conventional modularization techniques, they are still valid mechanism of complexity reduction in the context of object-orientation.

When decomposing, objects identified at each level of decomposition compose a layer at that level. This process thus creates a hierarchy of objects with the most complex object lying on the top of the hierarchy. Elementary objects are found at the bottom layer.

> The car system can be broken down into major interacting subsystems, such as the engine, the transmission, the cockpit . . . that are made up of more primitive components. For instance, the engine consists of carburetion and cooling subsystems. At the most detailed level (bottom layer), subsystems are primitive physical component like screws, metal sheets, molded parts . . . The car's cooling subsystem can be described in terms of radiator, water reservoir, rubber tubing, and channels through the engine block. The car stays alone at the topmost layer. At intermediate layers, we found engine, transmission, brake, direction, passenger cockpit. At lowest level, a car repair shop can list a computerized inventory of thousands of parts.

Hierarchy and layering technique correspond to aggregation/composition counterpart in object technology. But this is not the only structuring mechanism governing the world of complexity. As applications become more and more sophisticated, they associate many objects coming from various disciplines. Tasks in everyday life are executed with the contribution of many objects which cannot be put inside a hierarchy. In this case, the term *collaboration* is used to describe

such association, and objects are collaborative objects/agents assuming some roles while achieving a common goal. The same object may have multiple roles if it can execute more than one task. If aggregation/composition hierarchy is seen as a *vertical* mechanism structuring the world, collaboration is the *horizontal* mechanism. In fact, modern systems call for these two mechanisms simultaneously at various levels.

> If we consider now the previous car with a conductor inside interacting with the car and looking at the highway, we have both a hierarchical system (the car) and collaborative objects (conductor, car, local geometry of the highway, other vehicles, local meteorological objects like fog or snow, mobile phone, etc.). All these objects cannot be aggregated inside the car hierarchy, so, only collaboration could be the most appropriate description at the driving level. In a normal situation, all the car objects are hidden from the car hierarchy to simplify the discussion on driver reactions, but in case of an accident caused by the driver grasping his mobile phone, the quality of a hidden component, its airbag, will be of highest importance.

Collaboration is also the main working mechanism inside human organizations. Roughly speaking, the hierarchical vision is more appropriate for dumb objects. If we take individually molded parts of a car engine, we can do nothing with them, they must be mounted together to accomplish a given task. But, when objects become more and more intelligent, they can execute several functions besides those specific to collaborative tasks. They are autonomous *agents*; they can share their knowledge and capacity in diversified collaborative tasks involving many agents.

> When designing the structure of a company, people mostly see it as a hierarchical organization with a president, departments, managers, engineers, workers. A containment relationship is often proposed to put workers under identified teams and humans inside departments. But, the way we model the system is not static and depends upon the current context, time, space and culture. There is no universal valid model. Must we model the interaction model of this enterprise with a horizontal or a vertical scheme? With a tribal organization in undeveloped countries, the company is a real hierarchy as the head acts as a tribe chief or a king, but in democratic countries, a president of a company can be replaced by its administration board without affecting the functionalities of the company. So the model depends on space, time and culture. In North America, we can still have a tribal organization but, generally, workers/engineers/managers can be replaced so a collaborative model is more appropriate to account for its operation. Sometimes, we must choose a mixed model to account for the reality.

2.3 Functional Decomposition to Handle Complexity

2.3.1 Functional Modeling is Based on Processes

A *process* can be seen as a set of ordered and procedural steps (atomic actions or activities) intended to reach a goal (Fig. 2.3.1.1). A process is present in the UML standard which defines it, within the current grounds, as a heavyweight

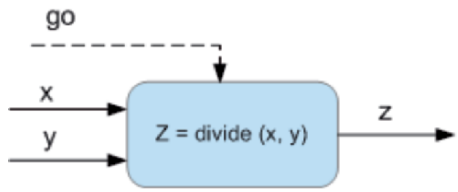

Fig. 2.3.1.1 Gane-Sarson or Yourdon-DeMarco bubble representing a process in functional modeling. Control is based on process/functional modeling. Arrows arriving on the process materialize input data and arrows leaving the process are output data. Dotted arrows are interpreted in DeMarco methodology as control signals. In the figure, the "go" signal fires the division and yields z data output when x and y data are made available at the inputs

unit of concurrency in an operating system and reserves the term *thread* to characterize a lightweight process or simply a single path of execution through a program. Function, process, procedure, thread, task, action, and activity will be examined in Chapter 5.

Classic functional modeling, not object oriented, have their adepts and a lot of blueprints still populate industrial drawers. Even, a good object modeler must understand what a "bubble" is because it is not excluded that some days; he will have to read, maintain or reengineer an old project made with a functional dataflow methodology (Yourdon, Ward-Mellor, Yourdon De-Marco, Gane-Sarson, Jackson, etc.). The DFD (Data Flow Diagram) can be found currently in modern CASE tools or general drawing tools as Visio. DFD comes from one (among the oldest methodologies) proposed by Yourdon which represented a C function as a circle (bubble) with inputs (parameters passed within the function) and outputs (parameters returned or passed by reference then modified when the function executes). A form other than a circle can be used to represent function, for instance, a rounded-edge Gane–Sarson rectangle.

2.3.2 How Process is Converted to UML

As mentioned earlier, control and process modeling have not lost their importance in UML since processes in object technology are viewed as operations defined in classes (structural part of a project) or actions/activities (behavioral part of a project). Control flows in the UML activity diagram can convey signals and be used to coordinate the execution sequence of tasks. Therefore, the "go" signal in Figure 2.3.1.1 is converted into a control flow in the UML. A control flow cannot pass an object or data but only "token" to activate the next process. A control flow is notated as an arrowed line connecting two actions/activities. Data flows are replaced by object flows. Datastores in the Gane–Sarson diagram will

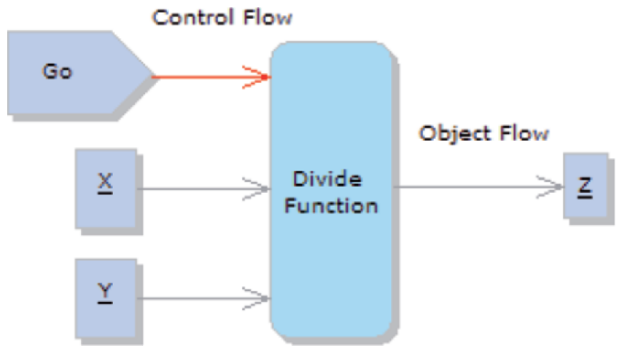

Fig. 2.3.2.1 Same Process as Figure 2.3.1.1 with an UML activity diagram. X, Y, and Z are objects. The arrows representing a data flow is now replaced by object flows (arrows connecting X, Y, and Z to the Divide function). Control is issued from the "Send" element which generates a "Go" signal. The bubble is replaced by an action or activity in the UML

be depicted as stereotyped objects. So, at low level, the UML has elementary equivalents to achieve conversion of all processes.

2.3.3 The Limit of Functional Decomposition: Half a Century of Modular Programming

With the resurgence of business processes for modeling business applications, we find it interesting to discuss what we have learnt from the past in order to avoid spending time in the wrong direction and find the best way to integrate various applications. One of the well-known mechanisms to handle complexity proposed in the early days of programming (Parnas, 1972) was *modularization* or the process of chopping a big program into smaller chunks, which is more easily understandable and manageable. *Structured design* (Yourdon and Constantine, 1979) addressed this problem from the functional design standpoint. The primary goal of the decomposition into modules is the reduction of overall software cost. The rationale of modules was multiple. Modules can be designed, developed with easily understood structure, and can be revised separately without affecting the behavior of other modules.

Good module interfaces, complete module descriptions, data flow analysis, task decomposition, information hiding (local variables), etc. were the buzzwords of modularizing process of functional generation. Fundamentally, *structured design* and *modular programming* consider every program as a huge process to be discovered and decomposed assuming that, at any level inside the hierarchy of decomposition, the subproblem stands alone as the original problem. People suppose that if all of the subproblems in the decomposition are solvable, then, solutions of these subproblems can be combined together to

build the solution of the original problem. This technique is applied recursively towards the lowest programming functions extracted from standard libraries or instructions that are parts of every programming language.

We know nowadays that those objectives are reached partially with the functional methodology. The modularization process operated by a C programmer generally differs from the way we think of at high level with an object/agent model.

> Students in the seventies remember that the main() function in C was always at the top of the call stack hierarchy. The first intuitive attempt was the separation of prototypes ".h" files from their corresponding ".c" and the repartition of the grape of functions under the main() into several files to be able to compile them separately, both to save compilation time and help organizing a tremendous number of functions spawned within medium scale program.
>
> The criteria adopted by C programmers to decide on what function must go into what files varied substantially from individuals. If main() calls A1(), B1(), C1(), those subfunctions and their dependencies can go into 3 separate files (vertical grouping) but programmers can decide that A1(), B1(), C1() form the first calling layer (horizontal layering). Besides those "geometrical" methods, some programmers group functions according to their usage, their domains, for instance I/O, low-level access, mathematical, etc. *Criteria are variable from one person to another.* Earlier, some rare birds, born with object paradigm in the head, grouped functions naturally as they must be in the object paradigm.

Functional decomposition does not fit well with modern software engineering since this idea seems to work only at low level for small local control program.

> When working with a function, the first called function, let us say A(), is at the top level of the call stack, so all called processes at any level below A() are hierarchically under A(). It looks like A() is the master and it "holds" all other processes just by the fact that it can wake them up. If other hierarchies B() or C() share a common function L(), the latter will be separated and stored in a library to allow B() and C() to reach its compiled code of L(). A() does not hold L() anymore. An enhancement notion of code reentrance allows B() and C() to call L() and creates a separate copies of independent state variables. In this sense, reentrancy was the faraway ancestor of object instance and code reuse.

Nowadays, we know with MDA that a system could be developed by exploring successively various models. Models are tied to domains and ontologies. Even if there is a one-to-one mapping of the classical development phases into models, the reality of models differs substantially from the classical way of viewing development steps. Therefore, it would be very difficult (not unfeasible) for any developer to operate a functional decomposition without any risk of mixing domains and ontologies through an inextricable hierarchy. On a single blueprint, it is frequent to see logical concepts mixed up with platform-dependent nodes.

Moreover, modern software is not based only on hierarchical structuring process but relies more heavily on a middle structural model mixing

both hierarchical processes to collaborative tasks in which intelligent and autonomous agents participate with their specific roles in a universe crowded with objects. If we simply consider a server in client–server architecture, the server must serve several clients in various contexts and when a client makes a call to a server for servicing, the calling side or the client does not "own" the server just for its own use by the fact that it can call the server for servicing. In a company, even if a president instantiates a production line, he does not own all his employees or other servicing companies, he only instantiates a collaborative process to reach a specific goal. We do not need to nest functions into functions as programming languages Modula and Pascal did and in this sense, modern software is "democratic," mimics human organization and is built on a *collaborative model* as each computing block identified as agent/object/component is more and more intelligent and versatile.

Let us now compare programming languages as Modula, Pascal, and C relative to the way they supported modularization. Pascal and Modula were built on the philosophy of modular programming as they support nested functions directly in the language. Nested functions are not allowed in C so the structure of C does not fit obviously to this way of thinking and the call stack of C is treelike with each node in the tree being a procedure or function. But a program is more than a bunch of processes. Niklaus Wirth, the Pascal inventor, my neighbor in the same university, stated that program consists of process and data with his well-known formula

$$\text{Algorithms} + \text{Data Structures} = \text{Program}$$

In fact, his equation fits only to a class of problems in which the system can be structured, not awkwardly, into processes. Moreover, algorithms and data can be found and fitted into this structure. The call stack constitutes itself a control structure but data that circulate inside this control structure belong to another different data flow network as data repositories must be added and data paths are superposed to the network of control flows. If we do not have appropriate data flows for a function, we must rip out a large branch of the tree and replace it with one that has better data coupling.

> A similar example is found in architecture. When we build a full residential district, every house must have water, electricity, phone lines, internet connections, etc. Several networks are therefore superposed to give a full residential service package. If the ground relief is complicated, it can impact on the way houses must be built to accommodate all servicing networks.

This way of programming considers that the nature of every solution to a problem is initially a process or a set of actions. Later, people try to fit data inside this structure as an overlaid structure. This is essentially a development process including two nearly independent steps which can give rise to structural mismatch between the tree control structure and the data flow network.

To share data between processes, *global data* were a programming buoy that provides a way for processes at different levels to communicate together. But when programmers are asked on what theoretical basis, they created such global variables, answers are rather hazy. Moreover, global data could easily get corrupted or incoherent, as they are accessible to all functions, even to those which do not have any right to access them.

However, functional modeling is still of topical interest as few of us are computer scientists. Financial analysts in business branch always think of plans, course of action, means, directives, goals, results, etc. Their domain vocabulary is mainly function-oriented, with actions, inputs, and outputs. In fact, computer scientists and business analysts are in two different spheres (domains). To conciliate things, it would be interesting to start talking in the language of business where the application is. Curiously, we can reason with functions and processes at a very high and abstracted level owing to the over simplification of things and due to the absence of complex control structure and data circulation network, still not identified at this stage of project development.

In the uniform methodology explained later, we still make use of processes at two levels, the highest and the lowest. In the requirement engineering phase, business process modeling is used to communicate with clients to identify tasks, plans, required inputs, and desired outputs and to understand their vision of the system. At the lowest level, in the implementation phase, when dealing with object operations packed inside classes, we need inputs and outputs. In the dynamic view, the activity diagram of UML version 2 reestablishes and legitimates the data flow concept. So, functional methodology can be used as an auxiliary tool for complexity reduction at the highest and at the lowest ends of the development process but not as the main concept for structuring the whole system.

2.4 How Objects Technology Handles Complexity

2.4.1 Object Paradigm

From a programming viewpoint, in a typical nonobject-oriented development, we have two separate things: code that is a bit of logic to perform task and data manipulated by code. Data and code are bundled together inside a new entity called *object* in object technology. Data can be everything necessary to model classical systems (data of a database system, state variables of a real-time system or function constants for a mathematical formula). Data receive the specific name of "attributes" at the early days of object programming in the implementation model and codes are organized around "methods," which are merely object operations.

Objects are both physical and conceptual things found in the universe. Hardware assemblies, software components, electrical devices, mechanical

machines, documents, human beings, and even concepts are examples of objects. To model a company, a designer views buildings, divisions, departments, employees, documents, and manufactured products as objects. An automotive engineer will see motor, carburetor, tires, doors, etc. as objects. Atoms, molecules are object candidates for a chemist in an OOS of a chemical reaction. Transistor, logic circuits, and op-amps are objects for electronics engineers. Memories, stacks, queues, windows, and edit boxes are objects for software programmers. The whole earth is an object when seen from the space.

Objects have states while working. States of a car are its "running, parking, accelerating, decelerating" states. A state of a bank account object is its current balance, the state of a clock object is its current time, and the state of an electric contact is "on" or "off." Humans or agents have more complex states, the number of state variables is tremendously large but, fortunately, while modeling, we typically restrict the possible states to only those that are relevant to our models.

Simplest objects are *passive objects*. An unanimated mechanical piece, for instance, the plastic cap of your keyboard, a character sent to the screen, or an electronic resistance or capacitor are passive objects. Their state may change when exposed to actions performed by active objects. Active objects can instantiate changes to the universe and to themselves. A clock is a simple *active object* in this sense as it can change its state every second. If we put a computer inside a robot and give it some energy, the robot can work on a production line. If sensors and some dose of AI are added, this dumb robot can become intelligent and its behavior mimics that of a human. Humans are biological, intelligent, and active objects as they can instantiate changes, though not always intelligent, to the universe. They are called agents in social or business contexts. The term "component" is used currently to account for passive and/or active and/or intelligent aggregate of objects that can be deployed repeatedly in many applications and contexts. To alleviate our description, according to the context, we consider the term "object" in its largest sense (as common vocabulary) including agents and components. An agent can be an entire and complex hierarchical organization and a component may also be a very complex system, but part of a larger system.

In object/agent/component technology, every task is performed by an object alone or a collaboration of objects. Each object contains some data of the system and objects are created with procedures and operations for acting on internal data. Once again, data are understood with their largest sense. They can be some data records of the system, data describing object states, or data manipulated by objects. Data can be considered as objects if necessary, but not all the time. For simplicity, attributes can contain values and operations can have values as inputs and outputs.

In the programming world, the access to data is made under the control of the object that holds these data (It is not the universal model of real world. In

real situations, objects can modify directly properties of other objects without their consent or approval). In other words, data can be accessed only by calling to an available service or operation of this object. So doing, the system is partitioned into entities that isolate changes. To accomplish their tasks, objects communicate with each other through messages. Messages can request a data or a change of data through encapsulated operations. Depending on how the change procedure is implemented, an object can control the identity of the object that requests the access to its data, either in read or write mode. Some degree of security is obtained by encapsulating the process and imposing systematic control to data access.

2.4.2 Information Hiding, Encapsulation

How object technology copes with complexity?

First, an object is a black box and it hides the underlying implementation. In the object world, only the creator or the company who builds the object knows the implementation details. Providing access to an object only through its messages, while keeping the details private is called *information hiding*. An equivalent buzzword is *encapsulation*. The consumer who uses this object has three possibilities:

1. Use directly the object through its public interface. "Public" means "visible to everybody."
2. Derive and create a specialized version of the original object if he wants to add or change slightly the object behavior (inheritance concept).
3. In the presence of parameterized class (a class is a mold for creating identical objects), supply parameters to create an instance.

So, complexity is partly cancelled by the information hiding mechanism. We can drive a very complex and high-tech car knowing nothing about multivalve ignition, antilock brakes, and vehicle stabilization system as the high-tech car offers nearly the same interface (not the same comfort) for driving as all other regular cars. A complex object is often monolithic, meaning that its structure is indiscernible from the outside (black box concept).

2.4.3 Composite Objects are Built from Aggregates

The perception of complexity comes from two aspects, the complexity at the interface (the object is difficult to use) and the complexity of a composite (objects are made from a concatenation of a large number of smaller components). If the structure is indiscernible from the outside (monolithic), we can reduce the second factor of complexity feeling but not the first one.

Composite objects are built by aggregating simpler objects and the numbers of objects and levels are not limited. If the object is not a finished product (e.g. a vision system), must be integrated into other systems, the term *component* is used to describe it. If the object is a human or implies a complex mixture of humans, business, and social objects, then terms such as "agent," "organization," "process," or equivalents prevail.

Aggregation/composition is the most important mechanism (at run-time, in real world) to deal with complexity in object technology.

> A human body is composed of a respiratory system, a circulatory system, a nervous system, a skeletal system . . . An aircraft contains a propulsion system, a navigation system . . . A program is composed of a user interface, a main core, local and distributed components. This relation is been called the *part-of*, *part-whole* relationship in the era of semantic networks.

Objects may also exhibit a complex structure involving both collaborative objects and aggregates. A computer is composed of a microprocessor, memory, interfaces, communication structure that is in turn built from two chip sets, etc. A computer is an aggregate of these objects but at the next level of decomposition, memory, microprocessor, and interfaces are collaborative objects. As long as we do not need to know "composition details" to make use of aggregated objects or monolithic objects, the process of building monolithic objects from aggregates is the main technique of complexity reduction. The interface with the user must be simplified to enforce this complexity reduction process.

2.4.4 Class Factorization as Complexity Reduction Mechanism

Another complexity reduction mechanism was the identification of identical objects and the class factorization mechanism. Class acts as category. Objects are individual instances of classes. For example, objects like Beethoven and Rantanplan can be instantiated from a same class Dog. The Dog class embodies all descriptive (attributes) and behavioral (methods) characteristics of a Dog object, and all the messages a Dog object can receive or send. A class was compared to some sort of factory which "manufactures" objects, a kind of mold. Each object created is an instance from its class or molded with the same class. Objects work through their defined operations (dog can watch over a house, bite foreigners, etc.). Operations can be activated by the object itself or by another object through activation messages. Messages are the only means used by objects in the object world to communicate with each other and evolve.

The act of building a class in programming environment allows n instantiated objects $\mathrm{X}_1 \ldots \mathrm{X}_n$ from class C to share the same code snippet defined in object languages without having to replicate a code. This sharing mechanism is limited to operations, not attributes. Each instance has memory allocated for its own set of instance variables, which store values peculiar to each instance.

Class factorization and classification step is early at the design level of a product. Composition/aggregation hides complexity at run-time, at user level. While making a composite object, a manufacturer must assemble all pieces together so he or she can master this complexity dimension.

2.4.5 Unique Messaging Mechanism for Object Interaction

In the object world, all communications are reduced to a unique and universal messaging mechanism that simplifies singularly the way objects interact. For instance, when writing the procedural sequence:

$$\text{int a, b, c; } \text{a} = 1; \text{ b} = 2; \text{ c} = \text{a} + \text{b};$$

We can reason, in object technology, as having a thread object T responsible for executing the following thread:

1. int a, b, c

   T instantiates three "variable" objects named a, b, c

2. a = 1

   T sends a message to object a *and urges* a *to initialize its data property to* 1

3. b = 2

   T sends *a message to object* b *and urges* b *to initialize its data property to* 2

4. c = a + b

   T sends *messages to* a *and* b *to get their values, performs the addition* (3 = 1 + 2) *then sends a message to object* c *and urges* c *to initialize its data property to* 3

Message is a useful metaphor. An object behaves like a social actor in this respect. It has a particular role to play within the overall design of the program, and all objects act independently, each inside its own task script. The "actor view" changes the way we reason with objects. Instead of calling a function a function in conventional procedural methodology, we send a message to an object requesting it to perform one of its operations.

> For instance, in an object-oriented programming language, some methods are fairly standard: a *constructor()* method builds the instance, a *destructor()* method

kills the object, a *draw()* method produces an image, *get()* and *set()* methods are called *accessors* as they allow to access to read/write internal variables.

All objects issued from a same class do not have the same state at a same moment; they share only the same behavior.

2.4.6 Inheritance as an Economy of New Class Description

The easiest way to explain something new is to start with something old. This is an economy of concept description by avoiding redundancy, an element of complexity reduction. Object technology allows making a new class from an existent class. The *base class* is called a *superclass* and the new class is its *subclass*. The subclass adds *differential properties* to the base class but keeps all of the original definition.

From an implementation viewpoint in object language programming, nothing is copied from the base class to its subclass. Instead, the two classes are connected so that the subclass inherits all the methods and instance variables of its base class. In semantic networks, the connection between the base class and its subclass is named "is-a" relationship.

A subclass modifies slightly the behavior of the original class as it adds new methods and instance variables. We actually get a newer version with more possibilities through this inheritance process. The new version may simply override some old methods, and change or update the way of doing things. One of the great features of inheritance is the possible coexistence of both old and new versions inside a same system. If we have done some work with an old version of a development platform P6.0 and we do not want to develop newer applications with this old version, then with a newer P7.0, we can still let the two versions coexist inside our system. Such functionality is current in object technology. Versions coexist in the same system without any risk of inconsistencies that can jeopardize older works.

2.5 Object Paradigm Applied to Control Applications: Reactive Systems, Control Arrangement

A real-time system is first a reactive system (Harel and Pnueli, 1985). A reactive system has the following organization:

1. *Presence of an environment*. A reactive system has at least implicitly three subsystems: the system to be controlled, the controller, and its environment.

2. *Concurrency*. All subsystems evolve in parallel.

3. *Presence of inputs and outputs*. A reactive system reads information from their sensor inputs and acts on the environment through their actuator outputs.

4. *Closed-loop*. A reactive system is a closed-loop system that uses feedback information to control its outputs dynamically.

5. *Continuous operation*. A reactive system is a continuous process without a defined end point. It is called a never ending process.

6. *Synchronization burden*. A reactive system observes variations of its own settings, scrutinizes the behavior of the environment, and affords responses or corrections very quickly. A reactive system has the burden to synchronize itself to its environment. It differs therefore from an interactive system by the fact that, in an interactive arrangement, the two partners communicate to each other but there is no constraint or burden put neither on the synchronizing aspect nor the preponderance of any partner for controlling the situation.

A real-time system is therefore a reactive system whose timing are clearly specified.

Control (Curtis et al., 1992) is a very old engineering notion that traces its roots to the industrial revolution of the last century. Control applies to feedback and reactive systems to bind the output to control variables or parameters.

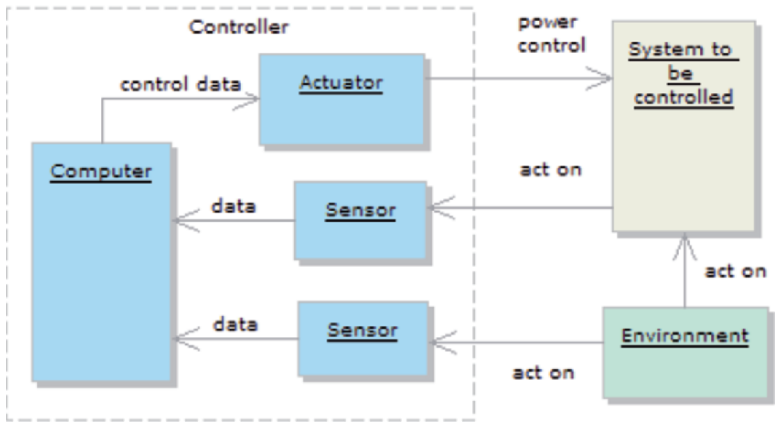

Fig. 2.5.0.1 Typical arrangement of a reactive system. Control includes sensing, actuation, and computation, mixed together to produce a working system. Actuators act on the system to be controlled. Output sensors sense system outputs. The environment acts on the system but the system perceives environmental disturbances via Environment Sensors. All these information are fed as inputs to the Computer that calculates in real time the necessary correction to feed corrective data to the actuator to maintain correct outputs

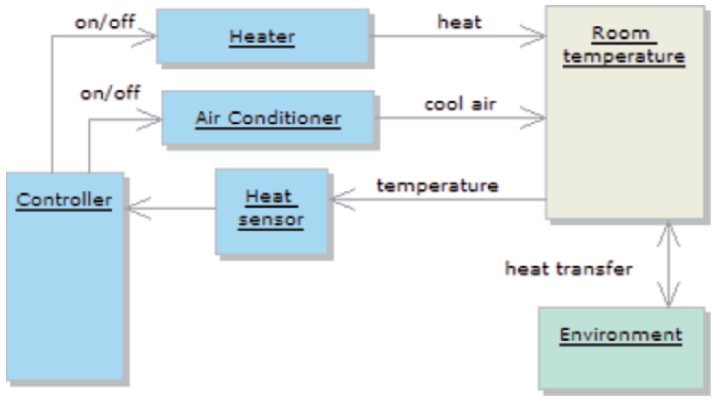

Fig. 2.5.0.2 Thermostatic control built on the principle of a reactive system

Figure 2.5.0.1 represents a typical control arrangement. If control was assimilated in the past to transfer functions (Bode or Nyquist plots, gain phase margin problem), modern control uses algorithms and computerized feedback. Control plays an essential part in the development of domains like electrical and mechanical engineering, and computer sciences. Examples of control arrangements can be found in applications such as power consumption and production, communication, Internet message routing, transportation, manufacturing, factory automation, aerospace, and military applications.

Many tasks performed by a manipulator arm in a manufacturing environment as arc welding, spray painting, continuous soldering or deburring, require that the end effector follows a defined trajectory. The typical arrangement described in Figure 2.5.0.1 could be adapted to a fast real-time system, for instance a welding robot which follows a welding path on a metal sheet. The "seam tracking" can be described as follows.

> When the robot is welding, the camera (Environment Sensors) scans the joint ahead of the welding torch. The camera feeds constant information containing either camera co-ordinates or path correction information to the robot (Actuators) to guide the welding torch. As this is happening while the robot is welding, there is no delay time between the scan and the weld execution (real-time aspect). If the joint changes along the path of the weld, the robot is able to accommodate for the variation (tightly coupled control). The arc welding system of this example is composed of a robot arm, the welding torch, everything just necessary to perform the initial function (welding). Others components, such as sensors, computer, control algorithms and actuators, guarantee that the work can be done efficiently, correctly. The Computer controls the operation of the system by taking information from sensors, compares the result against the desired behavior, computes corrective actions based on a feedback model and actuates the system to effect the desired change.

Tight control systems are often real-time and embedded systems as modern control systems are equipped with high performance processors capable of enormous amount of decision making and control logic. Many control systems are embedded by their application spectrum.

Social systems may also be control systems that require real time and embedding devices distributed in wide and multidisciplinary systems.

> For instance, control is a critical technology in defense systems, in the fight against terrorism. The technology makes use of micro systems and micro sensors to detect threats before they cause damage.

If, in the past, traditional view of control systems consider a control system as a single process with a single controller in an electromechanical environment with a lot of copper wires closing the feedback loop, multidisciplinary trends and modern problems recognize control systems as an heterogeneous collection of physical and biological information systems with intricate interconnections and interactions, not necessarily "copper wired" together. The media used to close the loop can be Hertzian waves and the components of a system can be distributed all around the planet including space.

2.6 Object Paradigm Applied to Database Applications. Data Modeling: Handling Database Complexity

Data are the life-blood in modern economy. We collect data every day, use it to decide, to guide our behavior. The tremendous amount of data available dictates efficient processes to classify, store, retrieve, and use them. If physical drawers and files are sufficient at home, business or governmental organizations need to automate tedious manual business processes and secure data repositories. Database business has been the most flourishing and profitable activity in the second half of the last century (about 13.8 billion in 2005 according to Gartner [available at: www.gartner.com]). To manage data, we need database which is more than a simple repository. Data are organized, indexed, and classified to allow quick access via queries. Databases need a good data model and high performance software motor called the DBMS, either relational (R type giving RDBMS), object types (O type giving ODBMS), or combined type as ORDBMS (Object Relational DBMS). Prerelational databases like hierarchical and network models are still in use.

A data model is a conceptual representation of data as structures. Data are modeled as entities/objects with their attributes, relationships. Data models focus on what data are required and how they should be organized rather than what operations could be performed on the data, relayed to the role of DBMS. Data designers, in this sense, have a better role than control and process designers: they must orchestrate only the *structural view* of systems and delay all the tricks of dynamic table joining at the query phase to the DBMS.

There are two major methodologies used to create a data model: the Entity–Relationship (E-R) (Chen, 1976) approach, its derivatives, and the Object Model leading to object databases. Relational database, with theoretical foundations as Codd algebra (Codd, 1990) and his well-known rules of normalization, is a mature and well-proven technology. For constructing, maintaining, querying, and updating data, it uses a long-established SQL (structured query language) standard, which has been adopted by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI).

> To stress at this introductory stage the differences between the relational and the object view of data, consider a banking application. If we build a teller machine to service a person, the object view is perfect for that application because the Person object is identified by its name and its account number. Other information about this person includes his address, a list of his accounts with all their individual attributes, a list of recent transactions, and so on. All this information can be organized as a hierarchical view of data related to the business relationship that this person holds with the bank. That is the perfect view for the teller which corresponds to the object description.
>
> However, at the end of the day, for the clerk who needs to count how many transactions happened that day, what is the total amount of deposits and withdrawals, how much money he still has in his strong safe, he has a totally different view of the data. Data are transaction oriented, not person oriented. In this case, a relational, "table", or "spreadsheet" view is more appropriate. He needs an account table that is joined to a transaction table, a person table and so on. Tables are more appropriate to answer to questions like: how many, how much . . . assorted with conditions because the tabular form is easier to handle while counting, summing.

Relational database includes a collection of data organized as tables, with columns representing data categories and rows representing the data records, for example, a list of product orders containing columns of product type, customer code, date of sale, and price charged to customers. Users can enter data rows in chronological order and view data from multiple perspectives by reindexing the table, for instance from the lowest to highest price.

Relational database is a good candidate for managing a huge amount of alphanumerical data of nearly the same size and type, but its rigidness comes from the unique implementation structure such as *tables* storing records of values (that is why relational is called "value-based"). Record identities are fabricated by adding keys (primary, candidate, and surrogate). When applying this tabular structure for storing semantic data structured as an ontology forest of trees or multimedia data, we see the limit of the relational model, and naturally look for other alternatives. Moreover, in the way relational tables are designed, information is distributed among several tables, and a time consuming and costly operation of joining tables must be triggered to get the answer from queries.

> A database expert said that if we model all the pieces of a Boeing or an Airbus plane with relational tables of screws, molded parts, plastic pieces, we must decompose the aircraft into tables and then join tables to reconstruct the aircraft.

This metaphor explains that not all data are best modeled with relational tables. In engineering, multimedia data are more efficiently stored inside object databases which binds objects through a richer and complex set of relationships. The resulting storage structure bears a resemblance to natural structure, something like a big grape of objects linked together inextricably like a complex semantic network, but surely not a spreadsheet. Each solution tends to occupy a niche in the market and for the moment the niche of RDBMS is quite huge compared to that of ODBMS that has not got through the step of reaching a critical mass in the marketplace. The success of relational models comes from many facts:

1. *Nature of applications*. Most database applications are transaction based and this situation will not change in the future. A bank, an insurance company, or a stock market is not concerned with the hierarchical and network structures of data.

2. *Rigorous mathematical formalism.* Relational model is supported by Codd algebra that allows some preliminary verification to be made.

3. *Easy model.* The relational model is implemented with tables. Spreadsheet and tabular forms are intuitive for everybody, particularly for financial analysts or business developers. The object model is powerful but is really too complicated and needs special training.

4. *Large operational base and maturity.* RDBMS has occupied the market for a long time and currently supports nearly 80% of database applications. Even if object or another paradigm seems to be promising, generally, people are reticent to change if they do not see any business opportunity. Companies involved in database loathe scrapping their systems for a new technology unless it offers a compelling business advantage.

5. *Dynamics of the market.* ODBMS manufacturers slow to propose an equivalent of RDBMS solution at the end of the last century with full OQL (object query language) support, high reliable environment. Furthermore, ODBMS needs a huge amount of memory to cache data in the client application's memory to eliminate extra call to ODBMS back end. Some technical problems still remain in the research domain.

   - There was a lack of object specialists. Object databases appeared as black art.
   - The mapping of relational schemas directly to ODBMS is not straightforward as inheritance is not natural with tables.
   - Object paradigm itself penetrated the market very slowly. Older versions of UML itself needed important enhancements that arrived finally only in 2003.

- RDBMS manufacturers offered attractive extensions to support data such as audio, video and images.
- RDBMS manufacturers proposed alternate solutions with ORDBMS.

The solution must be adapted to the nature of the problem to be solved. So, object databases will have their niche when ODBMS finally reaches its maturity and when researchers or software companies arrive to solve main technical problems.

Data modeling produce schemas. Large database projects may contain tens of thousands of entities/classes. Database modeling is still a job of an artisan and data schema grows with daily needs by patches and often without any evolution plan. Team members are often not permanent, so huge database is often a nightmare to maintain and diagrams frequently unreadable. Sometimes, reengineering is the only way to recover a badly maintained database becoming unmanageable though it could be perceived initially as a good design. If schema redesign is already an important step, the recovering process of data between two inconsistent sets of schemas is often a job of computer artists.

The solution of managing complexity in huge database passes through complexity reduction techniques of layering, hierarchy-zing and separation of concerns, either in a centralized or decentralized implementation model. Frequently, some management cultures assert that centralized control is better control. From a technology standpoint, this is justified by data integrity and from an economic standpoint by unnecessary replication of redundant systems. But, redundant systems may enhance data security and cache replication accelerates data access. Decentralized systems suffer from problems of linking multiple systems. Moreover, once decentralized, data sharing may never occur; technical concerns and local human culture are closely related. Decentralization puts human behavior and culture inside the initial problem. Independence, controllability, communication, database coherence, data security, etc. are parameters that must go inside constraint analysis phase. From a technical standpoint, Internet federation is a good example of a working decentralized implementation.

2.7 Objects and Databases: Bridging or Mingling?

At the implementation level, object software and relational databases are built for different purposes, and probably it would be easier to find a bridge between the two worlds than unifying them under the same development tool. At the conceptual level, more interesting things can be done and will be discussed in the uniform methodology. The incompatibility comes first from the application concern.

> A car is described by the engineering staff as a hierarchical system with many levels of aggregation/composition. When an engineer asks his system "what are all pieces that constitute a motor?", an object design allows the system to reach

automatically all components by navigating from the first level of composition. Object technology does not deal only with structural aspect of a system. It can handle functional, dynamic aspects at the same time. So, the state of the motor may be determined dynamically by computing the states of all components that make up this motor and this principle may be conducted recursively through all levels of the composition hierarchy. The object view appears in this case as the most attractive natural design.

Opposite to the engineer who studies only one motor in real time to optimize his motor, the director of the storehouse of spare parts of the same company may need a table view of the same data since he has for instance 20,000 spare parts of such a motor in his stock and he must compute how many pieces he must manufacture to support the spare part market. The table view will look more natural in his case.

So, inside the same environment there exist two specific views of apparently the same reality *Car*. In fact, there is very little connection between the two problems. The semantic structure build by the engineering staff describes how a Car object is connected to all of its parts and the maximum number of different objects that we can instantiate through this structure is often 4 or 5 (4 wheels, 5 passengers, etc.). The storehouse of spare parts is concerned with a large inventory of pieces. In fact, a person who works in this store may ignore completely that a mechanical with reference X must go with another with reference Y three levels higher, only their ID numbers, the number of pieces in stock, their costs are relevant to his concern. If we take the engineering model and instantiate tens of thousands of pieces out of this model, we must come back to the tabular structure to store all the piece descriptions and when we want to find out information, table jointure made through a hierarchical structure of tables are more difficult than having a model of tables laid out with classic relational model.

Accessorily, if a mechanical piece must change its dimensions or its description, there is a need for an update that must be propagated to maintain the coherence of the whole system.

A logical approach would be “respecting the nature of things.” Each problem has its optimal solution and solutions of problems are in different domains. Bridging between domains by creating a “bridging domain” is sometimes a better approach then mingling them. Moreover, the “unification‘” or “uniform” view of everything sound, always well, and, as a first reaction, people suppose that these magic terms are equivalent to process simplification, cost saving, and better business. In this sense, some terms are “political” or “managerial” and we must be very careful with them. When looking at the title of this book, we are concerned also with a “uniform” methodology. It is up to the reader to make the same critical analysis before approving it.

2.8 Object Paradigm and Component Reuse

Reuse attacks the complexity reduction at the design level. Complex systems can be built on complex components that have already made their proof in

the past and validated by the market. Reuse is a very sensitive and passionate research subject. Naysayer and fan camps are crowded and arguments on both sides are solid. We do not really want to waste the reader's time by entering this large debate. We think that a technology or a way of doing things must be mature and validated with time, so intrepid researchers need naysayers to alleviate their passion and these two opposite forces are both needed to get closer to the truth, that is, as said in the foreword, always questionable.

Reuse in the small level was already a known issue. Modular programming already promoted code reuse and libraries encouraged compiled code reuse. In a large context, software products could be built entirely from reused components. The virtues of that kind of thinking are obvious since a lot of software no longer would need to be written from scratch; and as components have already passed numerous quality and reliability tests, people expect a significant enhancement of productivity and quality.

Large reuse inside a company is already a reality. Many pieces of a car are standardized inside car companies to give customers the illusion of diversity. Reuse outside a company is a difficult issue because it would be hard to find a component that fits perfectly to a targeted application unless people ask the component vendor to make an OEM version especially for their company. Some experience of external reuse is already current on the Internet (OpenDoc, OLE, ActiveX controls, COM and CORBA objects, JavaBeans, etc.). On the design side, *design patterns* (Gamma et al., 1995), reusing design commonalities is an emerging idea, though not yet mature. Reuse may be at a compiled code level, source code level, design level, documentation level, data level (database replication), or cultural level inside or outside of a company. Reuse is also a cultural issue because if we organize our job so that newcomers can be productive immediately, we are practicing reuse.

Reuse problems (Biggerstaff, 1992; Krueger, 1992; Frakes and Fox, 1995; Sonnemann, 1996) may come from the fact that a component editor does not adhere to good object-oriented design practices and makes poor ad hoc implementation. In this case, reuse is not the issue but problems must be found elsewhere. Reuse success needs management commitment, investment strategy, organizational structure, and staff experienced with reuse concepts. It is also an attitude coopted by developers, system architects, and project managers. Managers who typically want to see progress earlier should plan and gratify reuse effort. External reuse (outside a company) needs standards, so organizations like OMG are very important in the process. Reuse must be planned, preferably, early in the development process, at the requirement analysis phase as an important requisite. Very often, its usefulness is not perceived as mandatory by developers. The following questions (some marked with * are inspired from the work of Frakes and Fox [1995]) stress the most important factors influencing reuse success.

Table 2.8.0.1 Survey questions to test the commitment to reuse concept in an organization

Questions: in your organization	*Comment or remedy*
Are developers masters of core object technology?	Employee training. Project manager must be a good object designer
Are reusable assets actually used and how are they found valuable?*	The past is the future of the present (Japanese proverb)
Do developers make use of programming languages which provide reuse support (e.g. by supporting abstractions, inheritance, strong typing, etc.)?	Watch your development environment. Adhere to good object-oriented design practice. Employee training
Does your organization make use of CASE tools that support reuse?	Buy the right CASE tool
Do developers prefer to build from scratch or do they make efforts to find available components?	Change organization culture
Do recognition rewards increase or promote reuse?*	Gratify reuse effort. Develop incentive
Do legal problems inhibit reuse?*	Consult good lawyers
Is there any repository for code libraries and components?	Create database and document repositories for this purpose
Are you interested in Model Driven Architecture for reuse?	Complete reading this book

*Inspired from the work of Frakes and Fox [1995]

We need experienced project managers who know how to properly evaluate risks and opportunities, to bring their teams through the constantly changing technological and business world. Reuse is also a cultural issue and can be practiced at any level in the organization. Avoid generalities, platitudes, or evident development patterns; developers must learn through experience how to design, implement, maintain, and reuse software components and frameworks. To practice reuse, we must first have available and good materials designed for reuse. So, it would be time to review our best projects and give them a second life.

2.9 Mastering Development Complexity: A Roadmap Towards Model Driven Architecture

The MDA is an industry-standard architecture developed by the OMG in late 2001 (Available at: www.omg.org). The MDA focuses primarily on the functionality and behavior of the application or system, and separate the technology in which it will be implemented. Thus, it is not necessary to repeat the process of modeling the application's functionality and behavior each time a new technology comes along.

The MDA unifies every step of the development of an application. It separates clearly *Platform Independent Models* (PIM) from one or more *Platform Specific*

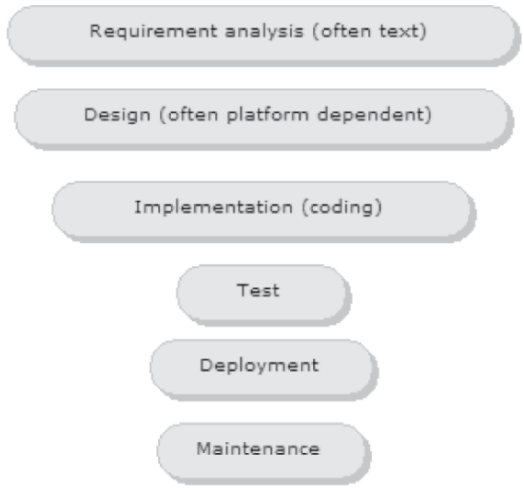

Fig. 2.9.0.1 Classic development roadmap. The design is often platform dependent. It takes into account all the requirements specified at the previous phase. The implementation is often assimilated directly to the coding phase

Models (PSM). Portability and interoperability are supposed to be built into the architecture. The PIM remains stable as technology evolves, extending and thereby maximizing software return of investment. More precisely, the goal of the MDA is to separate business and application logic from its underlying execution platform technology so that changes in the underlying platform do not affect existing applications. The evolution from one model to another normally needs a *model transformation* that can be done automatically by a tool assisted eventually by a human operator if needed. The benefit of this approach is that it raises the level of abstraction in software development at least in the early stages. Instead of going directly to platform-specific code, software developers focus on developing models that are specific to the application domain but independent of the implementation platform. MDA, contrary to its name, is only a conceptual framework. It does not define any particular software architecture or any architectural style.

Figure 2.9.0.1 displays a classic development roadmap. Figure 2.9.0.2 shows a development scenario compliant to the MDA concept.

In Figure 2.9.0.2, the PIM package is derived partly from the domain model that stores the whole assets of development knowledge, classes, logical architectures, schemas, and patterns. The new application under development will create new domain objects that enrich, in return, the domain model. The PIM is then transformed into several PSM models. Theoretically, if a common PIM is used as a basis to generate applications for two different platforms, these two applications will share the common PIM and will therefore

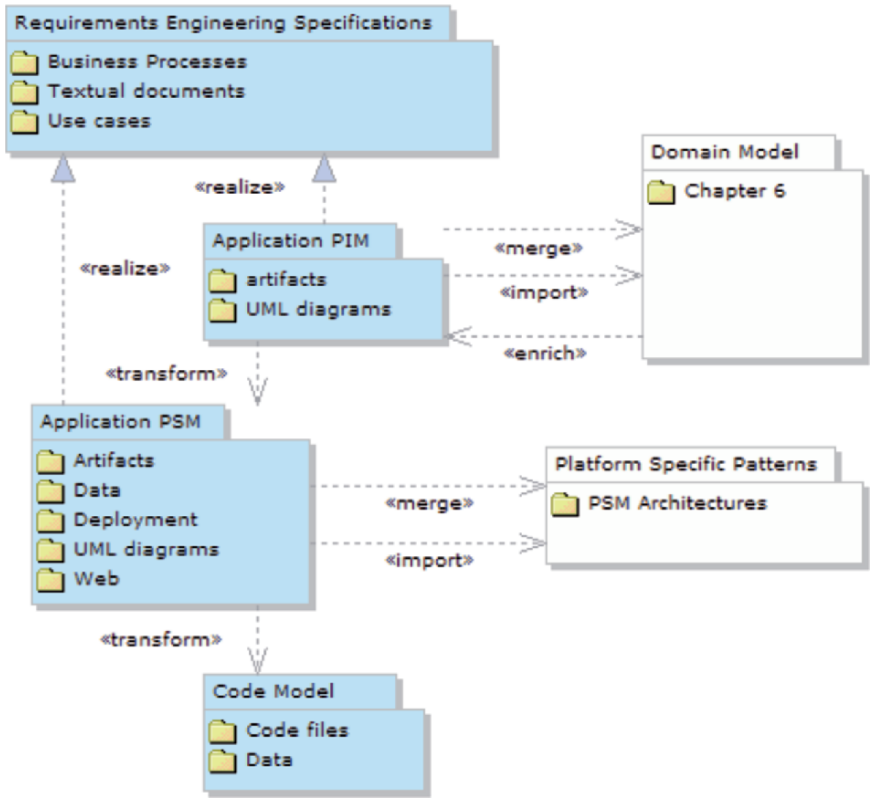

Fig. 2.9.0.2 Instance of development scenario compliant to the MDA (to be compared with the classic development roadmap) that enacts high reuse. As specifications contain both platform independent and platform specific, specifications must be taken into account by the PIM and PSM. Reuse assets are materialized by Domain Model and Platform Specific Patterns. Packages are shown with some of their contents. More than one PSM can be built from one PIM

have increased chances of interoperability at the logical level. To enforce this interoperability, bridges can be created to finalize the process as shown in Figure 2.9.0.3.

The MDA also addresses the problem of middleware proliferation. Middleware is a general term for any software that serves to glue together or to mediate between two separate and existing programs. Often found in a distributed environment, middleware is a layer above the operating system, above the application programming interface (API of Windows) but below the application program. Middleware masks some heterogeneity that programmers of distributed systems must deal with. They also mask networks and hardware intricacies. If an operating system hides hardware details and provides a homogenous programming model to programmers, we can say that middleware masks complexity details in a distributed environment. Examples of famous

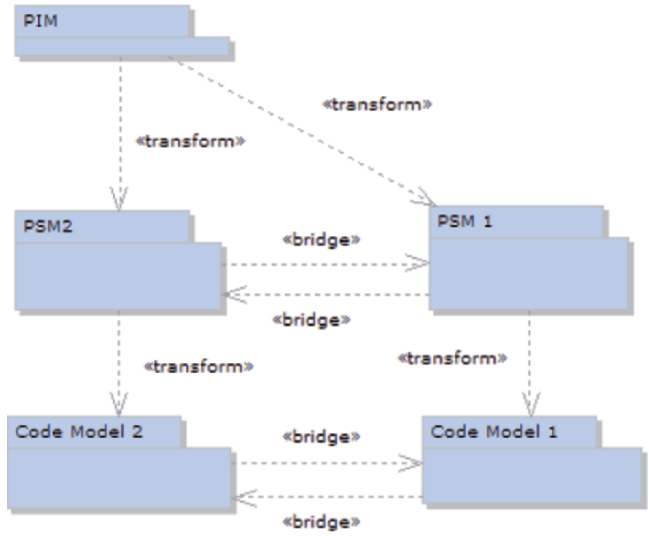

Fig. 2.9.0.3 Transformations and bridges between models. From one platform independent model (PIM), we can make several platform specific models (PSMs). Bridges at platform specific level and at code level can be created for interoperability

middleware are CORBA, Enterprise Java Beans, XML/SOAP, COM+, and .NET.

MDA-based development involves large initial investment in configuration, transformation mechanisms and transformation rule sets, assisted by humans at the early design phase. The transformation could be more easily automated at coding phase (PSM) as programming languages, grammar, and platforms are formal and known issues. If development time should be longer for a first application or for a company without any development assets, development time should decrease for subsequent applications developed inside known domains and platforms.

To sum up, the MDA makes a distinction between the logic of business, the logic of the application, and the platform deployed for this application or this business. The MDA proposes a framework to develop and create models and possibly later, create code directly from these models.