

## Chapter 6

### The Uniform Concept

Complex systems consist of a large number of objects/agents/components that interact in an intricate fashion. The dynamical behavior of such systems can be understood only with a correct model thoroughly decomposed into manageable chunks. Nowadays, with the power of computer simulation and software, more and more complex systems are attempted in many fields, stretching across natural, economic, social sciences, etc. Furthermore, complexity reduction techniques needed for analyzing complex systems are fundamentally the same across disciplines. There is currently great interest in bringing together researchers with different backgrounds for a truly multidisciplinary collaboration. The question is “could we really conceive a unifying modeling formalism that can be used across multiple disciplines?” Our research attempt targets to make use of the object paradigm as the starting point. As the UML implements a graphical notation to be used with the object paradigm, it is natural to include the UML inside the original package.

At the starting point, we have tried to answer some challenging questions as, for instance, “is there a solution to model for instance a whole robot made of electrical, mechanical and software components inside a same and smooth modeling tool like UML to reach an in-depth interpretation of the whole system?” or “is there a uniform way to describe a biometric installation in an airport, taking biometric information on humans than compare to data in a database and detect potential threats?” Doubtlessly, we cannot draw a mechanical plan, nor an electronic circuit with the UML, but we can explain for instance how a software decision is translated into a path change in the tool tip of a welding torch held by a robot and how a deviation in the trajectory of the same tool tip is translated into an information that alerts this machine to trigger a path correction algorithm. In this closed loop control experiment, there is a series of interactions between objects that must be described, not only in terms of vocabularies proper to each participating discipline, but with new revisited concepts of objects and messages. This approach is currently in use in computer simulation as all objects can be reproduced in a computer screen and the whole simulation world works only with messages between objects.

This proposal does not mean that engineers or practitioners belonging to other disciplines must change their way of thinking and describing things in their proper disciplines, but suggest that if a multidisciplinary team desires to understand and later simulate a complex system involving objects from various domains, there is possibly a way to access this unified description, via the object concept, augmented by some fundamental proposals of the uniform concept exposed in this chapter. It is worth noticing that the idea is a research proposal, and all interesting contribution of researchers in the future to enhance this concept would be highly appreciated.

## **6.1 Elements of the Uniform Concept**

### **6.1.1 Elements of the Real World: Objects and Their Interactions**

In a real world, the notions of class and role are inexistent. There are only objects and their interactions. To reach multidisciplinary objects, we need to clarify some attributes:

#### *1. Material or immaterial*

- (a) A dog or a house is a material object that we can touch and feel. Knowledge or soul is immaterial object. We cannot relate immaterial objects directly to matter (they can have a host object instead).

#### *2. Visible or invisible (e.g. air, light, electric, or magnetic field)*

- (a) A car is a visible object. The air or the oxygen we breathe is an invisible object. An invisible object can be immaterial, e.g. an electric or magnetic field. The light is considered sometimes as material (photons), sometimes as immaterial (waves), sometimes visible (from red to violet), sometimes invisible (ultraviolet or infrared).

#### *3. Inactive or active*

- (a) A pen, a computer mouse, and a roof tile are inactive objects. An inactive object does not solicit any service from other objects. They cannot instantiate any change to the surrounding world. An active object is an object that may initiate change to itself or to the surrounding world. A clock is an active object since it changes its own internal state (the time displayed). A powered robot instantiates changes to its world. When not powered, it becomes an inactive object. An inactive object can become active by induction if it receives energy (induced energy concept explained in Section 6.1.3).

#### 4. *Human or not human*

- (a) This distinctive feature is used as an example in a UC diagram to differentiate two classes of actors (human and not human). In some agent systems, developers may desire to make the distinction between human agents and organizations.

#### 5. *Inert or alive*

- (a) An object is inert if its state does not change with time. An object may be immobile, inactive but alive. Its internal state may undergo a number of internal changes without affecting the external environment. A bottle of wine let open in the kitchen at room temperature is transformed into vinegar so the liquid is an alive object as its state changes chemically with time.

#### 6. *Complex or simple*

- (a) From the modeling viewpoint, a simple object is trivial so we do not have to make a dynamic study of this object as its dynamics is also trivial. This fact explains why we bypass the dynamic study of some objects (they are simply too simple). An object is qualified as complex if its functionality and general characteristics cannot be easily understood without a minimum effort of decomposition and a brief study of its dynamics. Complex objects are often composite objects but it is not always the case. A monolithic object (not composite) can be a very complex object.

#### 7. *Object with missions*

- (a) The mission is simply an operation that is activated when an object is instantiated inside a system. A mission differs from a normal operation since it is a kind of survival algorithm activated at the object creation (this fact is not mandatory as an object may receive many missions that can be activated at run-time). For instance, when we put a battery and reset a clock, the mission of the clock is to maintain and display the correct time until its battery goes out. In most system, the program activated after the initialization and reset operations is a mission. Agents in organizations are often activated with missions. They do not need any regular activation signals or messages to accomplish what they are programmed to do.
- (b) Real-time systems are programmed mainly as event-triggered systems. For instance, when a battery is put into a pocket PC, the latter

undergoes a series of initializations to finally reach a state where it enters a loop and waits for commands issued by users. This starting program is a mission for this pocket PC. When a phone call is received by the pocket PC, the mission activates a servicing program that allows users to answer the call. Then, after the call is ended, the control will be returned to the mission.

#### 8. *Autonomous objects*

- (a) Stated simply, an autonomous object is not controlled by other objects or submitted to other forces. It has an independent existence. This does not mean that an autonomous object does not interact with other objects during its lifetime. It can be client or server for other objects but it can decide on its own of the moment or the opportunity to interact with other objects.
- (b) Sometimes, autonomous is understood as independent or self sufficient. But, an independent object can be neither autonomous nor self sufficient.

#### 9. *Others*

- (a) The list is not closed and we can always add more attributes to account for the richness of concepts accompanying objects in a large application.

Figure 6.1.1.1 shows, for instance, an alarm system that illustrates invisible objects that must be modeled in the system to explain some retroactions that a system is supposed to receive.

The particularity of this model is the explicit representation of the media that connects the alarm system to the neighborhood and explain why this system is only useful when there are some neighbors that make an effort to call the police when the owner is absent, instead of turning up their televisions to drown out the sound. This does not make sense to install an alarm system if the house to be protected is located inside an unpopulated region without any neighbor in the vicinity.

This system use the sound as the invisible transport medium that is considered as an object that receives a message from the siren. If the owner is at home, he will be waked up by the alarm and normally he must stop the siren after a while. This explains why the alarm expects a reaction of the owner before triggering another action (for instance calling itself the police).

Each object of the real world, even invisible, must have a representation if it participates effectively in the feedback loop.

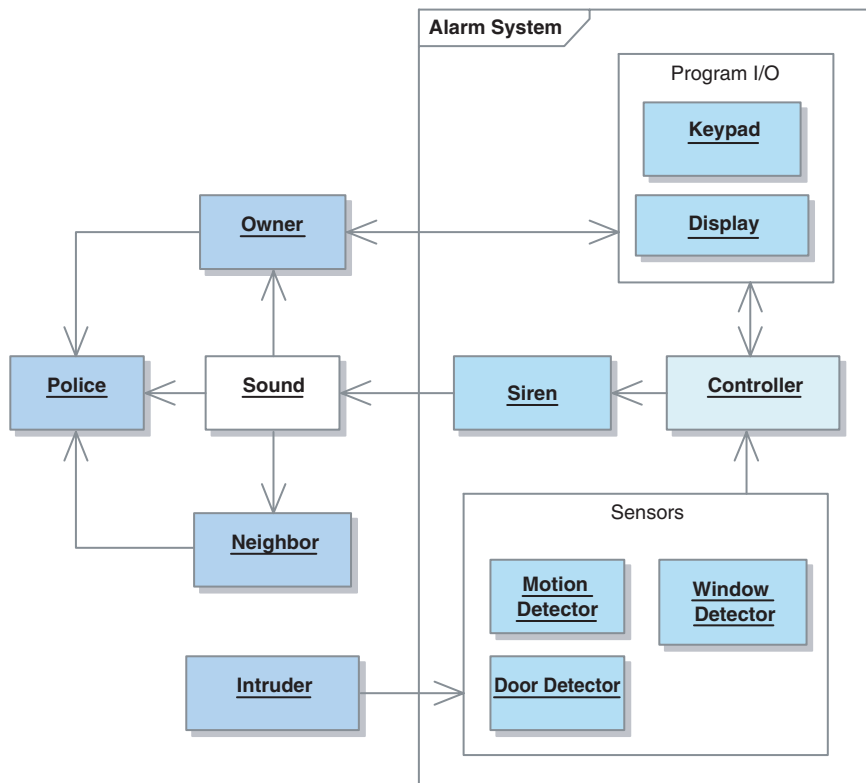


Fig. 6.1.1.1 Object diagram of a Home Alarm System showing the modeling of visible and invisible, human, and nonhuman objects. This representation shows that the Sound produced by the Siren has drawn the attention of the Owner and one of the possibilities to suppress the Sound is the rearmament of the alarm system by the Owner itself

## 6.1.2 Extension of the Message Interaction Model

Object interactions in software are, for instance, sending a signal to activate a process, communicating data. The interaction may imply the exchange of data or control signals, be responsible for the evolution of object states, and may generate new interactions.

In computer sciences, the well known client/server for instance is a specific model of asymmetric interaction. A client issues a request to the server that processes the request then sends the result back to the client. When both can be client and server, we tend towards a symmetric *peer to peer* model of interaction.

If we extend this interaction model to other disciplines now (for instance, interaction between two mechanical pieces, a chemical reaction between chemical

components, a biological evolution, a magnetic field acting on particles in physics, etc.), the interaction model must be reexamined.

Each interaction is considered as the act of communication, whether uni-directional or bidirectional. In the first case, we can distinguish a *transmitter* object to the *receiver* object in this communication. In the second case, we can distinguish whether they are both transmitter and receiver, and whether it is not really necessary to make such a differentiation. When interacting, the transmitter object executes an action that instantiates the interaction.

Moreover, the receiver object may be expecting the interaction or not. Even unexpected, the interaction may change the state of the receiver. According to the rule of encapsulation, an object can change its state by activating an internal operation. So, even if the action is instantiated by the transmitter object, the object paradigm models the interaction as “the transmitter sends a signal to the receiver to activate an internal operation of the receiver.”

For instance, when a person activates a push button, he sends an activation message to the push button to activate the *on()* operation and we write the interaction as *pushbutton.on()*. Moreover, the object paradigm oversimplifies the interaction model by stripping off all operation executed by the transmitter (person) that must activate all its muscles to accomplish such a task.

This model can potentially lead to a deformation of the reality and we will try to demonstrate this fact. This concept of displacing all the interaction to the receiver side to account for this state change is rather general. The link between the emitter and the receiver is reduced to its simplest form, i.e. a named or unnamed event that activates the operation at the receiver side. Let us consider a simple example with a human interacting with a mechanical and electrically controlled device to illustrate this modeling process.

A driver tries to get a ticket in a car parking system after pressing a button at the entering bar. When the ticket is out of the slot, a signal is sent to the gate control that opens the gate bar. In this arrangement, the driver is an object that instantiates a series of action. In object world, it will be described as:

- (a) The object *Driver* presses the object *Button*
- (b) The *Slot* object delivers an object *Ticket*
- (c) The object *Driver* takes the object *Ticket*
- (d) The object *Slot* detects that the object *Ticket* has leaved the slot
- (e) The object *Slot* sent a signal to an object *System* that opens the object *GateBar*.

If we represent conventionally this arrangement with an UML interaction diagram, from a user viewpoint (not as a designer viewpoint as discussed a little further), we get the following diagram (Fig. 6.1.2.1).

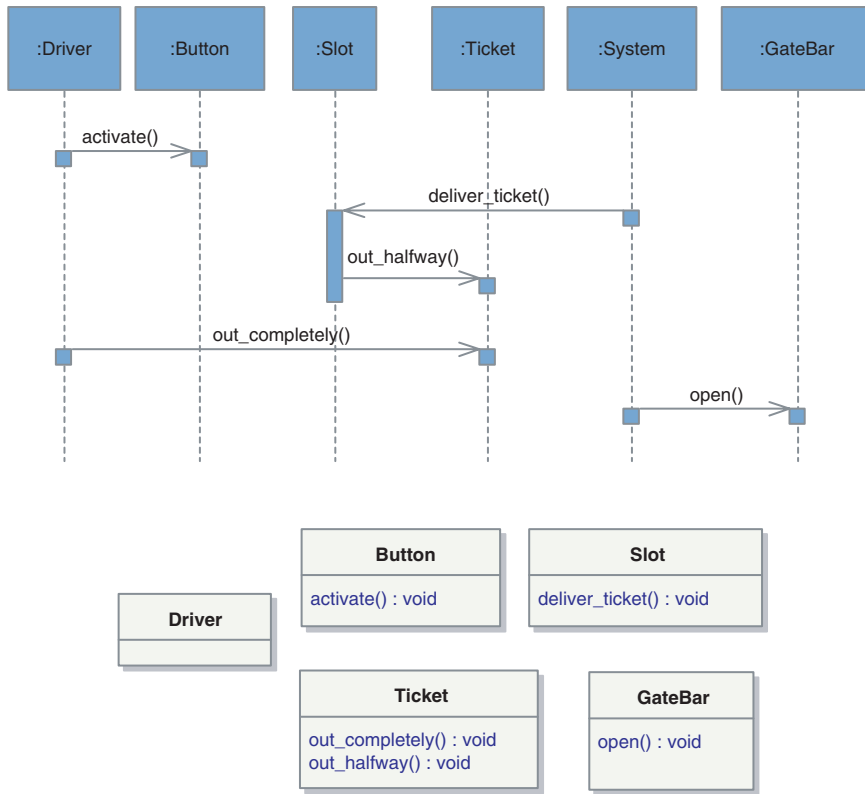


Fig. 6.1.2.1 Sequence and class diagrams showing the arrangement of a parking ticket delivery system from the user viewpoint

The user viewpoint is often incomplete as it models only elements visible at the user interface. This incompleteness appears in this example as discontinuities of the interaction chains. The first time when the system detects that the *Button* has changed its state to on, the second time when the System senses that the *Driver* has taken the *Ticket* out off the *Slot*. The system “behind the scene” is hidden from this representation.

When a *Driver* pushes on the mechanical button, it appears as the *Button* has its own operation *activate()*. The driver just sends a signal and the *Button* activates itself. In this interaction, the *Driver* has executed an action *push()*, but, unfortunately, this action cannot be represented in a sequence diagram in most UML tools of the market.

If we simulate this interaction with a program, in the *Driver* program, we emit a call to the *Button* object with the instruction:

`B.activate()` *B* is the name of a particular instantiated button

This call is an action executed at the Driver side (emitter side). The operation *activate()* modifies the internal status of the Button B (receiver side) from *Off* to *On* as a result of the *activate()* operation.

So, in every interaction, the emitter and the receiver both participate in the exchange mechanism and both objects may undergo state changes as a result of this interaction.

The message interaction model simplifies the content of the message itself if no data are conveyed in the exchange. In its sequence diagram, the UML represents only the operation at the receiver side for simplifying the model but in a full representation, both operations must be expressed.

A full representation has many advantages, for instance, we can list all the actions performed by the Driver at the parking system, in our case, *push\_button()* and *take\_ticket\_out()*. In Figure 6.1.2.1, messages are captured as operations at the receiver but it is not evident to detect immediately the corresponding operations at the emitter side. It appears as the Driver class has no operation defined at all so we cannot read actions performed by the Driver in this interaction diagram. We can however add them manually in UML class diagram.

This reasoning on message interaction model can be repeated for any other interactions of Figure 6.1.2.1. This model may seem somewhat unusual for many of us.

For instance, if a person A hits a person B, A executes *hit()* action, but B is able to execute itself a corresponding action *injure()* that changes its own state. In fact, the operation in B side means that B is receptive to the action of A and the action in B is a mirror of what A is trying to perform or expect as a result on B. But, if we look at the way the representation is drawn, it appears as if B is responsible for its own misfortune.

This example shows clearly a deformation of the reality if we extend this low programming model to express high level problems.

### 6.1.3 Induced energy

Messages in a software program convey activation signals and data through communication channels to wake up processes. Messages in noncomputer systems can convey energy. Energy is then another form of data and is mostly “induced.”

When the Driver in Figure 6.1.2.1 touches the button, he communicates a tiny chunk of energy to the button, so the later is able to move and make an electric contact. The button is naturally a passive object and normally does not have any mean to close the contact itself. But, the tiny induced energy it received from the Driver index transforms temporarily the mechanical button into an active object so it is able to induce change to its microworld.

Similar reasoning abounds in mechanical world. Roof tiles do not have energy but the wind can induce energy to tiles so they can hurt people. A ball in a gun



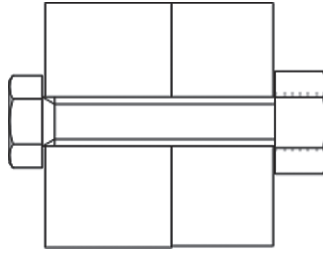


Fig. 6.1.3.1 At the starting point, we have four independent pieces Mechanical\_Piece\_1, Mechanical\_Piece\_2, Bolt and Nut. At the end, we have a monolithic piece shown in the figure

receives energy from a chemical reaction and can kill people. We can easily model the domino effect with the same concept of transfer of a chunk of energy from object to object. So, based on the principle of induced energy, an object, initially inactive, may be temporarily transformed into an active object and it can induce state changes to the surrounding world. This principle is the basic hypothesis to extend our reasoning to other disciplines with the message concept of object interaction.

The model of induced energy can be more sophisticated than the example of the Driver pushing on a button at the parking system. If we want to tighten two mechanical pieces together (Figs 6.1.3.1 and 6.1.3.2), we must bring, with one hand a bolt, into facing holes made in the two pieces, and with other hand, communicate a circular movement to the nut. In this case, the bolt receives a series of induced translation movement and the nut receives an induced circular couple.

The two mechanical pieces send bilateral messages to indicate the pressure during the tightening process. When the right couple is reached within the nut, this information is retransmitted to the right hand of the operator and the person will stop rotating the nut when a correct couple is sensed by his right hand.

## 6.1.4 Use of a Monitor Object

Electrical simulations deal with circuits, active and passive components, circuits, control elements, power devices, actuators, etc. Tools for simulating electrical circuits, both analog and digital, are rather crowded in the market. Without emphasizing any particular tool in this very competitive domain, simulation software has paved the way for helping generations of practitioners to design electrical circuits. Today's circuit designers use automated tools to model and simulate electronic circuit designs prior to fabricating them. However, for handling *multidisciplinary problem*, there are no or few facilities to model non-electrical systems or interfaces between electronic and nonelectronic systems. In the absence of support for various types of physical devices (mechanical,

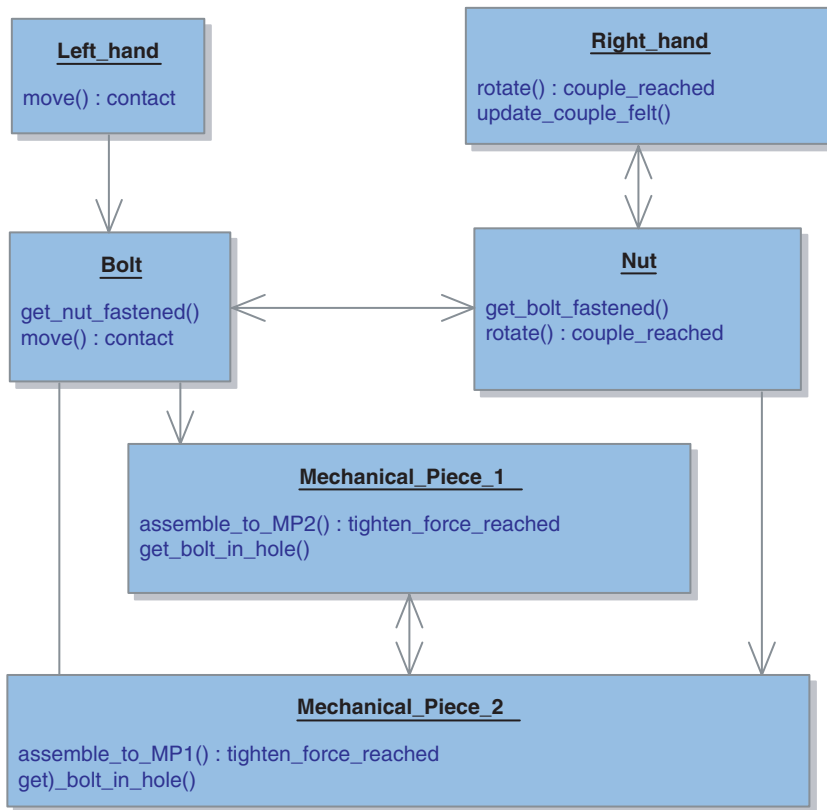


Fig. 6.1.3.2 Translation of mechanical assembly into objects and messages. The Bolt can move because it receives induced energy from the left hand. The Nut can rotate because it receives small rotations from the right hand. Couple sensed by the right hand is induced by couple sensed on the Nut

biological, chemical, etc.) and their coupling, we can make use of the bare object concept and the UML to design and model such interactions.

Figure 6.1.4.1 shows an example of electrical circuit with a switch, an electric fan coupled mechanically to a turbine, a variable resistor, a power supply (voltage source protected by a fuse), and an operator that turns the fan on by closing the switch. When changing the variable resistor from its highest value to its lowest value, we can see the variation of the current and in occurrence, the volume of the air flow. Figure 6.1.4.2 shows the object model of the electric circuit of Figure 6.1.4.1. The interaction model is very different from the corresponding electric circuit as it shows how objects interact together instead of how the current flows through electric components.

As stated earlier, the UML model of interaction is a simplified model as it does not represent operations in the source objects but only at the receiver

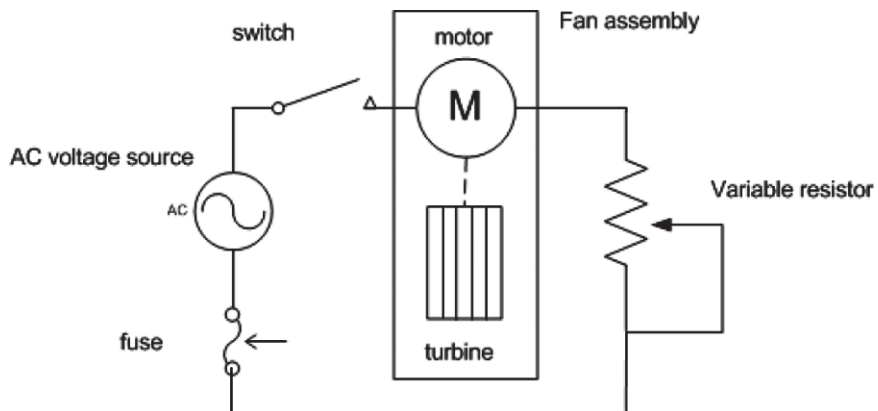


Fig. 6.1.4.1 Variable speed electric fan with its power supply. In real application, a variable resistor is not a good solution. A simple TRIAC is better but we make use of a variable resistor for illustrating the modeling principle

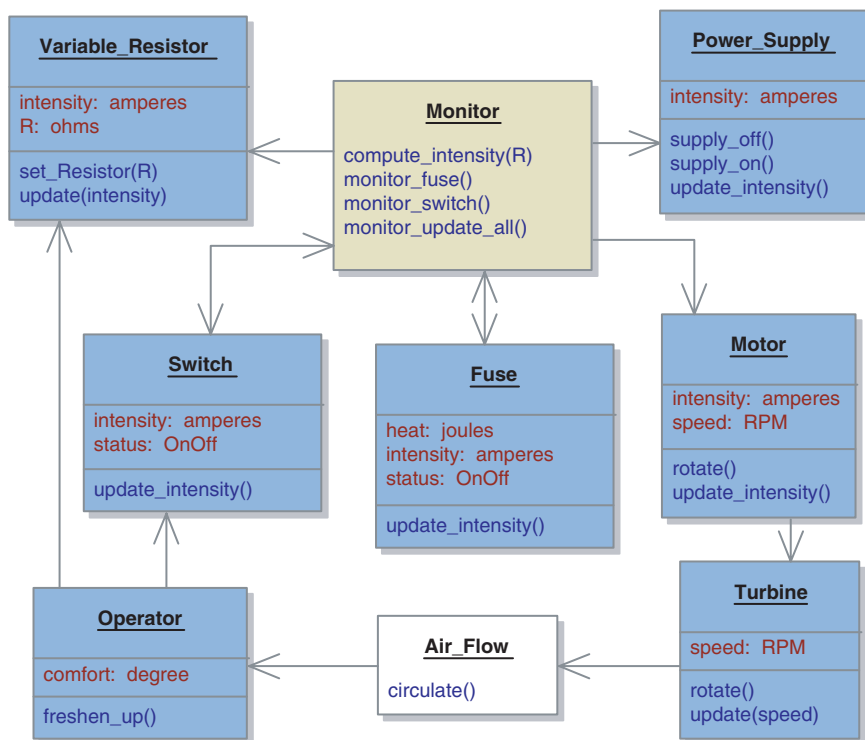


Fig. 6.1.4.2 Object diagram representing a control model. The invisible AirFlow object closes the loop and explains the action of the Operator on the variable resistor. The object Operator was added to evidence the source of interactions (see text)

side. The number of operations is therefore roughly divided by two with this convention.

Operations in Operator like *set\_Resistor()* or *set\_Switch\_On()* or *set\_Switch\_Off()* that explain the role of the Operator in this arrangement are not represented with this convention.

Invisible objects like *AirFlow* are made explicit to explain the *Operator* reaction. The *AirFlow* closes the loop and explains why the Operator needs to adjust the resistor R to get the correct air flow. However, this object is invisible in an electrical circuit. An *Operator* object (also absent while designing electrical circuits) is necessary to show operations performed by humans on the system.

Attributes are traditionally represented in the UML with their implementation type (integer, float, Boolean, String, etc.). In Figure 6.1.4.2, attributes are indicated with their *physical units* to catch at first sight the nature of participating objects, instead of their computer-oriented types (integer, float, string, etc.). So, *Power Supply* and *Fuse* objects have *intensity* with “user defined” *amperes* type. In fact, it would be desirable to be able to specify both the “computer type” and the “physical type” at the same time. Only the computer type is supported for the moment in UML standard. At code generation phase, computer types are needed to make a working system. However, only domain specialists can read this low level. The deployment of MDA allows us to express high-level models differently and adapt it to the corresponding audience.

Links between objects have normally a richer semantics in a multidisciplinary context.

Messages between *Operator* and *Switch*, *Operator* and *Variable\_Resistor*, *Motor* and *Turbine*, are of mechanical nature. Messages between *Turbine* and *AirFlow*, *AirFlow* and *Operator* are of physical nature and invisible. All others are signals and data.

The overall interaction process can be described in this experiment as a series of messages and state changes.

The *Power\_Supply* watched over the statuses of *Switch* and *Fuse* objects. If both their status are *On*, the object *Power\_Supply* determines the intensity with *compute\_intensity(R)* implementing a formula given, in our case, by the Ohms law:

$$\text{intensity} = (\text{Voltage from the AC source} - \text{Voltage on the Motor}) / \\ (\text{variable resistor R} + \text{internal resistor of the AC source} \\ + \text{internal resistor of the Motor})$$

The *Power\_Supply* object sends continuously messages to activate the operation *update(intensity)* that adjusts intensity in the *Variable Resistor*, *Switch*, *Motor* and *Fuse* to the same value determined by *Power\_Supply*.

*Power\_Supply* monitors at the same time the status of *Switch* and *Fuse*. If the intensity reaches an intolerable threshold, the fuse is overheated and it will activate the *blow\_up()* operation that chains with the *supply\_off()* operation to suppress

the electric current when it detects that the fuse is blown up. Through the operation *update(intensity)*, the current in all devices will be suppressed. All those phenomena are instantaneous.

Figure 6.1.4.2 describes a model of interaction, giving to the *Power\_Supply* some “active” properties (sensing *Fuse* and *Switch*, updating intensities in other passive devices). This attitude, based on “real” and “physical” elements present in the system, is acceptable as a power supply is a source of energy in most system and as such, can be considered as an event instantiator. However, the attribution of “active properties” to an element as *Power\_Supply* may be perceived as arbitrary.

An alternative style of modeling (Fig. 6.1.4.3) considers an invisible object *Monitor* that is responsible for executing all actions that result from the application of physical, chemical, electrical, biological, etc. universal laws. These actions must be “nonimputable” to any visible objects of the system. The use of a Monitor object for concurrency control has been described intensively in software literature (Buhr and Fortier, 1995). The notion of monitor used in simulation and modeling of multidisciplinary object systems extends the functionality of this important concept towards to kind of invisible object whose main role is to apply physical laws at real time to all the components of the system, to ascertain that their states will react correctly, in compliance with rules or formulas that can be implemented as operations of the Monitor.

In Figure 6.1.4.3, the operation *monitor\_update\_all()* is a particular operation called previously a mission that is activated by instantiating the *Monitor*. At the beginning, the *Monitor* senses an opened switch, so the initial current will be reset to 0. The operation *monitor\_update\_all()* executes three elementary operations *monitor\_switch()*, *monitor\_fuse()* and *compute\_intensity(R)* that in turn can be decomposed to find appropriate algorithms.

## 6.1.5 All Objects are Made Independent Even with Tightly Coupled Interaction Scheme

Components are independent chunks of hardware or software. All objects, considered at high-level models, must be made independent too, as if they own their proper “active component” internally. In the implementation model, cost optimization and other constraints often downsize the system to only one active component (controller, processor) that controls everything, either on a time-sliced basis and/or a priority scheme.

The suppression of the interdependence between objects in higher models allows us to *well perceive fully the role each object plays in a system*. Moreover, it simplifies the logical model as there is no interference scheduling aspects. In doing so, we separate the logical model from its implementation model.

In the previous example 6.4.1.3 with a *Monitor* object, we have 7 inventoried objects. Each object has inputs, outputs. Internal mission of passive components

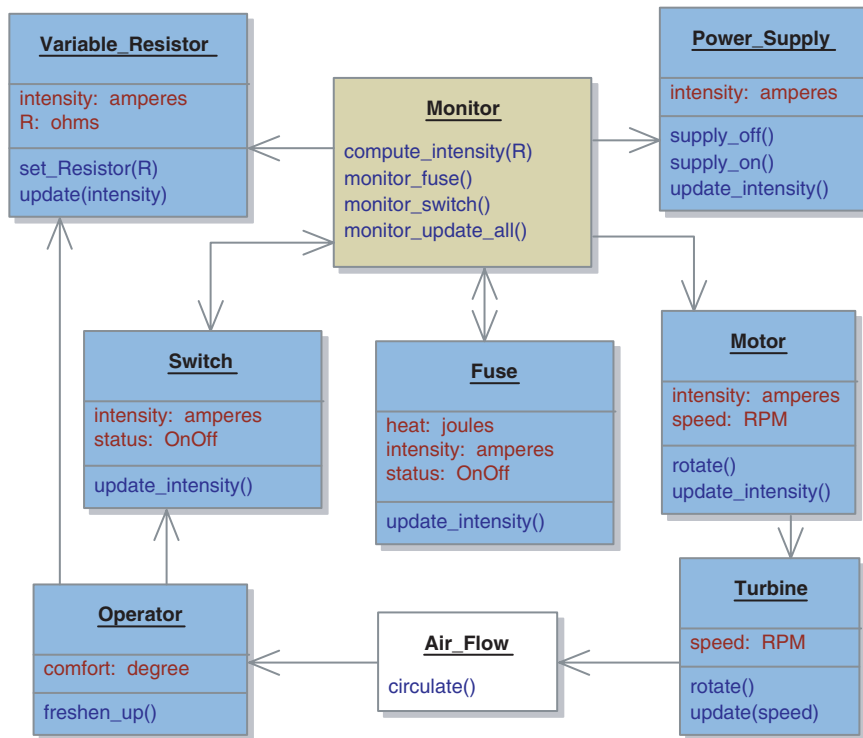


Fig. 6.1.4.3 Use of a Monitor object. When the Operator adjusts the variable resistor R (visible action), the intensity in the motor changes accordingly (invisible action accomplished by the Monitor). The whole chain of reactions that follows the setting of a new R value will be executed by the Monitor as well (computing the new value of the intensity, updating all the intensity values)

consist of monitoring their inputs and reacting appropriately to events according to their nature and the way they are designed to behave. So, if the *Monitor* updates the intensity in the *Motor*, the later will activate its *rotate()* operation to respond appropriately. By mechanical coupling, the *Motor* activates in turn the *rotate()* operation of the *Turbine*, and so on.

The *Monitor* is considered as having its proper processor that activates its mission *monitor\_update\_all()* when instantiating the *Monitor*. This processor works at the lightning speed to apply physical laws. This vision corresponds to the reality. If we set the switch off, automatically, all currents in the electrical circuit will disappear instantaneously. But, at simulation, as we have only one processor to manage the simulation tasks of 7 objects plus regular tasks of the operating system, the simulation process will share time with other computing tasks.

This independence of objects at the logical phase of design plays an important role in complexity reduction at *process level* (the decomposition of a system

in many chunks is a complexity reduction at *spatial level*). This independence can be applied to passive components as well.

If an object *Controller* in an alarm system has three sensors to monitor (window detector, motion detector and door detector), in a monoprocessor implementation, it must visit successively these three sensors (e.g. in the enumerated order). So, there could be a delay if a given system has dozens of sensors with various speeds to monitor. But, in logical design, it would be important to consider, in a platform independent model, as if each sensor owns a processor and all changes are detected immediately with zero latency. So doing, the operation of each sensor is made independent from the presence of other sensors. The latency problem in real time system will be taken into account later at implementation phase when scheduling the task of the processor.

*Tightly or loosely coupled* are terms used initially in multiprocessing system to characterize the communication flows and the degree of dependence between two or more computers (or processing nodes) in their achievement of collaborative tasks. Tightly coupled communications occur, for instance, between a processor and its memory, between two processors inside a biprocessor system, between a processor and its DMAC (Direct Memory Access Controller).

In a tightly coupled communication, computer nodes need to be built close to each other in order to support a high bandwidth of data exchange. At the opposite end, loosely coupled computers need only to communicate erratically, so a network topology is often deployed in this case. Internet communications are of loosely coupled type and intermediate computing nodes can be inserted between two communication points whereas tightly coupled systems often need direct connections without any data replication to optimize the speed of data exchange.

However, the two schemes may be mixed intimately in a system. A network of computers may contain tightly coupled components (computers) connected through a network that supports loosely coupled communications.

For modeling, objects are always considered as independent, despite the fact that they are involved in tightly or loosely coupled schemes. The flow of interactions is denser and faster in a tightly coupled than in a loosely coupled scenario.

### 6.1.6 Cause–Effect Chain

Cause–effect chain analysis is important in computer debugging (Zeller, 2002) and is generally a main concept used in all branches of sciences, particularly in environmental studies. A system is made of a large number of objects. A coarse grain analysis highlights only main objects so the conclusion may be erroneous as the decomposition process is not completed and all objects are not brought into light. The search of a failure in a system needs a thorough decomposition since the simplest piece of a system may be at the origin of a failure. To diminish

the number of test runs, the search may be started by a coarse grain analysis and a thorough decomposition is made in the neighborhood of the suspected origin of the failure. Causal knowledge is a preoccupation of cognitive science as well (Waldmann and Hagmayer, 2005). Sometimes, correlation or conjunctions of events replace the notion of causality if the number of parameters is abnormally high or cannot be identified practically.

The cause–effect chain is a concept that highly impacts the modeling process as the latter is directly dependent of the decomposition process or divide and conquer mechanism. The main conclusions that could be drawn when modeling with cause–effect chain in mind are:

1. *Rush conclusions are systematically suspected in a coarse grain system.* When a system is not decomposed and all elements are identified, the real cause is unknown and erroneous conclusions are highly improbable. When designing a solution to solve a problem, the design phase should be brought to its end.

Let us model a conversation between two persons A and B via the public telephone network that we believe is highly reliable. In a very hot discussion, if the conversation is canceled abruptly, both may conclude at this moment that the opposite person is impolite and has hung up. But, if we model the phone line with two wireless telephones, we may discover that a dog has broken the line in another room while playing with the child.

2. *The cause of failure may exceed the frontier of a system.* Often, when a failure occurs with a product in the market, the natural tendency is to incriminate internal components of the product itself, less by the way it interacts with other products. This short view may come from the bounded frontier of the investigation.

If the grass is wet, it has rained. But if we look at the state of grass of the neighborhood, we can discover that somebody in our home has inadvertently put the sprinkler on.

3. When getting back to the cause of an event or an action, the nature of the initial cause may have nothing to do with the current problem observed. Bronchitis may cause cough that in turn causes insomnia. The mechanical domino is the only process that may have the initial cause similar to the last action but more generally, the nature of effects are very different from step to step in the cause–effect chain. Described in terms of domains, a causal model is made of a set of nodes, a set of links between nodes, and a conditional probability distribution for each node. The nodes can thus cross the frontier of several domains before reaching the current failure or problem observed.



## 6.2 Requirement Analysis Model

### 6.2.1 “What to Do?” Phase

A rather rich literature covers this phase of development (Zave and Jackson, 1997; Sommerville and Sawyer, 1997), our intention is studying how to deploy this phase with UML tools and packing it into the most condensed but comprehensive form using graphical tools instead of huge amount of texts. This phase is an important part of the system design process involving the client, the whole team including business analysts, system engineers, software developers, and managerial staff. The primary purposes of this phase are:

1. Organizing and analyzing requirements into a form that can be verified and accepted by the Client
2. Producing comprehensive documents and diagrams that *can be used by the engineering staff as input to design*
3. Elaborating *criteria for validating the end product*. Generally, these criteria are put in contract between the Client and the Developer
4. Estimating *project Cost, Resource needed, and Time* for each activity involved in the process (CRT parameters).

To achieve these goals, business analysts and system engineers have to listen to user needs, and then create a complete and unambiguous specification document. Actually, this phase is performed very differently from one organization to another according to the organization culture and habitude.

From the Client side, many scenarios are possible; we mention only two extreme limits. Current practices will be located somewhere between these two drastic positions.

1. *The Client prepares a whole book specifying all the requirements.*

This attitude is often seen in governmental institutions and large companies. When large and costly projects are concerned, the Client generally takes all the effort necessary to document itself on the feasibility, available technologies, and at the limit, he has chosen himself the technology and the Developer side acts as simple Execution side.

From a technical viewpoint, the effort at the Developer side is oversimplified as the requirement analysis phase is already or partially made and the only thing is to compete with other developers for the project cost, timing, and QoS.

2. *The Client emits only vague ideas of the final product and explain his project informally (e.g. orally).*

This attitude is often observed with small and short-term projects, with companies that want products but have no internal human resource to establish a requirements document. Sometimes, the utmost effort of the Client side leads to some pages with a list of brief descriptions disorderly organized.

The second case does not mean that the Client will accept everything as design and difficulties could occur for the Developer side at the horizon, so the role of the cautious Developer is to build a complete specification to join the first case. Of course, the Developer may propose many proven solutions currently offered by his company to lower the development time or to deploy reuse mechanism to cut down his proper investment and optimize his profit. He can eventually charge the Client for building the specification and this practice is rather common and recommended as the client of the second type are likely to drop out the project after knowing its real cost than clients of the first type that have already a good idea of the project, its cost and has already committed to go ahead with the project.

### 6.2.2 Parameters of the Requirements Engineering Step

The success of a product or a system is the degree to which it meets the purpose for which it was initially intended. This “purpose” goes beyond the simplistic definition of goals or objectives in natural language prose. The result of this phase is materialized by a business contract signed between organizations before entering the design phase. This contract proves their commitment to go ahead with a project. Everything must be defined in depth in the “what to do” field and occasionally in the “how to do” field if specific or technological constraints are embedded in the project.

There are no theoretical laws or theorems that govern the management of this phase but the conclusions drawn from this phase impacts largely on material, human resources, and even existence of organizations. In the literature, requirement analysis is seen as inquiry-based (Potts et al., 1994). The term *requirements engineering* is often used and is mainly goal oriented (Dardenne et al., 1993; Lamsweerde, 2001). It embraces *requirement analysis*, *requirement specification*, and *production of textual and technical specification documents, artifacts*. So, for large projects, the requirements engineering can be nearly considered as an independent subproject. In the MDA context, it is considered as a *requirement model*. As this phase needs a complex interaction between human actors, recommendations of the requirements engineering step are:

1. *Talk the language of the captive audience.* As said before, this phase involves the client, the whole team including business analysts, system engineers, software developers, and managerial staff. As people come from different horizons, not everything is suitable for viewing.

- (a) For instance, the UC diagram may appear as a very complex formalism and cannot be easily understood by everyone, so, even if a project is analyzed using this tool, conclusions must be reformatted into a more easily digested form.
  - (b) The business process diagram that bears some resemblance with older DFD with inputs and outputs is more suggestive and less problematic than the UC diagram.
  - (c) Managerial staff and client are more sensitive to project cost, resource needed, and timing. Those points must be particularly formatted with a suggestive and convincing presentation.
2. *Conduct the necessary inquiry phase.* The Client is not a technical person. He is often unable to formulate all relevant requirements precisely, and has limited knowledge about the technical environment in which the system will be developed or even used. This is the reason why he relegates the task of building the system or solving the problem to the Developer staff. Even if the Client gives the impression to specify all, experiences show that a lot of inquiries will be necessary to avoid misunderstandings.
- (a) Content *analysis* (Berelson, 1952; Kassirjian, 1977) is a research technique for the objective, systematic, and quantitative description of the manifest content of communication. Content analysis is used in political science, social psychology, survey techniques, communication, advertising or propaganda campaign, etc. Textual information is imprecise, may convey tendentious clauses, and is systematically suspected by content analysis researchers. Techniques in this domain are applicable to formalizing and quantifying the universe of discourse engaged between Client and Developer sides.
  - (b) Very often, the Client ignores the regulation and it is up to the Developer to inform him and updates the specification accordingly.
3. *Fight against the natural tendency of some developers to shortcut to requirement analysis phase.* The technical staff at the Developer side is highly technical, is often biased, and has a clear tendency to consider that they lose their time in endless discussions.
- (a) Experience in our university shows that students that often jump directly to the programming phase are more receptive to requirements analysis based on technical diagrams like UC diagram and BPD instead of textual clauses issued from content analysis technique. The two aspects are all necessary in a requirements engineering process: we can start with diagrams and end up with text.

4. *Reach a complete specification.* At the end of the requirements engineering, specifications must be validated by comparing the whole specification against Client objectives. From a practical perspective, the result at the Developer side is a set of activities decomposed into their utmost details, where Developer can put resources, estimate time, estimate components to be built, components to be acquired, the overall cost, his markup. So, the perception of completeness varies with the experience of the Developer in the domain. Experienced developers can quantify very quickly whereas newcomers in the domain must work hard to reach the same result for the first time (they will acquire experience later).
5. *Build unambiguous specifications that can be used as inputs to design phase. Several techniques or several views may be required.* Unambiguous specifications need often technical tools. As said before, the Client could get lost with the technical level of UC diagram and BPD. Ideally, a comprehensive set must contain both technical diagrams for engineering purpose and accessible textual clauses that can be served as communication support with nontechnical stakeholders. Many views or submodels of a same system may be needed to get a thorough understanding.
  - (a) TROPOS (Bresciani et al., 2004), a tool devoted to agent-based system design, builds the early requirement models based on activities centered around:
  - (b) *Actor modeling*: Identifying and analyzing both of the environment and the system's actors and agents
  - (c) *Dependency modeling*: Identifying actors which depend on one another for goals to be achieved, plans to be performed, and resources required
  - (d) *Goal modeling*: Analyzing actor goals, from the point of view of actors, by using means-end analysis, contribution analysis, and AND/OR decomposition
  - (e) *Plan modeling*: (technique complementary to goal modeling)
  - (f) *Capability modeling*: Identifying individual capabilities of actors.

## 6.2.3 Requirements Engineering

Figure 6.2.3.1 summarizes most important steps of the requirements engineering phase with an UML activity diagram.

*Step 1: Have a very good idea of the Client needs (Elicitation phase).* We start with two possibilities corresponding to the two extreme cases, one with a Client that has established a requirement document and another without any prepared document for the Developer side. As we must, in all

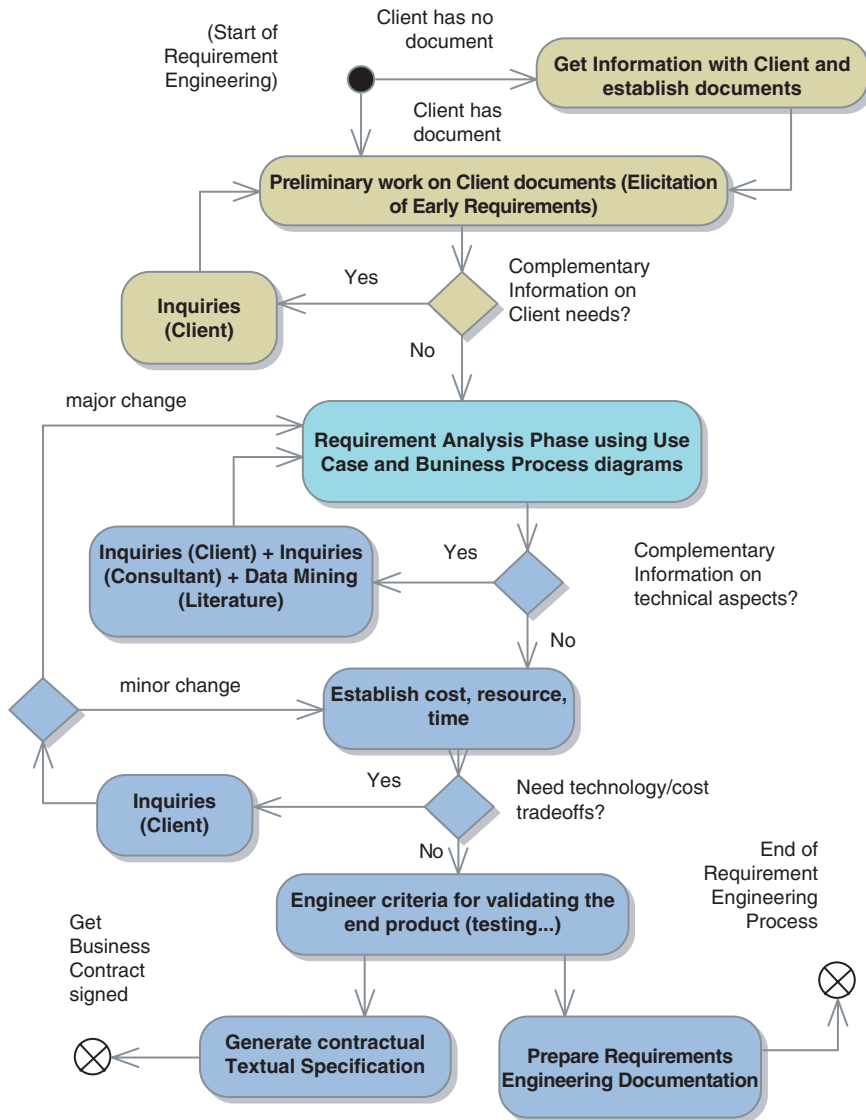


Fig. 6.2.3.1 UML activities diagram showing activities of Requirements Engineering phase. We make use of use case (UC) and Business Process (BP) diagrams for the technical part of this phase

cases, reach the same result and a thorough understanding of “what the Client needs,” preliminary textual documents can be used to explicit the Client intentions at the beginning of the process.

Even in the case where the Client has established himself a Specifications document, this work can never be established with all the precision of a document targeted to be used at the Developer side. So,

this textual document must be revised and completed carefully by the Developer.

The main interaction between the Client and the Developer sides during this step is mainly Inquiries between humans and techniques used are *Content Analysis* of textual conversations.

*Step 2: Start of Requirement Analysis phase using mainly Use Case (UC) Diagrams and BPD (Technical decomposition).* This technique makes use of UC diagrams and BP diagrams currently developed and maintained by the OMG. Other diagrams as Class, Object, Sequence, Activity etc. can be occasionally used as well. But, be careful not to start an early design inside the Requirement Analysis.

Essentially, all activities of the project are identified, structured and decomposed into activities chunks so that Developer can estimate, in the next step, Cost, Resource, and Time (CRT parameters). During this step, the Developer may discover that some aspects are not clearly identified in the step 1 and therefore need inputs from the Client.

As the requirement analysis using UC and BP diagrams are essentially of functional type, the result of this phase is a set of activities structured around some criteria (actor, goal, plan, capability, etc.) and ready for the design phase. To give an idea, less than 100 activities are found for small projects, between a hundred and a thousand activities are current for medium-size projects and if more than 1,000 activities are identified, we are facing a large project (project size can be measured in software, classically, with man-month of number lines of code in any language).

*Step 3: Solving CRT parameters (Budgeting).* In this step, generally, system engineers require the collaboration of a financial analyst. Activities must be decomposed at a granularity such that the Developer can easily find back existing components, discover activities that they currently handle in the past and for which they have enough experience to be able to put rapidly a cost, identify required resources and quote a time.

Once the time of each elementary activity is known, a Gantt chart (a way of visualizing a series of tasks and their dependencies) describing the project schedule can be built for estimating the overall timing of the project. Sometimes, it is necessary to simulate a *work breakdown structure*. The project schedule is just the combination of all the breakdown structures with allowance made for interdependencies between tasks. Specific project management software could be of great help, instead of using only spreadsheets.

It would be advisable at this step to contact the Client and give him some insights about the approximate overall project cost and time. If

early adjustments are needed, they will save efforts to generate detailed specifications.

*Step 4: Prepare criteria for validating the end product (Validation of conformance).* The validation testing is introduced into a project to assure that the final product will carry out the functions described in the requirement specification. Normally, it protects the two parties, the Client by forcing the Developer to engineer the right product or solution, and the Developer by forcing the Client to accept the same product if it passes all validation tests.

Testing scenarios have to be well defined at this step so that conformance is possible to demonstrate and validate. Terms as “appropriate response” or “perform satisfactorily,” if not accompanied by more precise and testable scenarios must be banned. One of the directions that can save a lot of time later would be trying to write the User’s Manual early at this phase as a starting point for validation. But, User’s Manual is not enough for validating a product since this manual mostly gives standard scenarios with “normal usage” of the product, while validation may test for limit cases, error handling scenarios, functional metrics and all mirrors of specification clauses. A validation document, more accurate than the User’s Manual, is something that a Developer can give to the Client and the latter can give it to his technical service to support customers having trouble using the product. Some products in the market include this technical document at the end of the User’s Manual in order to help customers diagnose themselves their problems.

*Step 5: Generate contractual textual documents and prepare instructions for the design phase.* Documents required for the contract varied greatly according to the engineering domain. Generally, we find: overview description of the project, architecture of the system to be built, product specifications, product validation, financial, and legal documents.

Documents for the design team include: overview description of the project, architecture of the system to be built, product specifications, product validation all relevant diagrams developed at step 2.

### **6.3 Requirements Analysis with Technical Diagrams**

This phase can be started once the elicitation process of textual and early requirements is completed and the developer knows what the client wants. Normally, we have a lightweight document containing an enumeration of early requirements containing *goals*, *unstructured description of elements* entering the composition of the system to be built. Such an “unordered structure” is perfectly normal as Client and Developer want to go forward with an idea (or stakeholders’

*Table 6.2.3.1* Subphases of the Requirements Engineering process. We focus only on the first two subphases. The requirement analysis (subphase 2) makes use of Use Case for representing graphical representation of the system to be built. The subphase is unavoidable as we must make a good text content analysis to understand the Client needs.

<i>Subphases</i>	<i>Techniques used</i>	<i>Targeted concepts</i>
1. Elicitation of early requirements (unstructured and unordered data)	Text Content Analysis	Goals. Context description. Early product description. Features of the system to be built.
2. Requirements Analysis	All UML diagrams, mainly <i>use case</i> and <i>BPDs</i>	Activities, actors, their relationships. Actions, activities and their inputs, outputs.
3. Budgeting	Spreadsheet or specialized software	Determine CRT (Cost, Resource, Time) parameters from activities
4. Definition of validation of conformance criteria (Tests specifically designed for acceptance process)	Requirements Validation Tools or manual list of test scenarios	Limit cases, error handling scenarios, and functional metrics.
5. Document Preparation	CASE tools for generating technical and textual documents from diagrams	Legal, contractual and technical documents (Differentiate between “soft goals” and “rigid” constraints)

intentions) but the domain is not still correctly mastered. They have put some main ideas into preliminary documents but the true analysis work has not been started. So, the project needs to be investigated more systematically in order to identify components and activities, organize, and structure them.

The UC diagram (Jacobson et al., 1999) in UML standard has being recognized as formalism well adapted to this investigation phase. The BPD already studied in Section 5.3 can be used to describe inputs, outputs, events around important processes that need to be specified. Both these diagrams are of functional flavor. *Why an alignment on functional view with object technology?* The answer is: This is a phase where technical considerations have to be balanced against social, organizational, and financial ones. We need to get the CRT parameters, so, going further than a rough identification of activities will propel us directly into the design phase, or at least in the first strata of the design phase. As requirements engineering is an incursion into the functional view of a system and not a complete study, it was recognized as a phase where the most and costliest errors are introduced into a system. One of the directions to negate this effect would be a flexibility of the two parties involved to accept some adjustments to the project even during the design phase. To illustrate



the Requirements Engineering process, let us consider early requirements of a Remote Monitoring System of Health Conditions (RMSHC).

*Early requirements of a Remote Monitoring System of Health Conditions:* A Client desires to manufacture a system that limits a patient's health risk, reduces the time and frequency of check-ups in a hospital, and may occasionally serve as pharmaceutical field tests. This technique can bring peace of mind to people giving them the insurance that they are constantly monitored by qualified people. The system is a kit containing a wristwatch Blood Pressure Monitor (BPM), a pill box, and a mobile phone that automatically transmits patient data to an internet center where doctors and nurses can monitor the patient at will. The BPM takes at determined frequency systolic and diastolic blood pressure and pulse readings. The patient can activate manually the BPM with only one push button command to inflate, measure and display. The request can be sent by the doctor or the nurse through the telephone. The pill box is of electronic type. It must keep track of the time a pill is removed from the box. If the patient forgets to take a pill, a reminder is sent through the mobile phone or the Nurse can contact the Patient orally.

The next step will be the analysis of those early requirements.

### 6.3.1 Requirements Analysis with Use Case Diagrams

Some readers may be obfuscated and perplexed about some ideas advanced with the way we make use of UC diagrams; but in the requirements engineering phase, as the system is not built, as we need a *succinct tool* and formalism dealing with fuzziness, it would be more important to be able to answer quickly “what is the project cost?” “how long it takes to develop?” or “what is the existing resource (or to be built) that is not considered as being part of the project but that must be deployed to support it?” than satisfying mathematical considerations.


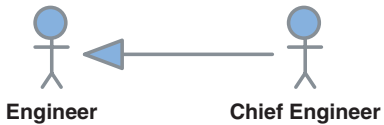
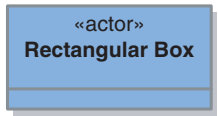
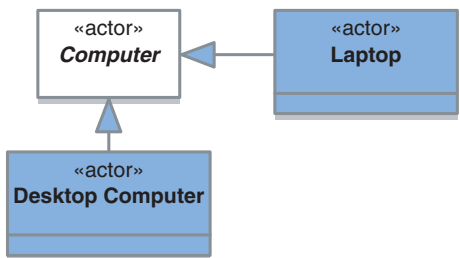
UC diagrams are used as techniques for formalizing roughly functional requirements and factorizing actor roles (Sandhu et al., 1996). Their syntax is already explained in Section 4.16. This diagram is based on three main concepts: *Actor*, *Use case*, and *Relationships* connecting use cases to use cases or use cases to actors. The four most important relationships in use case concepts are *include*, *extend* and *communicate*, and *inheritance*. *Inheritance* relationship is used to factorize actor roles. Table 6.3.1.1 summarizes our limited interpretation of concepts in a UC diagram.

Using use cases as requirements engineering tool is a technique has some limitations (that can be seen as “advantages” if it can discourage premature design) as it addresses only the functional view. Relationships and actors is an attempt to organize and structure this functional view, however, their semantics is rather limited. In order to make this diagram more expressive, some interpretations must be extended and enriched, so some additional extensions are necessary:

1. *Use cases* can represent *goals*, *plans*, *rules*, or *requirements* that all need activities to be realized. Goals and business rules are integral parts

of organization behavior and guide their day-to-day business decisions. Some authors consider them as nonfunctional requirements. In fact, the relationship between goals, rules and activities is rather complex as people cannot really verify how products, manufacturing processes, solutions to conform to business goals and rules if they are not mapped or inserted in

*Table 6.3.1.1* Convention used in this book: basic concepts in use case diagram and extensions of concept interpretation

<i>Concepts and graphical notations</i>	<i>Comments and examples</i>
 <p><b>Stickman</b></p>	<p><b>Stick form Actor</b></p> <p>A stick form actor is used to represent a <i>role</i> accomplished by a human, a group of humans, a human organization, or a mixed system involving humans.</p>
 <p><b>Engineer</b>                      <b>Chief Engineer</b></p>	<p><b>Inheritance relationship between Actors</b></p> <p>In the example, a Chief Engineer can accomplish all the tasks assigned to an Engineer and more.</p> <p>This inheritance relationship is viewed exclusively from the role consideration. A physical person may assume an unlimited number of roles or a role can be played simultaneously by many humans.</p>
 <p><b>Rectangular Box</b></p>	<p><b>Nonhuman or object/system Actor</b></p> <p>A rectangular box actor represents an actor role, the entity behind the actor can be human or nonhuman (this case embraces the human actor role). Generally, it depicts a system, a subsystem, a component, an object, and an undefined system (black box) to be developed.</p>
 <p><b>Computer</b>                      <b>Laptop</b></p> <p><b>Desktop Computer</b></p>	<p><b>Nonhuman actors involved in inheritance relationship</b></p> <p>In this inheritance relationship, a Laptop and a Desktop Computer have all the properties and operations of a generic abstract Computer.</p>

(Cont.)

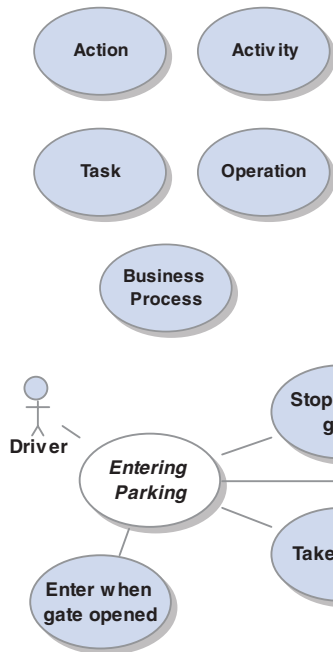
Table 6.3.1.1 (Continued)

Concepts and  
graphical notations

Comments and examples

**Use case representing an action, an activity, a task, an operation, a business process, a collaboration, a scenario, etc.**

An *action* represented by a use case is an important action that can take few CRT parameters, or at the limit zero, but, if not represented, may obscure the interpretation of a use case diagram. (Example: *Log On* use case)



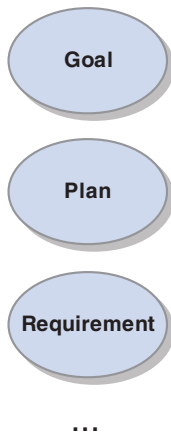
An *activity*, decomposable or not, is the most frequent and usual use for a use case. It can be executed by any general actor, a collaboration of mixed actors. An activity can be a scenario describing a sequence of actions/activities executed in a certain order.

*Tasks* and *operations* are also activities in other domains. In business modeling, *business processes* can be represented as use cases at requirements analysis.

*Example:* Scenario represented by an abstract use case and four use cases representing activities in a parking system.

*Entering parking* is an abstract use case that is "replaced" by four other use cases. This representation avoids counting *Entering Parking* twice.

**Extension: Use case representing a goal, a plan, a requirement, etc.**



*Goals* influence the way products or solutions are designed in a system and impact heavily on all activities of an organization.

*Rules* will be mapped somewhere into manufacturing, business processes or verification processes.

*Plans* are a collection of activities to reach some goals.

*Requirements* will be mapped into specifications that must be realized in the design phase.

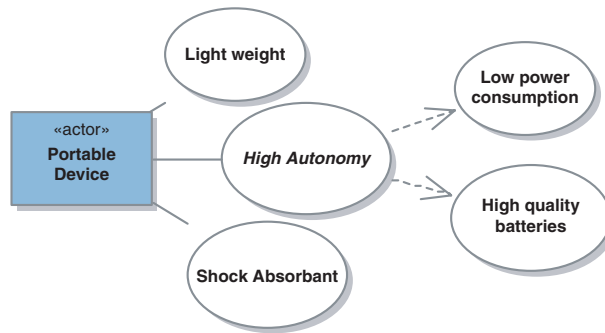
So, all these concepts are of functional flavor even if the relationships are hard to establish without a precise context. We make use of use cases to represent them all. When a use case represents a semantic node (a node in a diagram that, if absent, will obscure the interpretation or understanding of the diagram), have no CRT parameters attached to it, or is replaced by a set of activities in the same diagram, we can simplify its graphical notation ("dashed contour" or "transparent background color") to alleviate its visual load on a diagram.

*Example:* Goals

Table 6.3.1.1 (Continued)

Concepts and  
graphical notations

Comments and examples



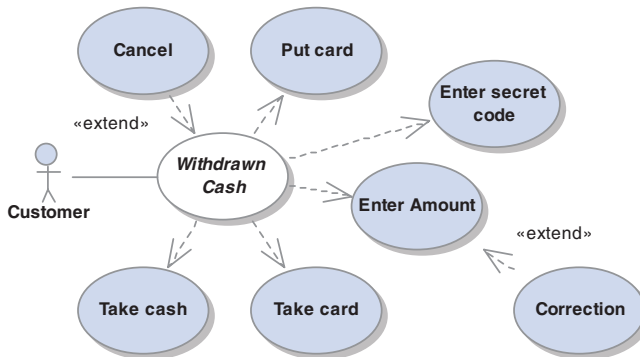
*High Autonomy* is converted into two goals connected with <<include>> relationships. The <<include>> stereotype is suppressed as the line style (dashed) and the orientation identify unambiguously this relationship.

#### **Extension: Communicate is used as an undefined relationship**

The *communicate* relationship can be used as an undefined relationship that can mean *involve*, *concern*, *dependent*, *have property*, *obey to the rule*, *satisfy*, etc. This fuzziness is an advantage as it avoids spending too much time looking for a precise meaning at the requirements analysis phase.

*Example:* The *communicate* relationship in the previous extension is transformed into *satisfy* relationship (A portable device must satisfy Lightweight condition).

#### **Include and extends relationships**



These two powerful concepts structure and connect use cases together and support decomposition mechanism.

An <<included>> use case is embedded inside another use case. It is activated or involved each time the including use case executes.

An <<extended>> use case is an activity that is executed under some conditions. Even if the probability of calling this activity is very low, from a development viewpoint, as CRT parameters must be deployed to handle this case, its importance is equal to that of other regular activities.

*Example:*

In this example, the *Customer* is connected to *Withdraw Cash* via a communicate relationship. *Withdraw Cash* is transformed into an “empty” use case as the main activity is transformed into five more elementary use cases and replaced by ALL decomposed activities (this transformation is not allowed for nonexhaustive decomposition as we still need

(cont.)

Table 6.3.1.1 (Continued)

Concepts and graphical notations	Comments and examples
	<p>compute residual CRT parameters from the original use case). <i>Correction</i> and <i>Cancel</i> are software pieces that must be designed even if they are “exceptionally” activated. <i>Cancel</i> can be operated in any of the five decomposed activities as it is defined at their root.</p> <p><i>Important Note:</i></p> <p>Sometimes, the <i>extend</i> relationship is interpreted as an equivalent of a <i>switch</i> or <i>case</i> instruction in programming language. This fact is only coincidence. The use case diagram is not a flow chart or an activity diagram (it is not a dynamic diagram), so, this interpretation does not hold in all cases.</p>

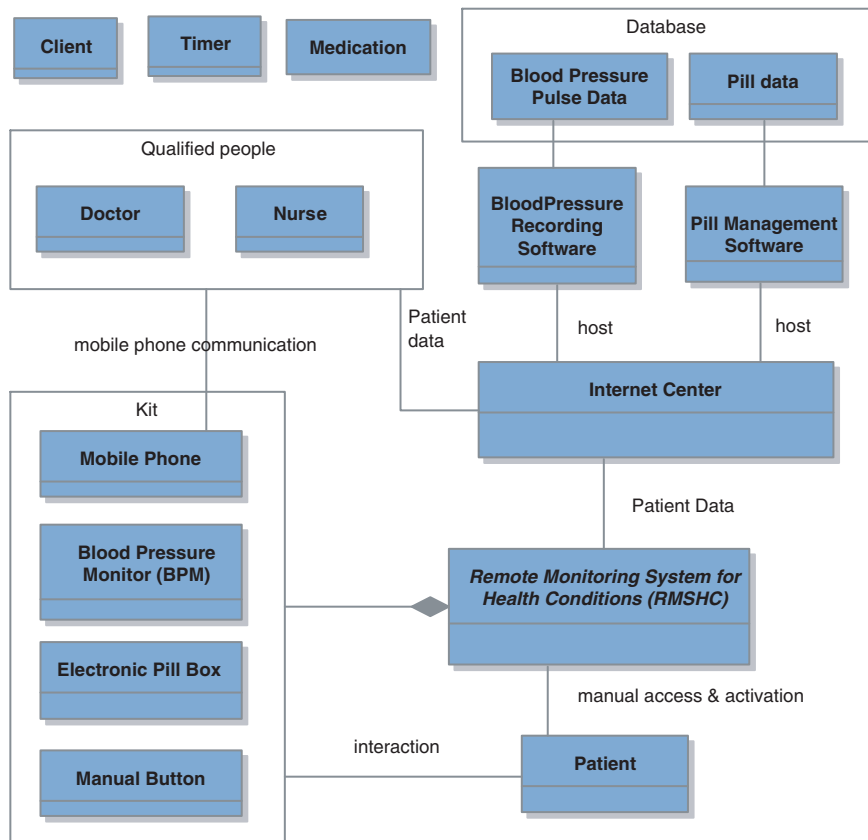
some way inside the whole development process. Goals may appear as required attributes of the end product but the resulting state of a system is the fruit of hard labor and intensive design activities. Instead of involving ourselves in interminable debates, a pragmatic approach is using use cases to represent these concepts but, if the Developer thinks that a given use case does not correspond to a real activity or replaced by some activities elsewhere, its graphical representation can be alleviated and it will be considered as a semantic node.

- (a) When a use case represents a semantic node, have no CRT parameters attached to it, or is replaced by a set of activities in the same diagram or conditions the way the whole process is engineered, we can simplify its graphical notation (“dashed contour,” “transparent background color,” or “abstract” representation) to lighten its visual load on a diagram.
2. The *communicate* relationship can be used as an undefined relationship that can mean *involve*, *concern*, *dependent*, *have property*, *obey to the rule*, *satisfy*, etc. The fuzziness of the model relative to the meaning of this relationship is not really important at this phase and does not impact greatly on the estimated volume of activities necessary to reach CRT parameters.
  - (a) To alleviate the visual load, all the communicate relationships in a use case will be shown as full line without any stereotype and navigability. The relationships *include* and *extend* are shown with dashed lines, the *include* relationship without its stereotype, and the *extend* relationship may be shown with or without its stereotype. The *extend* arrow is oriented from the extended use case towards the extending use case, so this fact distinguishes it from the *include* relationship noted with a similar arrow with a reverse direction.

### 6.3.2 Conducting the Requirements Analysis Phase

The extended conventions allow us to represent all types of requirements: functional, nonfunctional, goal type, etc. inside the same formalism (UC diagrams) and thus extending the expressiveness of this diagram and reducing the textual part to its minimum. To illustrate the requirement analysis process, let us return to early requirements of a RMSHC (Fig. 6.3.2.1) and analyze this textual document, paragraph by paragraph, presuming that this text comes from the Client and is dissected by the Developer.

The textual analysis of Table 6.3.2.1 allows us to identify all actors/objects, use cases, and their relationships with this early requirements specification. Once all the inquiries are elucidated with the Client, a complete set of diagrams



*Fig. 6.3.2.1* Class diagram of the RMSHC system. This class diagram allows us to create quickly separate instances to be used as nonhuman actors in subsequent use case diagrams to support several views of a system

*Table 6.3.2.1* Example of Text Content Analysis in order to establish use case diagrams structuring requirements. Fuzzy clauses must be elucidated with the Client. A more complete table template is shown next. Diagram reference contains diagram numbers (the same concepts may be displayed in multiple diagrams)

<i>Step</i>	<i>Initial text</i>	<i>Concepts</i>	<i>Reformulation</i>
1	A Client desires to manufacture a system that limits a patient's health risks, reduces the time and frequency of check-ups in a hospital, and may occasionally serve as pharmaceutical field tests.	Actor/Object	Client
1a		UC/Activity	Manufacture
1b		Actor/Object	System: Remote Monitoring System of Health Conditions (RMSHC)
1c		Goal	Limit patient health risks
1d		Actor/Object	Patient
1e		Goal	Reduce check-ups time in hospital
1f		Goal	Reduce frequency of check-ups in hospital
1g		Goal	Serve occasionally as pharmaceutical field tests
2	This technique can bring peace of mind to people giving them the insurance that they are constantly monitored by qualified people.	Actor/Object	Technique (abstract)
2a		Goal	Bring peace of mind
2b		Goal	Give insurance of good health monitoring
2c		Actor/Object	Qualified People (abstract, to be clearly identified)
2d	The system is a kit containing a wristwatch BPM (Blood Pressure Monitor), a pill box, a mobile phone that transmits <i>automatically</i> patient data to an internet center where doctors and nurses can monitor the patient at will.	UC/Activity	Monitor Patient Parameters
3		Actor/Object	Kit
3a		Actor/Object	BPM (Blood Pressure Monitor)
3b		Actor/Object	Pill Box
3c		Actor/Object	Mobile Phone
3d		Relationship	Component of RMSHC
3e		UC/Activity	(Phone) Transmit patient data ( <i>automatically</i> is a fuzzy term)
3f		Constraint	Transmission starts each time data is available or at regular intervals? ( <i>to be elucidated</i> )
3g		Actor/Object	Internet Center
3h		Actor/Object	Blood Pressure Pulse Data
3i		Actor/Object	Data Management Software (implicit specification)
3j		UC/Activity	Manage all Patient Data (implicit)
3k		Actor/Object	Doctor
3l		Actor/Object	Nurse
3m		Relationship	Doctor and Nurse are "Qualified Person" (See 2c)
3n		UC/Activity	Monitor Patient Conditions

(cont.)

Table 6.3.2.1 (Continued)

Step	Initial text	Concepts	Reformulation	
4	The BPM measures at determined frequency systolic and diastolic blood pressure and pulse readings.	UC/Activity	Measure systolic pressure	
4a		UC/Activity	Measure diastolic pressure	
4b		UC/Activity	Measure pulses per minute	
4c		Constraint	<i>At determined frequency (unspecified for the moment. Limits have to be elucidated for the design phase)</i>	
5	The patient can activate manually the BPM with only one push button command to inflate, measure and display.	UC/Activity	(Patient can) Activate measurements manually	
5a		Actor/Object	Push Button	
5b		UC/Activity	(BPM) Inflate	
5c		UC/Activity	(BPM) Measure (See 4, 4a, 4b)	
5d		UC/Activity	(BPM) Display	
6	The request can be sent by the doctor or the nurse through the telephone.	UC/Activity	(Doctor, Nurse) Communicate orally (via Patient Mobile phone)	
6a		UC/Activity	Request manual activation (BPM)	
7	The pill box is of electronic type. It must keep track of the time a pill is removed from the box.	Attribute	(Pill Box (already defined in 3b). <i>of electronic type</i>	
7a		UC/Activity	Keep track time pill removed from Pill Box	
7b		Actor/Object	Pill Management Software	
8	If the patient forgets to take a pill, a reminder is sent through the mobile phone.	UC/Activity	Forget to take pills (this use case is abstract as it does not correspond to any CRT parameters but explains only the process)	
8a		Actor/Object	Pill Data (implicit)	
8b		UC/Activity	(Pill Software) Check Pill Data against Medication (implicit)	
8c		Actor/Object	Pill Software (implicit)	
8d		Actor/Object	Medication (prescribed by Doctor)	
8e		UC/Activity	(Doctor) Prescribe Medication (implicit)	
8f		UC/Activity	Enter pill data to Internet Center (Nurse or Doctor?)	
8g		UC/Activity	(Pill Software) Elaborate Electronic Reminder	
8h		UC/Activity	(Pill Software) Send Reminder by Mobile Phone	
8g		UC/Activity	(Nurse) Contact Patient forget taking pills	
Step	Initial text	UC concepts	Reformulation	Diagram reference
				UC1, UC5, etc.



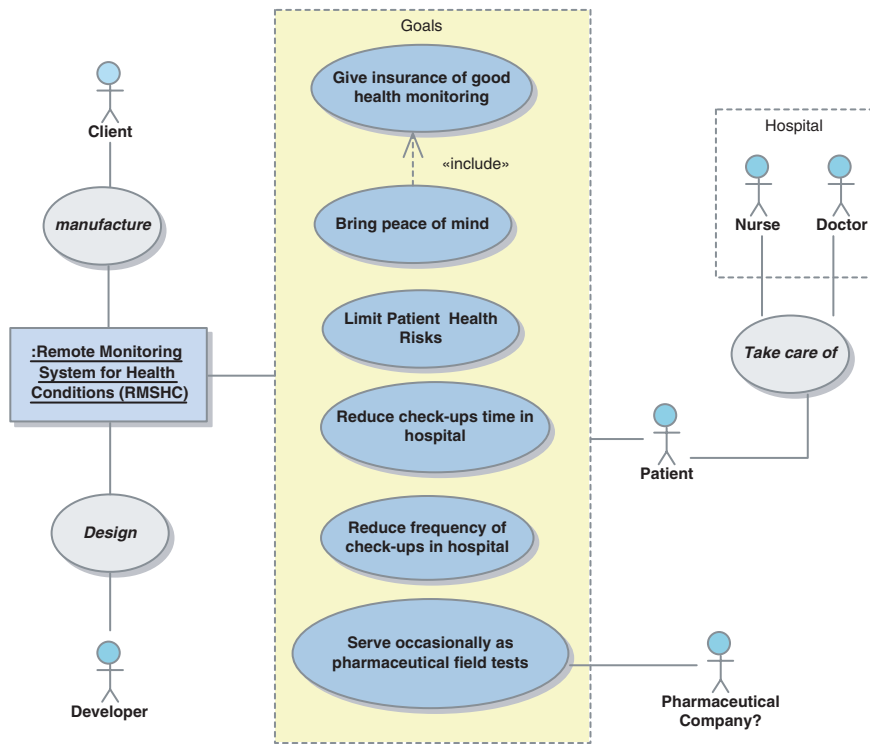


Fig. 6.3.2.2 Use case (UC) diagram depicting goals enumerated for the RMSHC. Except RMSHC system, human actors are created directly from the UC diagram (not from classes) to show another possibility of generating actors (but we suggest to create all actors by instantiating them from classes). Patient is connected to the contour of goals and therefore is connected to all goals, idem for RMSHC. Pharmaceutical Company crosses the contour to reach individual use case is not connected to other use cases

can be built to compute CRT parameters with all use cases identified through this Requirements Analysis phase. When establishing diagrams, each concept identified in Table 6.3.2.1 can figure more than once to support many views of the system.

In Figure 6.3.2.2, the RMSHC is instantiated from a class diagram (its name is preceded by a “:” and underlined). All human actors can be either created from scratch in the UC diagram or instantiated from a class. To implement the second choice, a class diagram must be created to define all classes and afterwards, objects can be instantiated from these classes at will to support all UC diagrams.

The goal *Give insurance of good health monitoring* is included to *Bring peace of mind* to show a possibility of making hierarchies of goals if needed. Goals are shared mainly by two actors, the Patient and the System. If the interpretation at

the left side is “*the System must fulfill all the projected goals,*” at the right side, the Patient “*benefits*” from these goals. The last use case *Serve occasionally as pharmaceutical field tests* (this use case is connected to RMSHC, Patient and Pharmaceutical Company) involves a design agreement between three actors, the Patient serves only as test object. So, the meaning of this unnamed connection varies considerably with the context. As said before, it is advantageous to have undefined relationships to avoid getting stuck with the problem of having to precise all semantics not necessary at this exploratory phase of the development.

Moreover, in Figure 6.3.2.2, instead of connecting all the goal use cases to the RMSHC system itself, we have connected them to the Client or to the Developer, the interpretation would be “these actors (and not the system) *must respect* goals while developing the system.” Semantics is questionable and confuse as goals are attached not directly to the system but indirectly to those responsible for developing them. Essentially, this kind of errors is tolerable and does not impact on identified use cases.

Figure 6.3.2.4 gives information about the management of the Electronic Pill Box, clarify the communication process between the Nurse, the Doctor, and the Patient and highlight activities involved in the management of the medication. This diagram uses the property of the contour to pack the two objects *:Nurse* and *:Doctor* inside a same entity named Qualified People (or Hospital) to save connections. So doing, a unique connection is drawn from Qualified People to *Communicate orally* and another from Qualified People to *Consult Patient Data*. All connections can go through the contour of Qualified People to reach individual objects *:Nurse* or *:Doctor*. Two *extend* relationships are defined within *Communicate orally* to show that this communication is either for requesting a manual activation of the BPM (Fig. 6.3.2.3) or urging the Patient to take the forgotten pills. *Forget Taking Pills* is an abstract use case attached to the object Patient. This possibility shows that use cases with zero CRT parameters can be represented as semantic nodes in order to explain the context and give a coherent interpretation.

In Figure 6.3.2.4, several activities are made abstract as they do not impact directly on the determination of CRT parameters but are there to explain a context justifying the usefulness of some software pieces. For instance, if the Patient forgets to take pills, the Electronic Pill box cannot register any pill taken out of the box, the Pill Management Software will detect this anomaly. Two actions are therefore programmed, the first one is sending a Reminder through Mobile Phone, and the second is sending a warning to the Hospital that urges the Nurse to contact personally the Patient if the electronic Reminder has no effect. The presence of abstract use cases is thus essential to capture the meaning of the whole process.

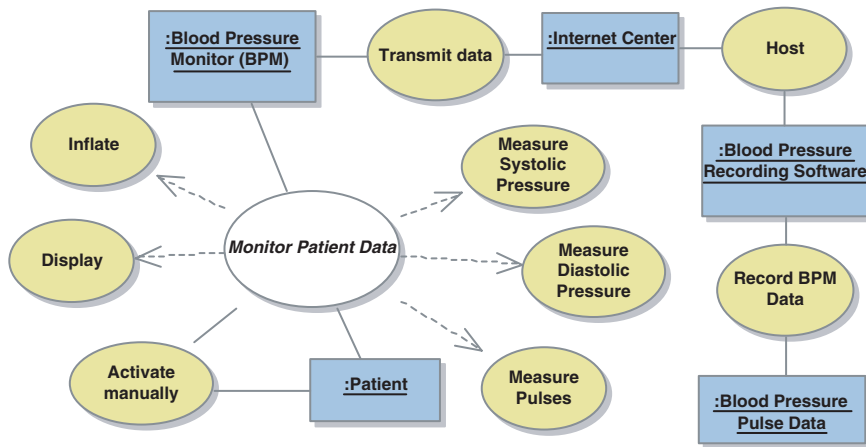


Fig. 6.3.2.3 Activities of the Blood Pressure Monitor (BPM). The central use case Monitor Patient Data is turned into abstract as it is replaced entirely by included components. The Activate manually needs the intervention of the Patient

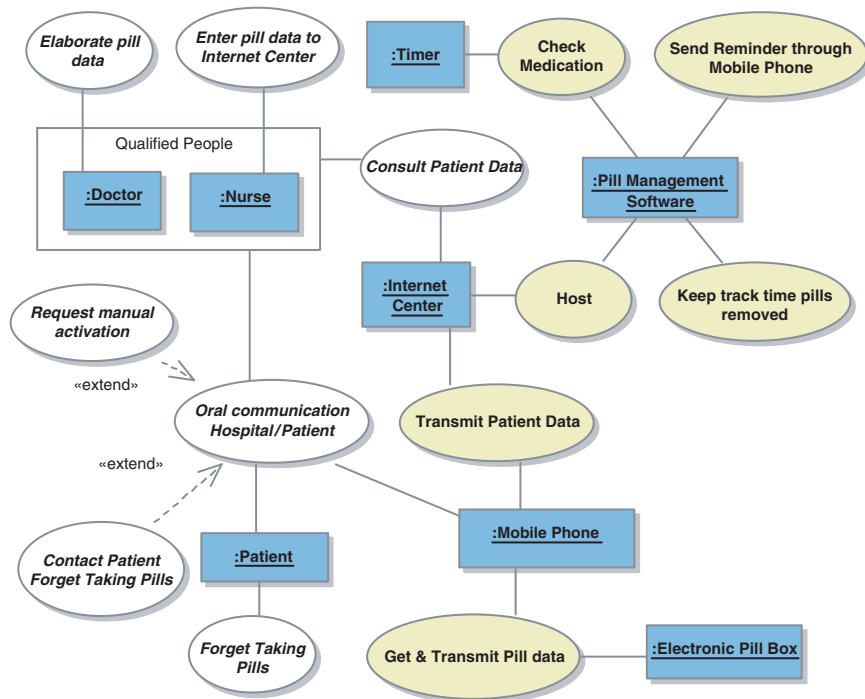


Fig. 6.3.2.4 Use case diagram showing activities related to the management of the Electronic Pill Box

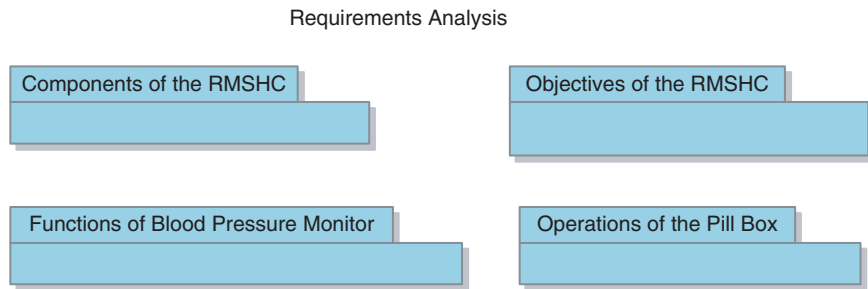


Fig. 6.3.3.1 Package diagram of the Requirements Analysis of the Remote Monitoring System for Health Conditions

### 6.3.3 Use Case Modeling Advices

UC diagram is a very simple formalism and it addresses to the first phase of the modeling process, even qualified by Developers as phase 0 outside the technical model itself. As it could eventually be read by nontechnical persons, is of functional type, and contains very few concepts, it is not easy to make it very expressive. So, the best way to describe a system with use case is like *telling a story*. Therefore, here are some general guidelines to deploy this process:

1. *Diagrams can be organized as a hierarchy of packages.* For instance, for the previous example of RMSHC (Fig. 6.3.3.1), we can create four packages with only one level (a complex project can have more than one level).
2. *Actors/objects are connected to use cases.* Actors are connected directly together only through inheritance (Fig. 6.3.3.2). UC diagrams also accept object style connection between actors but this connection can be transformed into use case connection style (see Fig. 6.3.3.3).
3. *Actors in a UC diagram can be objects derived from classes.* Actors can be created from scratch with the UC diagram. If many actors of the same type must be created to support many views, it is advisable to create a class and derive all actor instances from this class. This method is very handy in Requirements Analysis phase.

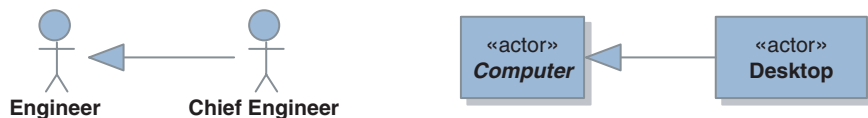


Fig. 6.3.3.2 Actors are connected directly through inheritance only

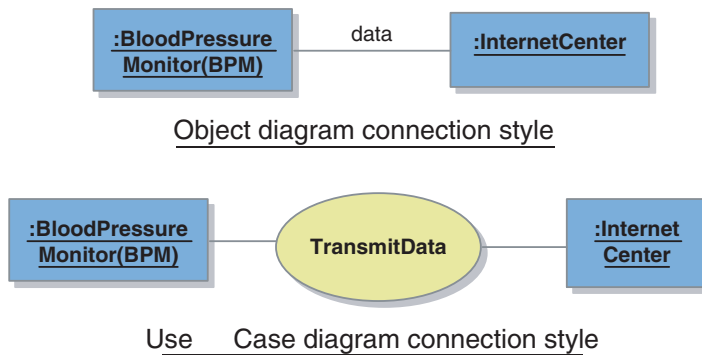


Fig. 6.3.3.3 When actors are connected together, we have an object style connection. In a use case diagram, a use case is inserted between two actors meaning that actors are involved in the activity

4. *Distinguish goals, rules, and competencies from activities.* Goals, rules have no CRT parameters. Conversely, activities have CRT parameters, so goals and rules can be made abstract to distinguish them from regular activities.
5. Use inheritance for actor roles to reduce use case repetition or connection complexity (Fig. 6.3.3.4).
6. *It is not necessary to connect a use case to all actors involved.* The use case model is essentially based on responsibilities, competencies, goals, and activities of main actors. So, only main actors responsible of the activities are relevant. If there are several, the first neighbor will be chosen. A UC diagram will become very complex if each use case is transformed into a "bubble" of a DFD diagram.

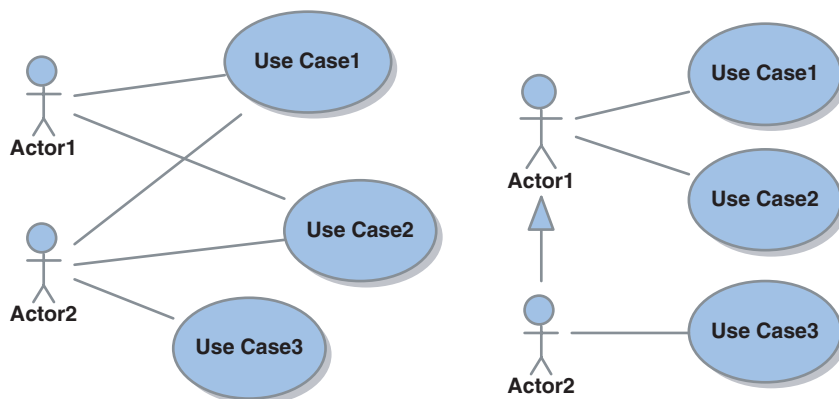


Fig. 6.3.3.4 Inheritance between roles. The left diagram is replaced by the right diagram

- (a) In Figure 6.3.2.4, the use case *Keep track time pill removed* takes data from the Pill Box, make used of the phone for recording, so the temptation to connect this use case to the Pill Box and the Mobile phone is high as the action on the Pill Box fires a cascade of actions leading to the recording process. If such reasoning is repeated for any use case, the connection network would be inextricable. While modeling, a given effect may depend upon a cascade of multiple activities (cause–effect chain). The criterion retained is the responsibility (or the first neighbor within a long cause–effect chain) in this early phase of Requirements Engineering, so only the object *Pill Management Software* is connected to *Keep track time pill removed* as the latter is a component of *Pill Management Software* itself connected only to its first neighbor and responsible host *Internet Center*. As said earlier, the utmost goal is to identify activities with CRT parameters by telling an understandable story.
7. Be careful in adding new relationships to a UC diagram. Normally, we can stereotype any relationship between actors/actors, use cases/use cases, or actors/use cases, but *the tendency to transform a UC diagram into an entity relationship diagram of structural flavor must be avoided*. To explain the composition of the system, a true class diagram may be used to create classes with aggregation/composition and possibly early inheritance relationship between classes.
8. *Well define the frontier of the system to be developed*. As said earlier, diagrams are targeted to tell a story. But a system is generally part of a context and interacts with other systems. The story needs all of them. It is recommended to choose a way to identify parts devoted to the current development (graphical markers, special numbering, textual enumeration, special recapitulating diagrams, etc.). The choice is left to the reader.

### 6.3.4 Using Business Process Diagrams

Some types of problems are best expressed with a combination of BPDs (Figure 6.3.4.1) and UC diagrams, mostly when activities require a clear specification of inputs, outputs, and events. BP diagrams can be seen as high-level DFD of the past with a richer graphical notation. Therefore, it is not necessary to use this diagram for any identified use case. Only high level and important processes containing complementary information that UC diagrams alone cannot afford, are good candidates for such BP diagrams.

In a previous Remote Monitoring System for Health Conditions (RMSHC), a good candidate for a BP diagram is the Pill Management Software Component (actor/object). At this Requirements Analysis phase, the Developer must evaluate

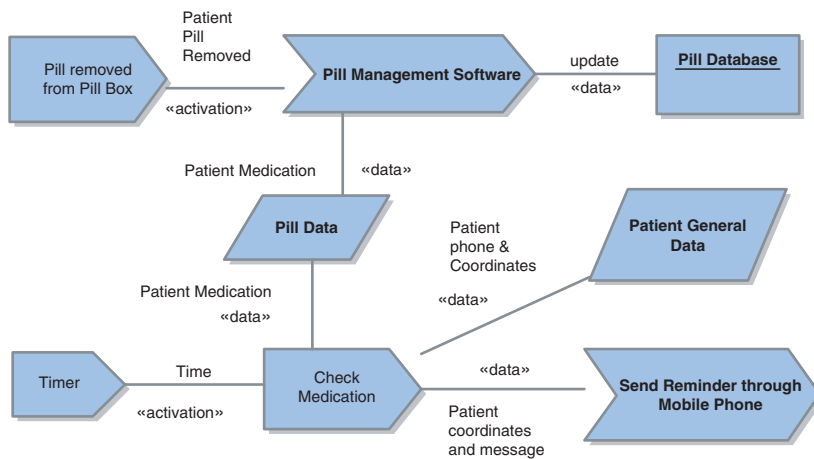


Fig. 6.3.4.1 Business Process Diagram showing the functional aspects of the Pill Management Software Component of the RSMHC project. Data and control flows are stereotyped to indicate the nature of the information and names given to flows are application dependent

information that must accompany the BP diagram to avoid entering into an early design of the system itself.

As said before, the BPD has the DFD as ancestor, so business processes can be cascaded at will. Flows are of data or control types. Stereotypes are used to typify them in the modeling domains and names given to these flows are application dependent. Business processes can be used in the design phase too. At this stage of the project development, its role is essentially to show input and output parameters of important processes in the system for communication purposes with stakeholders only.

## 6.4 System Requirements Specifications

We bypass the Cost Resource Time Estimation process that alone can require a whole book to pack project experiences of project leaders and discuss about tools used for this purpose. Risk factors can be taken into account at this step. A project estimate is not just a matter of picking the use cases to be included but counting in efforts for auxiliary tasks such as project management, testing, general tasks, documentation, deployment, contingency reserve, etc. The cost is not the same for neophyte and professional companies that have built, in the past, similar systems or solved analogous problems. The former can charge the Client for the experience, but the latter can charge more for training their own staff.

The original COCOMO (Constructive Cost Model) model was first published by Dr. Barry Boehm in 1981, and reflected the software development practices of the day. Emphasis on reusing existing software and building new systems using off-the-shelf software components was found in improved COCOMO II (Software

Cost Estimation with COCOMO II, Prentice Hall, July 2000) that is now ready to assist professional software cost estimators.

Another direction is proposed by Extreme programming (XP) group [Extreme Programming Installed by Ron Jeffries, Ann Anderson, Chet Hendrickson] teaching how to work with an on-site customer, define requirements with user stories, estimate the time and cost of each story, and perform constant integration and frequent iterations.

In very simple projects, cost estimation can be realized with spreadsheet software. A number of companies have developed their proper cost estimation applications based simply on some macros.

IEEE has recommended practice for software Requirements Specifications in 1998 inside a document of 37 pages referenced IEEE STD 830-1998 (ISBN –7381-0332-2). Hereafter, it is a proposed template (contents must be adapted to users' application domains):

#### Title Page

Project Title, Project Number, Date Signed, Date of Delivery, Authors, Version, Total Number of pages, etc.

#### Document history

This document is needed if there are many versions, many persons involved at various steps for a large project. The current document may be part of a more important document.

#### Table of contents

All chapters listed from here. Executive summary would be the first item.

#### Executive summary

Overview of the project, its impact on the market, its impact on the organization, its overall cost, etc. Brief summary of all important results, numbers, or statistics.

#### Introduction

Give the context of the project, stress its importance

#### Overview of the architecture of the product or of the solution proposed

As we are still in the Requirement phase, the architecture must be sufficiently vague so as not to become a constraint, unless this constraint is requested by the Client. The architecture of the RMSHC corresponds to the class diagram of Figure 6.3.2.1.

#### Context of the Application and User Interface

Describe in detail the context of the application and define the typical Customer of the product, the Person or the Organization targeted by the Solution. This definition allows us to define at the design phase exactly what kind of interactions the system has at its user interface.

#### Requirements for each component or unit enumerated



Component1/Unit 1

Use case and business process, or relevant UML diagrams can be used to describe the structure (avoid to formalize as a constraint) and the expected behavior of the unit.

...

Component N/Unit M

User interfaces specifications are considered as parts of Components or Units.

### Project Cost breakdown

Large projects need details about components or subunits.

### Schedule

Detail schedule and order of sub components delivery.

### Product or solution validation process

Define all test scenarios, inputs and outputs, acceptance conditions, and metrics.

### References and complementary documentation list

## 6.4.1 Requirements Engineering Model in the Model Driven Architecture

Owing to some specific constraints of the Client specification, it is not evident that the Requirements Model is always platform independent. Moreover, platform independence is not a binary concept. There could be different levels of platform independence, so what we really put in a model must be practically and explicitly defined in the model itself and annotated in the project so there must be no misunderstanding of the way developers interpret this independence concept.

Is the decision of building all databases in the RMSHC project with relational technology (written in final and contractual document as a constraint) really platform independent? It would be hard to demonstrate that the relational rules do not impact on the way schemas of entities and relationships are designed at the “logical” phase.

Owing to the “gray” interpretation of independence in PIM concept, if a back-tracking is made to retrieve the utmost goals of MDA which is reuse of design patterns, separation of the logical part of a development, separation of concerns, and possibility of evaluation of the correctness of embryonic designs, there is certainly some relationship between models and domains with a background canvas of reusing development knowledge and intellectual assets of a company. So, seeing models as a succession of *cohesive models* (a model is cohesive if, taken at any level, forms an understandable, intelligible system) could be another view of the development roadmap that can be superposed on the binary concept of the PIM and PSM. Therefore, a complex PIM may be layered and compartmented. The same subdivision may exist in the PSM as well.

From the perspective of the Developer, the Requirements Engineering Model (REM) is oriented towards the Client needs. The reuse at the Developer side means a core development that can be shared by more than one application of a unique Client. Moreover, the Requirements Model is established with the goal of evaluating cost, resource, and time, is not supported by a technical and crystal clear interpretation, so *it would be difficult to make now a perfect correspondence between the Design Model and the Requirements Model* as we cannot really transform something very technical in something that is not. This is the reason why we consider the Requirement model as separate and disconnected to the series of Design models to facilitate the design process. From a practical viewpoint, we instantiate a new project name, recreate all classes. The connection between the two models is the “realize” relationship that indicates which part of the design will realize which use case or actor in the Requirement Model. This “link part” can be considered as a small independent “Link Model” (Fig. 6.4.1.1) that maps elements of the Requirements Model to components of Design Model or Implementation Model.

This absence of correspondence does not mean that the Developer cannot practically pass the validation final tests included in the contractual commitment signed at the end of the requirements engineering phase. The REM is used as a canvas for the design phase.

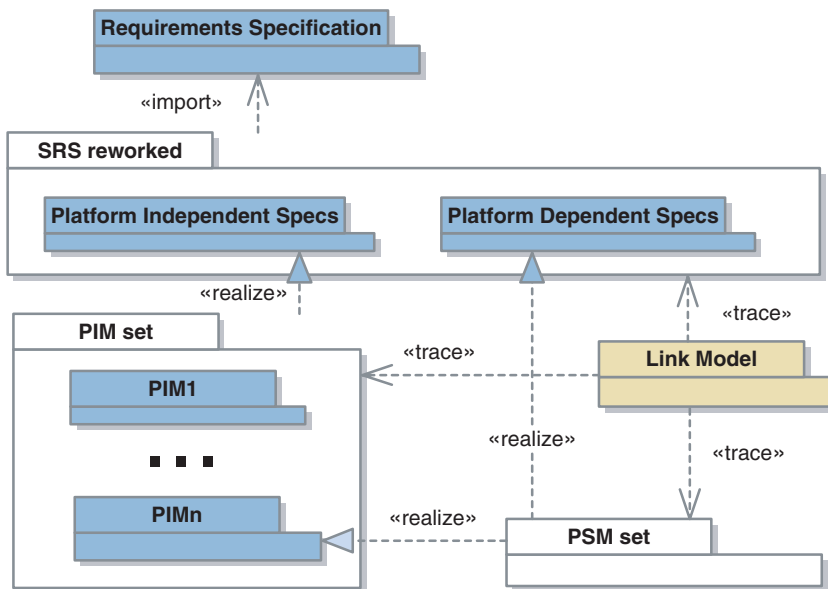


Fig. 6.4.1.1 A link Model traces all mappings between the SRS elements and components of PIM set and PSM set

The layering process using platform dependent criterion is often “horizontal” (some components are implementation dependent among other components that can be made “logical”) in the Requirements Model and *vertical* (successive model refinements) in the design phase. This is the reason why, at the beginning of the design phase, the system can be partitioned into at least two parts to separate the requirements specification into two distinct sets of constraints, a *platform independent set* and a *platform specific set*. As the partitioning process into models is not limited uniquely to the criterion based on the platform, inside the PIM or the PSM, we can define submodels or layers, even layers with compartments to isolate reusable parts of the development, thus propelling forwards the MDA objectives as far as possible to optimize reuse.

### 6.4.2 Dealing with Platform Dependent Components in PIM

At the early phases of design, components that will be imported “as is” are identified, their interface completely specified, and their exact operations schematically represented by activity diagrams or any other dynamic diagrams (state machine, sequence, etc.). The component may itself be platform independent (reuse of abstract development patterns) or platform dependent. In the latter case, there could be many attitudes:

1. *Transform a PSM into a PIM*: Create a new interface of the component for the development making abstraction of all implementation details
2. *Find PIM part of the PSM*: Use implementation independent parts of the components
3. *Come back to old habitude of text comments*: Clearly identify part of the PIM that must deal with some sort of implementation constraints
4. *Delaying*: Drop the whole development part that is “infected” by the platform-dependent component and delay its development to the next PSM phase
5. *Long-term planning*: Find all platform-dependent components of the market, analyze their specifications, extract the core logical concept, and build a virtual platform-independent component as if the Developer must develop this component himself (the best but time consuming attitude).

### 6.4.3 Various Steps in the Design Process

Figure 6.4.3.1 summarizes steps over the development roadmap.

Concerning the opportunity of automating some steps in this long process, the answers are based upon our current knowledge, research results on modeling

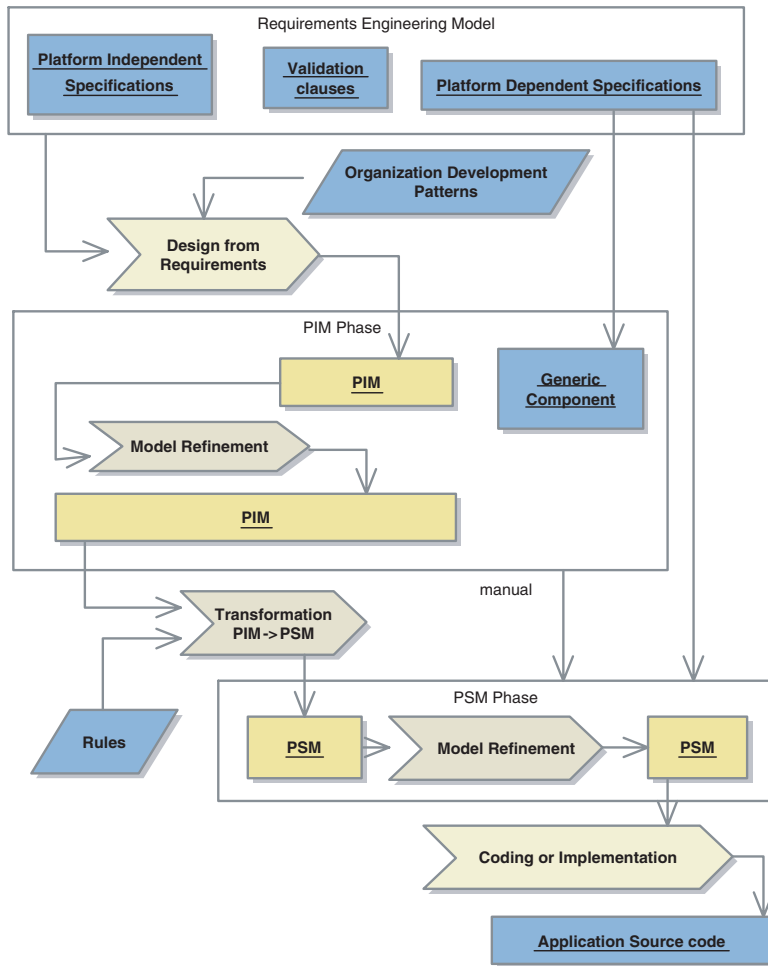


Fig. 6.4.3.1 Design process roadmap represented by a business process diagram combined with an object diagram (this diagram can be drawn with an activity diagram too). Starting from the Requirement Engineering Model, the first PIM is designed then refined. The transformation PIM → PIM is Model Refinement. The passage from PIM to PSM is called PIM → PSM transformation. Many PSM model refinements may occur. When exiting the last PSM, a coding process transforms finally the PSM into a running system

tools, development of new concepts (like MDA), and evolution of modeling standards. Obviously, what we can do today is more than in the past and the future should be better, but, we cannot automate everything, we cannot automate all along the development roadmap, and we cannot have the same automation result for all application domains. Parameters that influence this process are:

1. *Activities, goals identified in the Requirements Engineering phase are not based on highly technical diagrams.* The structure is thus sufficient to understand what is to be done in a project. So, is it really convenient to design transformations and rules for such imprecise and incomplete inputs? When designing, we transform the REM into PIM. Development patterns can be reused. It would be hard to automate completely this process as it needs human skill, trade-offs. Moreover, automating a process is worth the investment when the same or similar situations happen more than once. Domains like administrative applications, e-commerce web sites, comes back very often with similar requirements, so development patterns that can be adapted quickly to generate working designs. But, in more general cases, the best we can really do would be reusing existing components and deploying development patterns, not really automating the process with transformations and rules.
2. *PSM is commonly assimilated to code.* It is particularly true if the Developer starts coding after building his model with UML diagram (his PIM). Things are worse if the code is written directly from Requirements Model without any design (surprising fact but this practice is more common than expected). In medium size projects, there are many steps covering the platform-specific stage. The coding phase is only the last step, the early phase is the transformation from PIM to PSM. Some intermediate phases can go between, involving middleware, platform-dependent utilities, sideline developments, etc.
  - (a) If we build for instance a real-time system, in the PIM, all objects are considered as independent and their reaction to events, inputs are instantaneous. But, at the first stage of the PSM, if we decided to allow only one processor for handling all tasks in the real-time system, we are facing the problem of task scheduling that must require a dynamic study. So, scheduling is typically PSM bound and this phase is platform dependent as a technological choice has been proposed for the implementation. Only the transformation of last PSM to PSM is coding.
3. If the passage from Requirements model into PIM models is not straightforward, *the passage from PSM to application source code (last PSM) should be partially possible by defining rules, code patterns*, etc. This automation is possible with standard like the UML that imposes consistent rules for creating models and diagrams. The transformation of a PSM design into code and running application is possible nowadays with very simple, demo applications. This code automation is on the way but true applications still need some research

efforts to create dynamic diagrams that can express unambiguously algorithms. We ignore all automatic phases after the source code generation (compilation, interpretation, intermediate code, linking to running code, etc.) as they are low-level processes and are currently handled by standard development platforms offered by main computer software players.

#### 6.4.4 Hierarchical Decomposition or Bottom-up Method?

Cutting up a complex object into smaller chunks is a fundamental problem of many disciplines. At the starting point, large system must be divided into manageable subsystems. Each subsystem can be considered practically as a smaller system that can be completely specified with the same modeling concepts (goals, context, interface, inputs, outputs, etc.), exactly as the original system.

A cordless phone consists of a base unit that connects to the landline system and also communicates with remote handsets by radio frequency or higher bands (900Mz, 2.4 GHz, and 5.8 GHz). The handset and the base can be studied as two separate systems. Their functionalities are nearly identical so a common set can be identified. Special functions that allow the connection of the handset to its base will be described as input/output function from the modeling viewpoint. A small part of the project will be devoted to the integration of the handset to its base.

When all loosely coupled parts are separated and specified, each part can then now studied with apparently two opposite methodologies: *top-down* hierarchical decomposition and *bottom-up* component building. In fact, they are perfectly complementary approaches. We cannot really start with bottom-up as long as all the required functionalities are not still identified correctly by a top-down approach. If we start by building components (bottom-up) without having previous knowledge of what problems to be solved, possibly the results could be very interesting but probably, we have to find applications or customers who want to integrate our innovations.

Bottom-up building process makes use of either elementary objects or off-the-shell components available in the market and integrates them in systems. To master the whole behavior of the final system, structural, functional, and dynamical characteristics of each component/object must be known in its utmost details. Documents, diagrams, and all artifacts are normally available either in paper or electronic form. A development database containing all their descriptions and characteristics is a must for serious designers. The structure and the behavior of a mechanical assembly, an electrical circuit, a software component, or a whole organization containing many objects/components/agents can be studied globally only if we have access to individual properties and behavior of each constituting element.

If the previous required RMSHC system needs a BPM (Blood Pressure Monitor) and if this Developer decides to acquire this component as a COTS (Component Off The Shelf) obtained from OEMs (Original Equipment Manufacturers) to include it into his design, detailed technical specifications and undisclosed information are generally available from 3rd party developers or OEM.

A processor included in a system as an intelligent device or controller is an electronic component. Even if the internal structure is not disclosed, all the available specifications (hardware, software, limit conditions, environmental parameters, instruction set, timing of all signals, etc.) give to the integrator all the necessary information to predict the behavior of a system having this processor as component.

The behavior of an electrical circuit (for instance RLC or active filters, etc) is known through differential equations deduced from either passive or active component characteristics.

Kinematics and dynamics of a robot tool tip can be computed if we know all dimensions, masses of all articulations and tools.

We can evaluate the success of a university course and find out what to enhance only if all parameters are known (provenance of students, individual academic results, quality of the course, quality of the professor, quality of supporting resources, biases caused by circumstantial group dynamics, etc.).

To summarize, the behavior of the whole system is made from the behavioral contribution of all its constituting elements or components. When designing a system, we start identifying all functional requirements by a top-down approach, find elements/components, or build assemblies to satisfy those requirements by a bottom-up approach, then specify the structure and study the dynamic behavior of the whole knowing in principle all about parts. Classical systems engineering and industrial manufacturing process breaks down large, complex systems into component parts then rebuild them so that the characteristics on the whole can be controlled efficiently. The automobile industry is probably the most visible systems engineering success following this design principle. At each stage of rebuilding the whole from parts, full testing and dynamic studies must be conducted thoroughly. If the quality of each step is not fully controlled, initial deterministic systems may exhibit chaotic behavior, but this is another problem.

## 6.5 Designing Real-Time Applications with MDA

The special characteristic that makes the software industry stand apart from other domains is its aptitude to adapt to fast technological changes. The MDA is not a new concept (it has been applied already to define ISO-OSI standard of network communication) but has recently been an incentive for generalizing the software development process, or more generally the design process of all systems, software or multidisciplinary projects. Roughly speaking, the technique stresses the importance of separating the PIM from other PSMs to save the maximum knowledge and experience assets of a company that must

daily deal with very fast platform changes and that which wants to effectively deploy reuse. Moreover, the MDA methodology offers to create a formal architecture of high level for reasoning about system and for communication purpose.

In this view of the development roadmap, the previous blue prints of the RMSHC studied with class, use case, and business process diagrams are parts of a new development component named REM.

A PIM provides formal specifications of the structure and behavior (functions and dynamics) of any element of a given system but abstracts away any technical or implementation details. Implementations in many platforms can therefore be realized from one PIM. The essence of PIM is reuse of known patterns of design, validation of the logical architecture early in the process, delay of problems like scheduling, system integration, or interoperability to PSM. Several PIM may exist in a complex project. The passage from one PIM to another is called *model refinement* or *model transformation*.

The PSM can be viewed (in this proposal) as a PIM plus incorporation of platform-dependent elements (agents/components/objects). They are still not a code. Those platform-dependent elements will be studied with the same diligence as for PIM, except that the whole project is now targeted to be implemented in a defined environment.

In PSM, emphasis is put on technical aspects of project design that is influenced somewhat by project management aspects. UML diagrams can be deployed along this long design phase. UC diagram can still be used for goal, responsibility, capacity, activity, and resource specifications if necessary. At the starting point of the design phase, package diagrams are used to structure tasks necessary to make a Project Assignment.

A project is like a human or social system (temporary organization) (Gareis, available at: <http://www.p-m-a.at>) that afford resources, manage them in time and possesses algorithms to deal with unexpected problems or to resolve conflicts that may appear during the development process. When executing the design phase, there are two projects that are running side by side, the *Client project* and the *Developer project*. The Developer solves the problem of the Client, builds the Client system while managing its proper organization to afford necessary resources at appropriate time to help Client project to get closer and closer to the Client goals and expectations. So, project management (this “art of getting things done by others” is absolutely necessary for midsize and large projects) is the *Developer system* and the system to be built is *Client system*. Even if the role of the Developer is constantly managing development, he must adapt or match the structure of his organization to the nature or the domain of the Client system as all client systems are different.



### 6.5.1 Project Breakdown Structure into Packages: Re-Spec Phase

There are many ways to start designing a system. The choice depends upon system complexity, problem domain, and quality of the Requirements Engineering phase, experience, and skill of the design staff relative to the product to be designed. The general guidelines are:

1. Well study the SRS (System Requirements Specifications)
2. *Operate first a horizontal repartition.* If the system contains obvious independent subsystems, break the project into several smaller independent subprojects. Structural diagrams representing a coarse structure of the system to be built, being established at the Requirements Engineering phase, may serve as a starting point. Identify all subprojects by their package names with a package diagram. A series of packages (smaller independent sub projects) can be created at the end of this process.
3. *Operate a vertical decomposition.* When entering each subpackage created at the previous phase, operate a vertical decomposition to find subpackages. It would be important to make a clear difference between a horizontal repartition and a vertical decomposition. A vertical decomposition supposes that the package at the top contain all subdecomposed packages and the suppression of a subcomponent impairs the existence of the top project. In a horizontal decomposition, if a subhorizontal project is interrupted; all remaining projects can be continued normally. Sometimes, due to the tight coupling of horizontal subprojects, it would be difficult to make such a distinction. But, developers must always make such an inventory. Think of *collaboration* for a horizontal repartition and *aggregation/composition* for vertical decomposition.
4. *Repeat horizontal and vertical decomposition again* for complex system to isolate more elementary subpackages. This two-step (horizontal then vertical) decomposition process is conducted until a high degree of coupling between objects/agents/components is detected. Leaf packages reached at this time are *monolithic projects* and contain small subsystems that can be now studied with conventional object methodology.
  - (a) It is worth noticing that elements inside packages may interact together via loosely coupled interactions. For instance, in a Car, the *braking system*, the *steering system*, and the *powering system* interact together in a driving activity, but they are

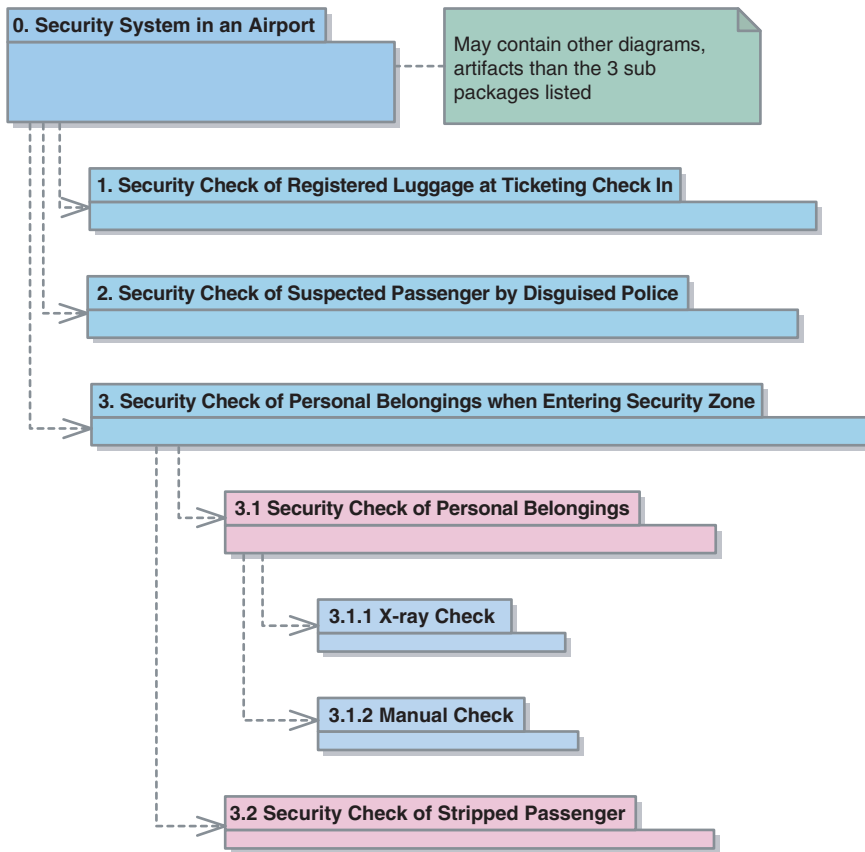


Fig. 6.5.1.1 Decomposition of a Security System in an Airport. The topmost imports three packages 1, 2, and 3. Package 3 is decomposed at second level to 3.1 and 3.2, and 3.1 is decomposed at third level to 3.1.1 and 3.1.2

nevertheless considered as subsystems (packages) that can be studied independently.

- (b) The problem of detecting the crucial moment when we must terminate high-level package decomposition is a matter of developer experiences. Leaf package can contain packages for organizing the design process. Leaf package is a kind of development unit (Fig. 6.5.1.1).
5. *Distribute original specifications among packages and respecify if necessary.* Frequently, specifications must be reworked when the original system has been broken down into packages. Each package can be considered as a *work unit* for an individual or a group of developers for a while. A design respecification may be needed to have a better view of

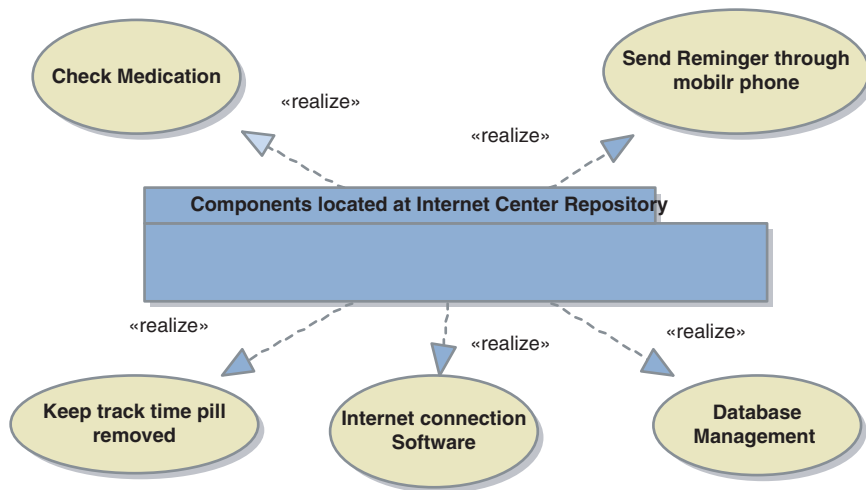


Fig. 6.5.1.2 Re-specs with use cases and packages illustrated via the RMSHC example

the initial system, the responsibilities of all subpackages, and a clarification of objectives and contents of the work unit. This respecification facilitates the design and the communication of participating development staffs or individuals. Metrics and evaluation criteria can be defined if necessary. This re-specs phase reuse SRS with additional use cases with *realize* relationship stressing the responsibilities of the current package (see Fig. 6.5.1.2) and its relationships with other packages. The repartition of packages can let people think erroneously that they are independent, but generally, we can have a tight coupling between packages.

- (a) In Figure 6.5.1.1, package 3.1 *Security Check of Personal Belongings* must contain some diagrams that link packages 3.1.1 *X-ray Check* and 3.1.2 *Manual Check* together. For instance, a diagram may be used and stored in 3.1 node to show that *Manual Check* is used only when the agent making *X-ray Check* is not confident with what he has observed on the screen. At the beginning of the project break down, the exact connection cannot be known until the two component packages are thoroughly studied. Dynamic diagrams that link packages together are generally built when leaf packages are completely designed and junction points determined.
- (b) This Project Breakdown and Re-Specification phase (Design Specifications by opposition to Requirements Specification) is important as it divides the project (if the nature of the system allows such a division) into several manageable subphases. Re-specs may

formulate intermediate deliverables and sometimes unsuspected and interesting components emerge from this process.

## 6.5.2 Project Scheduling in the Re-Spec Phase

Before entering project scheduling, it would be necessary to first study if there is any functional dependency between project packages. Functional dependency imposes often *time dependency* or *temporal coupling* and an order of package development in the Developer system. It is worth noticing that this time dependency must not be confused with the time dependency in the Client system.

For instance, the package 3.1.2 *Manual Check* is fired AFTER 3.1.1 *X-ray Check*, so there is a time order or a *time dependency* between those two packages. Time dependency occurs when a package needs results or data from other packages. But, this dependency regulates the execution order in the Client system. From the Developer view, the *Manual Check* process is completely independent from the *X-ray Check*, so those packages can be studied in any order.

Project dependency occurs, for instance, in the construction domain. We cannot build anything when the foundations are not in place. Painters would be the last equipment to be hired.

Project scheduling is simple if the Developer side has only a single project. A high number of companies have an organizational structure that can accept multiple projects to run simultaneously. So, scheduling is a complex optimization exercise of matching this organizational structure (with existing projects running) with the schedule of a new Client project. Multiproject management (Speranza and Vercelis, 1993; Silver et al., 1998; Jin and Levitt, 1996) must deal with complexity and uncertainty. The breakdown of the Client project into more manageable parts plays in this context an essential role for workflow optimization. For a survey of this project scheduling please consult Herroelen et al. (1998).

## 6.5.3 Starting a Real-Time System with the UML Sequence Diagram

There are many possibilities to start designing a system. The choice of the starting diagram depends upon the domain, the complexity, the experience of the developer, and his biased inclination towards using a particular UML diagram. When developing database schemas, developers start almost with class diagrams. For a real-time system, if the developer knows the application domain, he can start with a class diagram directly, but, if he is a neophyte in a new domain or has never developed any similar project, it would be more appropriate to start with a sequence diagram, an interaction overview diagram (an activity diagram that packs fragments of sequences



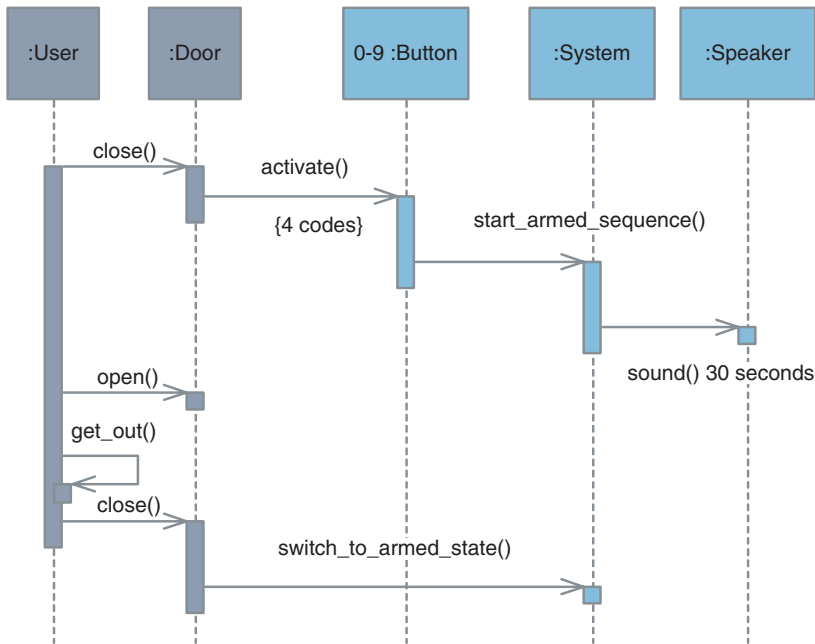


Fig. 6.5.3.2 User's Instruction "How to arm a home alarm system" expressed by a sequence diagram

A *scenario* or a *sequence* is a particular path in a graph (or a sub-graph) extracted at any two levels, not necessarily at the root of the graph. Each application defines some typical scenarios and explains them as *Users' instructions*.

When using a mobile phone, the instructions we must learn are "how to power on, how to power off, how to change batteries, how to charge, how to make a call, how to answer a call."

Therefore, when starting a real-time system study, the same Users' instructions sequences can be used to identify objects (with their corresponding classes) and messages (object operations). User's instructions stop all interactions at the system interface and consider the system as a monolithic object. Design sequences penetrate inside the boundary of the system and try to find out all internal objects. Figure 6.5.3.2 is a sequence diagram that explain to a User (User's instruction sequence) how to arm the Alarm. Figure 6.5.3.4 is the same sequence at the interface, but it goes inside the system and identifies internal objects (Developer sequence). The two sequences are identical if we stop at the interface of the system, but the Developer sequence explores the system internals to find out all constituting objects. By adding sequence over sequence, all objects are identified and a class diagram can be built at

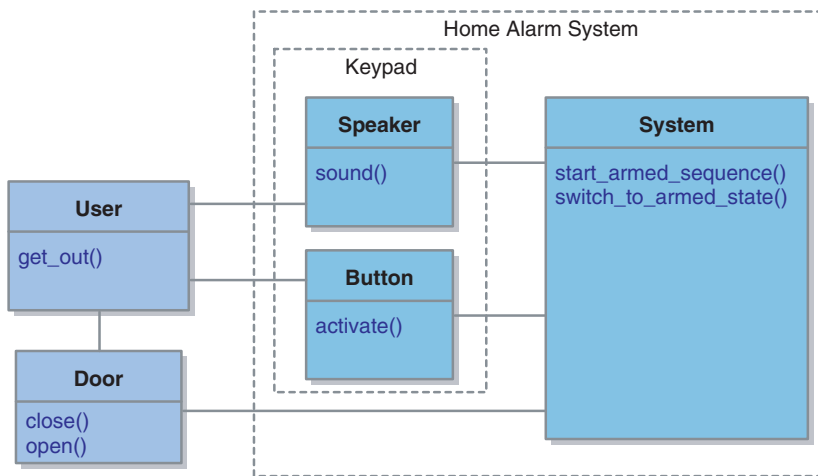


Fig. 6.5.3.3 Corresponding classes identified while drawing the sequence diagram

the same time to capture all operations (messages in the sequence diagram) (Fig. 6.5.3.3).

The number of sequences in a complex system is very large and sequence diagrams are time consuming to draw. They are good for capturing the attention of an audience, but not very handy for exploring the algorithm of a complex system. So, this diagram is handy for presentation, for starting a system investigation of its first elements and interactions, but once main components of the system are identified, other diagrams (state machine or activity diagrams) are more appropriate for finishing the algorithm. In other words, sequence diagrams are powerful for expressing the chronology of interactions between several objects in an expressive form but are not really targeted to find out systematically all interactions of a system.

The sequence diagram of Figure 6.5.3.2 is focused on the User's interaction at the interface level. The Alarm is seen as a monolithic system with high-level operations. Only the keypad is visible to the User and the sound is perceived through the Speaker integrated in the keypad assembly. *0-9:Button* is a shorthand notation that packs 10 buttons 0-9 of the class Button. The User acts on the Door and the Door, in turn, acts on the System (The design phase will reveal the existence of an intermediate object that is the microswitch in the door frame). Other sequences will be necessary to explain other functions (disarming, detecting intrusions, changing code, etc.).

The sequence diagram of Figure 6.5.3.4 starts as the sequence of Figure 6.5.3.2 but crosses over the frontier of the interface to explore internal components. This fact shows the difference between a sequence established at the Requirement specification and a sequence of the Design phase.

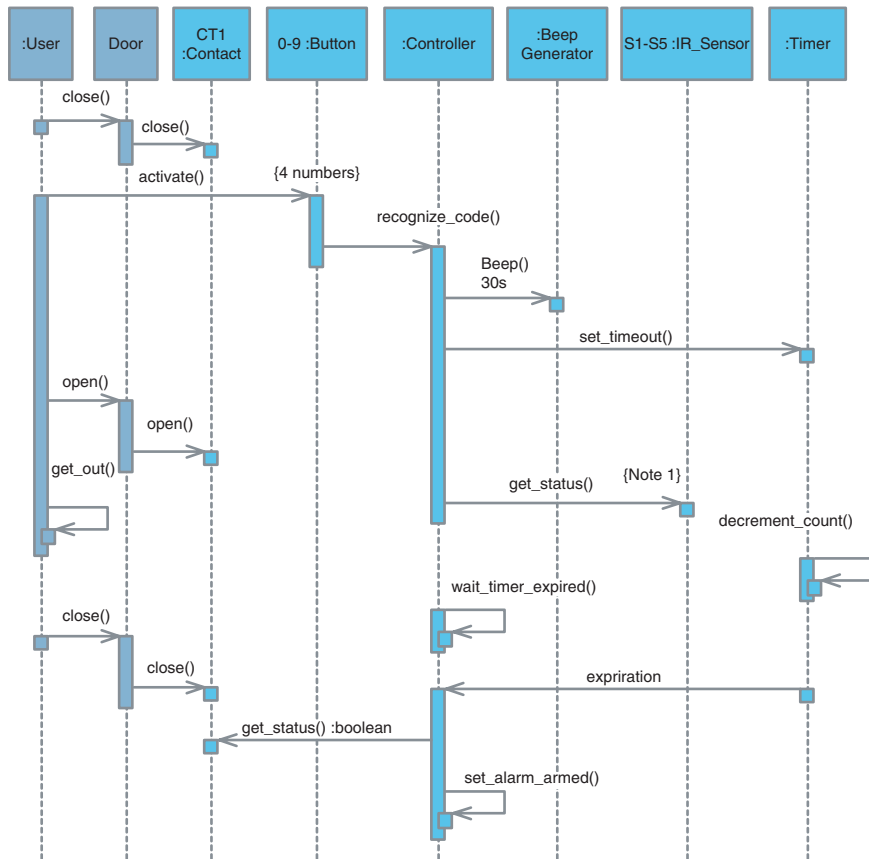


Fig. 6.5.3.4 Sequence diagram for developing the Home Alarm System. The frontier of the System is crossed over to explore internal components. Successive decompositions are necessary to identify more elementary objects. (Note1: If Infrared Sensor is activated, the alarm cannot be armed since there is somebody inside. But this sequence is not represented)

## 6.5.4 What Does a Logical Model in Real-Time Applications Mean?

Sequences used to depict objects in design phase are nearly identical to user functions but they cross the system frontier to find out all objects necessary to design the system. Real-time models deal directly with buttons, switches, sensors, controllers, etc. so apparently, it is somewhat questionable what a real-time logical model really means? To apply the MDA concept, the logical design must be considered as a *high level and logical model*. To reach these objectives, it would be necessary to understand specific characteristics of a logical model and to observe the following recommendations when elaborating a logical real-time model:



1. *All objects are made independent at the logical phase* to isolate the roles performed by each object in the system. Timing constraints specified at the logical phase are those needed for the operation of the object itself, leaving aside all other constraints proper to a specific implementation.
  - (a) The Timer operates independently from the Controller in this logical model, even if, at the implementation phase, the Controller and the Timer are mapped into a unique microcontroller. The Beep Generator can be implemented by a subroutine of the microcontroller at the implementation phase as well, but at this logical phase, its functionalities are separated to evidence its role.
  - (b) Microcontrollers (microprocessors equipped with parallel, serial interfaces, timers and a small amount of RAM, programmable ROM memory) are used at the implementation phase as intelligent devices. To minimize system cost, generally, they are used as all purpose devices. In this respect, a scheduling problem is superposed to the proper timing of the system defined initially. For instance, if we make use of the integrated timer of the microcontroller both for the Timer function, the Beep Generator, and real-time interaction with users at the keypad interface. As all these functions are activated simultaneously, there is an interference of the Beep Generator timing with the timing of the Timer, and with the keypad busy wait loop. At this logical phase of development, this fact must be ignored and all functional devices are considered as independent. The scheduling problem is a design problem at the implementation phase but not at the logical phase.
2. *Even if a real-time system makes use of real world components, their functionality is made generic at this logical phase of design.* Primary functions of devices like controller, sensor, switch, button, display, motor, actuator, sensors, etc. are well known, so a library of classes containing such logical components are very useful and is recommended to build a library of generic components. If more specific devices are needed, they can be derived from generic devices.
  - (a) An actuator is a special mechanism designed to act upon an environment. A motor associated with a harmonic reducer (high ratio hollow shaft gear reducer) is an actuator used to move an arm of a robot relative to another mechanical part. Actuators can be of hydraulic or pneumatic types. So their classes must be defined individually to take into account their specific features. Actuators and sensors are complementary devices in real-time applications. Computers,

microprocessors, or microcontrollers are intelligent controllers in this context.

3. *At logical level, the model often follows the natural order of events and its chronology. The way real systems implement the capture of events is not of concern at the logical phase, so we do not have to reproduce exactly the implementation arrangement.* In a real-time system, we have to manage passive components like a switch, a button, and a passive sensor with an intelligent element as a Controller. There are two methods for repeatedly checking whether an event occurs: a busy wait loop or an interrupt. In a busy wait, a loop is created to scan all passive devices. From a modeling viewpoint, the busy wait loop reads the status of the passive component (switch or button contact), hence the arrow is oriented from the active component to the passive component requesting its status. It is an acceptable solution if the Controller is idle most of the time. As an alternative, an interrupt signal can be sent to a Controller requiring it to process an *interrupt handler*. A real interrupt layout needs hardware to generate electronic signals. With the interrupt sending a signal to the Controller, the arrow is reversed and oriented from the passive towards the active components. Figure 6.5.4.1 shows how to represent these two different cases.
4. *When modeling components that are highly implementation dependent, we can consider them as black box components. All internal operations*

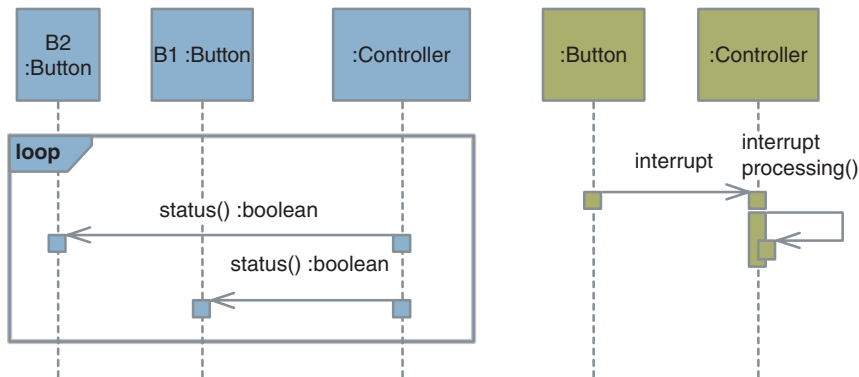


Fig. 6.5.4.1 The status of a passive component in a real-time system can be detected by reading the status of the component with a busy wait loop or mounting appropriate hardware to send an interrupting signal to the Controller (arrow direction must be reversed). When making a logical model, this decision can be postponed to the implementation phase, so the two representations are considered as equivalent at the logical phase

*are brought to the interface and self messages abstracting actual internal structures can be used to postpone their development to the implementation phase.* Self messages are those that are redirected to the same object (User getting out of the house, Timer decrementing its count, controller setting its alarm armed in Figure 6.5.3.3). In the execution of a self message, it appears as if the operation is self sufficient and does not need any collaboration of other objects to perform the self message. But, more frequently, the presence of several self messages is symptomatic of a system not insufficiently broken down or not decomposed at all. At the limit, an active system without any interaction with its environment has only self messages. Self messages can be very useful in the context of logical model development as they can be used to postpone implementation-dependent components to next steps.

5. *A logical model must be compatible with implementation that can be derived later from this first design model.* The logical model is the first piece of reuse. The realization of previous points will contribute and reinforce this goal, guarantee that no knowledge is lost, and proven practices can be reused in a regular and systematic manner. Design quality and productivity are enhanced accordingly. It is highly desirable to have a smooth transition between PIM and PSM. The term “smooth” means that classes are imported, derived, and reworked. The transition between SRS calls upon a “mapping” process. A mapping is generally understood as a nonsmooth process.

### **6.5.5 Starting Real-time Systems with the UML Interaction Overview, Activity, or State Machine Diagrams**

It is not mandatory to start a system with a sequence diagram as shown in the preceding section; the UML has in its arsenal several diagrams that support other starting methods. Despite its high expressiveness, a sequence diagram relates a story in a precise context and, with the way it was designed, it would be difficult to display more than one scenario. Even a long sequence requires some work to organize it into smaller chunks of sequences.

The design of a whole system needs full algorithms with many scenarios fitted together in a complex network ruled by run-time conditions, the last UML 2 standard proposed the interaction overview diagram as a complementary formalism to be used jointly with the sequence diagrams for building a more complex dynamic structure involving many scenarios. Theoretically, by linking all interaction overview diagrams together, we must be able to build the whole algorithm. But, it is widely recognized that a sequence diagram is interesting

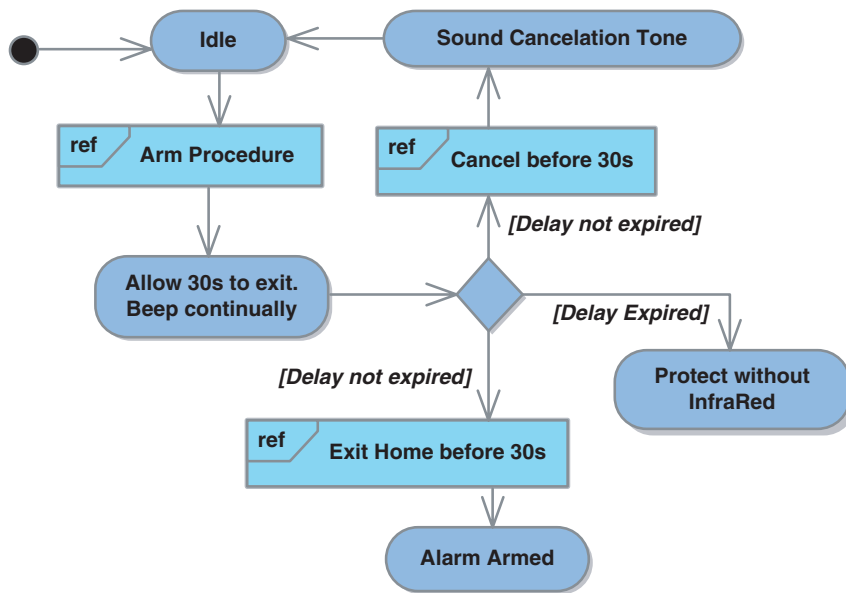


Fig. 6.5.5.1 Interaction Overview Diagram packing many references to sequence diagrams inside an activity diagram. Arrows are control flows and activities are mixed with “ref” fragments that must be detailed elsewhere

to start investigating a system, but we rarely see anyone making use of this formalism to develop a full dynamic study of a medium-size system. Probably, the sequence diagram is very time consuming to establish and needs space to align all objects horizontally. However, it is intuitive and can easily catch the attention of nontechnical persons.

The interaction overview diagram (Fig. 6.5.5.1) is the third diagram of the Interaction Diagram Suite and is a variant of the Activity Diagram (Fig. 6.5.5.2). The focus is put on the flow of control where nodes are Interaction (Unit of observable behavior) or InteractionUse (a reference to Interaction). A sequence is a piece of behavior and simultaneously an interaction so nodes can be mapped into sequences through *ref* fragments.

Figure 6.5.5.3 shows the use of the State machine diagram as a starting method for studying a system. Class and object diagrams can be built simultaneously to support the structural view. To summarize, there are many ways to start designing a system when studying real-time systems:

1. *Use sequence diagrams only* (appropriate only for very small system).
2. *Use sequence diagrams associated with interaction overview diagrams.* Interaction Overview diagrams layout high-level algorithms and sequence diagrams fill nodes referenced with “ref” fragments. Control

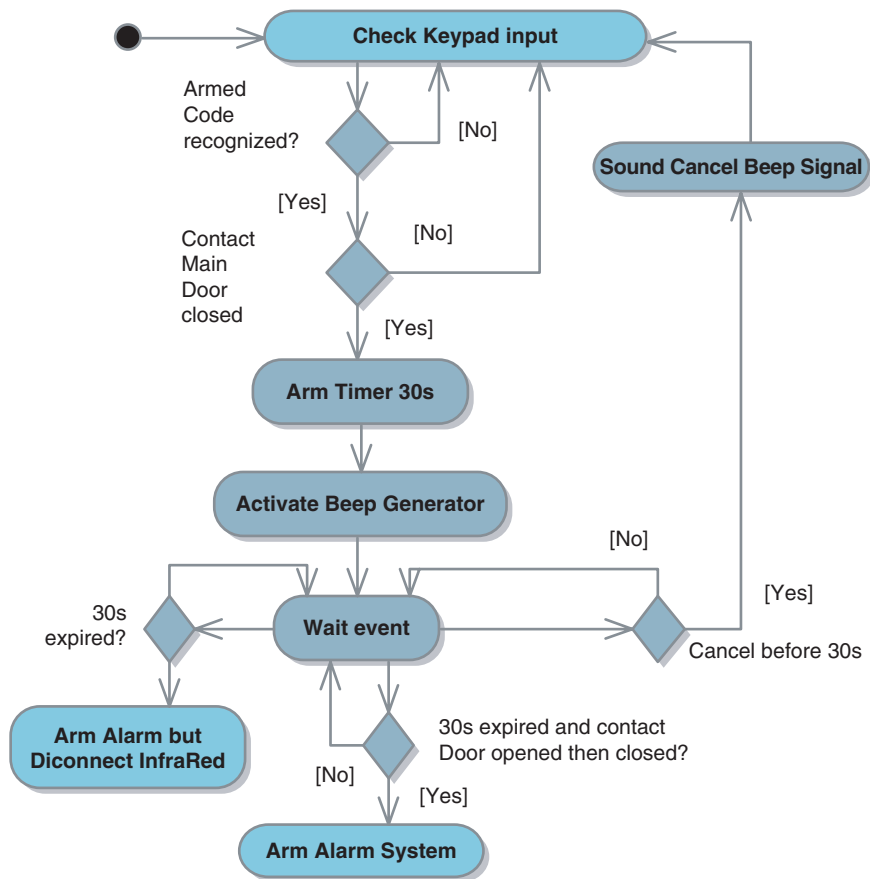


Fig. 6.5.5.2 Activity diagram as a starting method for the design. A class diagram can be built simultaneously to store operations (activities). Each test is also an activity. This diagram is only a chunk of an algorithm and is targeted to show a possible way of starting a design

structures like “alt,” “loop” fragments, etc. can be used as well to enhance expressiveness.

3. *Use activity diagrams associated to structural diagrams* (class diagrams to register operations/attributes and object diagrams to list all objects that collaborate inside execution scenarios).
4. *Use state machine diagrams* replacing activity diagrams in the previous method. Class and object diagrams continue to support structural views. Electrical engineers are very fond of state machine diagrams as they are used in the past for studying sequential logic circuits.
5. *Use structural diagrams only*, relaying dynamic studies to the implementation model. This method is not really appropriate for real-time systems

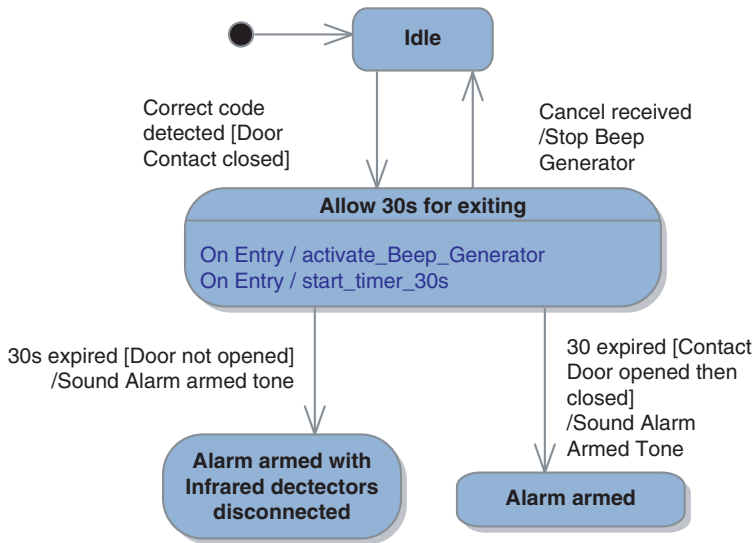


Fig. 6.5.5.3 State machine diagram as a starting method for designing

as a logical model also needs a dynamic study. This method is an extension of the habitude developed with database design and E-R concept. As database applications are an important issue, this point will be discussed separately later.

## 6.5.6 Building Real-Time Classes and Object Diagrams

Historically, real-time systems are built using low level languages (often assemblers) and are targeted for dedicated hardware and specialized and light weight (operating system) kernels. Efficiency, performance, and other criteria proper to embedded or mobile systems are of primary concern. Software modularity and reuse are secondary factors. Recently, with the complexity and the integration of real-time systems into a more connected world, things have moved towards the design of modular and reusable real-time components. Finding the right abstraction to build extendible and reusable components requires experienced object-oriented designers. From a software viewpoint, reuse was a reality.

Standard object libraries (C++, C#, or Java) are components, interfaces, algorithms, containers, and data structures that can be used and reused in building object software. To get the current date of the day, we have just to instantiate an object from the class Date belonging to the standard library. Storage structure like B-tree does not need any programming effort.

A common piece of software is merely an art of associating interacting objects mostly derived from standard libraries. If necessary, new objects can be

instantiated from existing classes of standard libraries with additional algorithms and operations. If they are catalogued and systematically reused, they constitute effectively company software assets. Inheritance explores similarities and enforces the whole process.

Embedded and multidisciplinary systems throw a more stringent challenge to the industry of embedded components. Hardware and software cannot be dissociated and classes are not limited only to software classes. The description of multidisciplinary component classes is therefore necessary.

A Vending\_Machine class can be designed and specialized later to sell drinking bottles, chocolates, candies, or coffee. Instances of these classes all need coins, product selection function, commands sent to actuators to perform appropriate actions according to targeted sub domains. The challenge would be how to develop the most universal component that can be reused in many contexts, in many domains, roughly over space and time continuums. Only actuators must be dissociated as they must be adapted to the final product sold. Commonalities grouping or gathering are the master concern to construct new classes that describe multidisciplinary components.

Robot articulations and visions systems are nowadays COTS (Components Off The Shelf). They can be purchased separately in the OEM (Original Equipment Manufacturer) market. The application is not defined in the components themselves. It is up to the final integrator to decide upon the application purposes.

Generic components are not limited only to things but may contain humans, agents and organization. For instance, in military applications, tactical intervention units can be seen as well defined multi-agent components with a given composition, defined operations and responsibilities. With internal knowledge and the nature of the missions, they can adapt operational processes to achieve some goals defined only at runtime. The same concept may be seen in Public Security Domain (Fire Intervention Unit, Crisis Intervention Unit), Medical Domain (Reanimation Unit, First Care Unit, etc.) or in Financial Domains (Tax Recovery Patrol).

Generic task models (Brazier et al., 1998) will help considerably deploying system reuse. Diagnosis, analysis, design, and process control, scheduling, dynamic scheduling, etc. are, for instance, current tasks in everyday life. Doubtlessly, experienced designers would be more predisposed to extract common behaviors in a specific domain than newcomers. New domains cannot be mastered immediately. So, the deployment of reuse concepts is the concern of experienced developers, but at the starting point, there must be awareness, conscience, and organization determination and resource devoted for exploring this avenue. We mean resource since, at some period of the reuse process, the organization has the impression that it does not solve customer problems directly but something with very long-term benefits. But, this way of doing things is *highly strategic* and will make the difference between two performers in a domain. The MDA supports this generic step by inserting a first generic design as a starting model or by importing/merging elements of a Common Generic Model (CGM) available for all applications. Therefore, genericity will condition the way we design our classes in the logical model (Fig. 6.5.6.1).

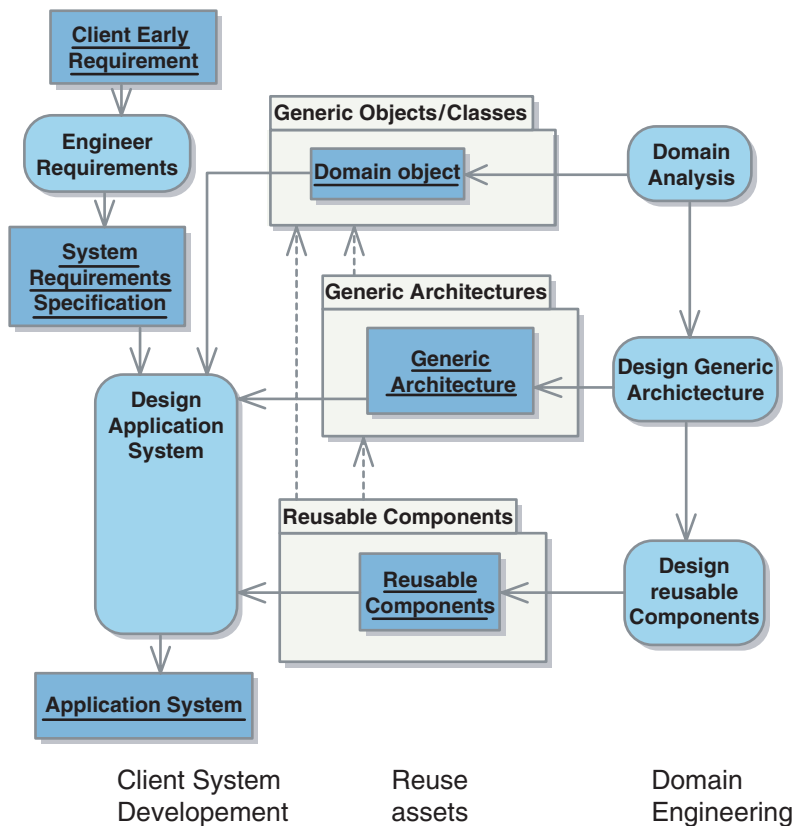


Fig. 6.5.6.1 Domain engineering with two activity flows (Domain Engineering and Client System Development) that may be conducted in parallel or not. Relationships between packages are import/merge. Objects and Components are underlined. Flows connected an activity to an object are object flows. Flows between two activities are control flows. Designing an application consists of choosing domain objects, domain generic architecture, domain components, or COTS, taking into account SRS, then generating final application system

It is worth noticing that Standard Template Library (STL) represents already a breakthrough in genericity in C++ programming language. Comprising a set of data structures and algorithms, STL provides components available to many apparently different applications. Adopted by the ANSI/ISO C++, STL is an important tool for every C++ programmer.

Problems to be solved in a domain evolve daily with circumstances, with new actors so the dynamics of systems appear each day under a new appearance but the number of objects in a domain is nearly static. The key factor of reuse is thus how to build all *objects/classes of the domain while solving the problem*. Subsequent problems will add new actors, and new operations to the existing system.



This methodology of development will naturally manage reuse by creating two applications:

1. *Domain Engineering of reusable components and architectures.* Objects in this domain have general features (attributes and operations). They are integrated inside general architecture
2. *Client Application.* The client application imports, merges objects of the domain and mixes them with specific objects of the application.

For the first systems, Developer can start a Client application then select what he can keep in the Domain for later use. It is not necessary to have a domain fully stuffed before starting a Client project. A rich domain attests, however, the experience of the Developer in the domain.

In the case of an Alarm System (Fig. 6.5.6.2), a full library contains all current sensors and devices with defined features like Alarm Controller, Backup Battery, Keypad, Siren, Motion Detector (for human), Special Motion Detector (for human having pets), Door/Window Contact, Central Monitoring Station Model (with list of operations), Control Panel Container, Smoke Detector, Glass Break Detector, Panic Button, Pressure Mat (for under rugs), Closed Circuit TV, Alarm Screen, LCD Display, etc.

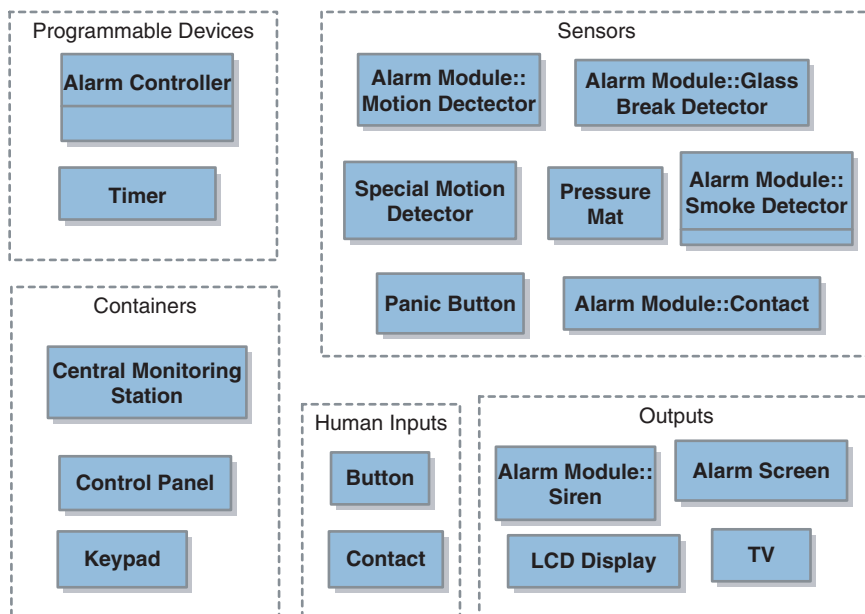


Fig. 6.5.6.2 Classes of Alarm System Domain (Generic Classes package). Some classes may appear more than one time according to the classification used. Classes appearing in this package are nearly not connected together, except some evident inheritance. Panic Button and Button can derive from an abstract Generic Button

Real-time systems are naturally multidisciplinary systems (an *electronic* Controller containing a *software* program, an *electrical* Siren, a *mechanical* Panic Button, a *mechanical container* as a Control Panel, etc. are all objects instantiated from their corresponding classes). They deal directly with real components so they can appear to some, as an implementation model and therefore lose them for some genericity. In this domain, practitioners are not disposed to go back more in genericity for not losing contact with reality. The application is till considered as logical if there is no specific implementation constraint or if the type of constraint imposed at Requirements does not contradict or destroy genericity. At this logical phase, each element of the system is considered as working independently and there is no scheduling to deal with timing constraint.

Figure 6.5.6.3 illustrates what is to be stored in the Generic Architecture package of an Alarm System. It mimics the concept of Library found in the software programming language. The number of entries is unlimited and depends upon the experience of the Developer. It is not necessary to create all generic architectures in the domain to be able to start applications. This library will grow naturally with time and the diversity of applications developed for clients.

The two important relationships used in this package are aggregation/composition and inheritance/derivation illustrated with the two examples. In Figure 6.5.6.3, *Control Panel Type 2* class derives from *Control Panel* class but *Control Panel Type 3* class contains *Control Panel Type 2* class (no inheritance). This subtlety can be answered from the engineering viewpoint. The single inheritance relationship can be used only if the nature of the parent and the child class is nearly the same or very close to each other. So, for instance, a *Panel* cannot be derived from a *Keypad* but contain it as they are two objects of very different nature. In our case, *Panel*, *Panel Type 2*, and *Panel Type 3* are all panels so effectively that they are of the same nature but it is not mandatory that they must be tied by inheritance (the conditions is necessary only). We must search the reason outside the “nature” scope. In the first case, the microarchitecture says that *Control Panel Type 2* derives from *Control Panel* as it makes use of the same components developed with zero or very minor modifications (involving insignificant industrial process). The holes are already there to accept the insertion of three LED; so components are added without any serious redesign. For the *Control Panel Type 3*, there is a replacement of the plastic Frame of type 0 by another *Plastic Frame of Type 3* that accepts more buttons. The final size is not the same, all the packing process will be changed accordingly, but internal electronic board and all other components of type 2 are used in type 3. It is not mandatory to adopt such a modeling solution. Our purpose is to show that Developer can adopt proprietary rules to model the architecture of his system as long as he has logical reasons to justify the solution retained and diagram elements can be easily understood by the whole group.

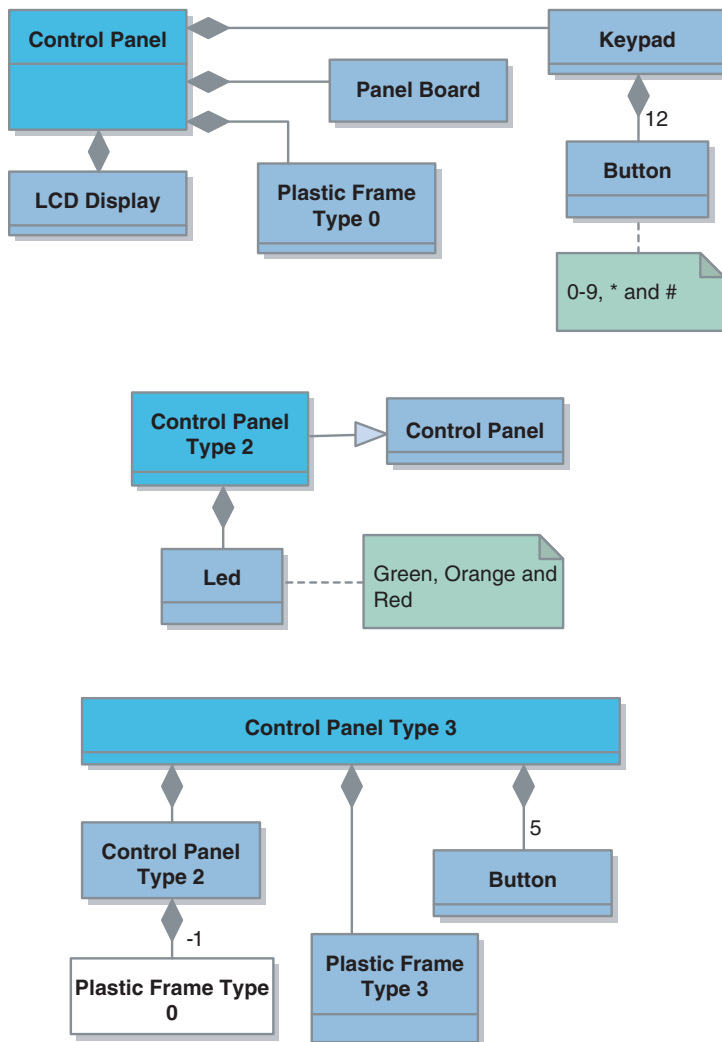


Fig. 6.5.6.3 Contents of Generic Architectures package (Alarm System). A Keypad may be defined originally as containing 10 Button objects. Structural relationships structure classes, but generic architectures remain generic in this package and are not targeted to any particular applications (or at least, they are targeted to support a whole family of applications). Two relationships (aggregation/composition and inheritance) are mainly used to build microarchitectures (see text for comments). In the third case, Plastic Frame Type 0 with multiplicity of -1 must be removed from the Control Panel Type 2 (proprietary notation)

Please note that our use of a multiplicity -1 to exclude some elements of an assembly is not recognized (for the moment) in the current UML standard. Properties modified or generated with the presence of the subtracted element must be suppressed. We can avoid the presence of the aggregation/composition with multiplicity -1 by suppressing Plastic Frame Type 0 in the Control Panel in Generic

Architecture Package and by transferring this relationship in the Application Package. That is the method adopted for the Panel Firmware but, voluntarily, to enrich the methodology, we want to show two possibilities (one solution with the Plastic Frame and another with the Panel Firmware) to illustrate two modeling variations.

A third level *Reusable Components package* (Fig. 6.5.6.4), as pointed by its name, contains microarchitectures, not ready for end user applications, but sufficient to build internal and interesting components for the Developer side. *Generic Classes package*, *Generic Architectures package*, and *Reusable Components package* store knowledge and know-how of a company, fully contain debugged composite multidisciplinary components ready to be integrated inside client applications. It is worth noticing that, very often, with small applications, we cannot really make a difference between the two packages (*Generic*

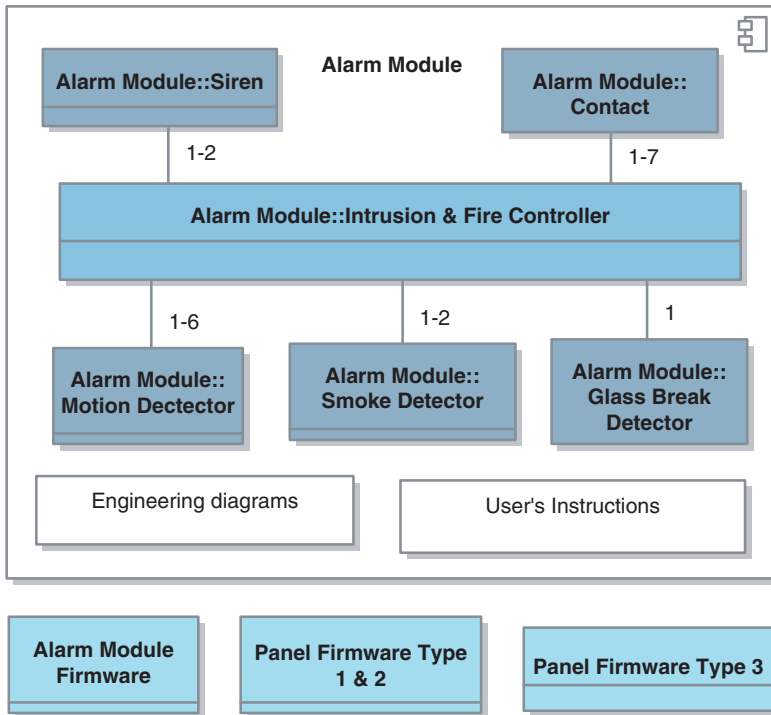


Fig. 6.5.6.4 Contents of Reusable Components package (Alarm System). This package contains monolithic components with well-defined interfaces, and well-specified characteristics, artifacts, and documentation. Component classes can be derived as objects/components in any user's application. Classes are connected together by associations whose real meaning can be understood only when examining Engineering Diagrams (artifacts). Each component is an elementary system defined at a smaller scale than the Client application, but its internal structure and behavior are well known. The last three classes "Firmware" give all firmware compatible with the component

*Architectures* and *Reusable components*). From the development viewpoint, *Generic Architectures package* contains simply elementary *constructs* but not *components*. The term *components* implies monolithic constructs with well-defined interfaces, well-specified characteristics, and well-documented OEM pieces that can be easily integrated inside real application. Moreover, components can be sold often “as is” in the OEM market. So, in the presence of some confusion, try to answer “whether elements of Reusable Components package can be or not easily converted into COTS.” If the company decides to change its policy and sell COTS instead of addressing to the end user market, this package will be of topmost importance.

This hypothetical example illustrates the long process of designing applications based on reuse at several levels (Classes, Generic Micro-Architectures, and Components). The contribution of original development for any new application is reduced to its minimum. The maximum reuse portion is null for the first time, and will be reached after several developments. A 100% reuse occurs when a Customer buys a standard product developed for another customer in the past. It is not always evident that client applications can really make use of reusable components defined in the corresponding packages. For an unusual client application, it is possible that the Developer must redefine entirely new classes, generic architectures, and builds new components (Fig. 6.5.6.5).

### 6.5.7 Completing Dynamic Studies with UML State Machine and Activity Diagrams

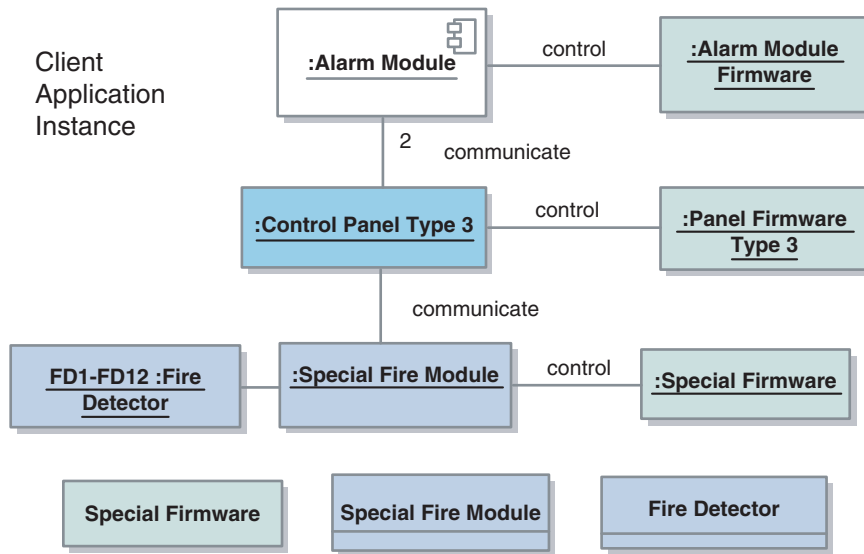
One of the objectives of the MDA is the possibility of verifying early in the logical phase nonplatform-dependent algorithms that govern the way objects behave individually and in collaboration with other objects in the execution of a given task. Each task has its proper set of participating actors and each actor participates with a subset of operations.

In a system like the Home Alarm, tasks are for instance *Idle* (alarm disconnected, wait for rearming), *Monitoring with infrared sensors*, *Monitoring without infrared sensors* (alarm armed, watch various sensors).

With the first task *Idle*, the Keypad is monitored to wait for the rearming code, and the main door switch acts as a condition for rearming. Other alarm sensors do not participate in this task.

With the second task *Monitoring with infrared sensors*, the Keypad wait for command whereas the Alarm Module monitor all sensors when the occupants are out of the protected volume. All objects participate in this collaboration. The third task *Monitoring without infrared sensors* excludes all infrared sensor objects.

Before digging into the way we study the behavior of systems, let us synthesize some results scattered in the preceding chapters.



Classes to be redirected to Generic Classes Package for reuse purposes

Fig. 6.5.6.5 Instance of Customer application (Alarm System). A Customer application contains objects instead of classes. They combined objects instantiated from various packages (Reusable Components, Generic Architectures, Generic Classes) taking into account Customer Requirements as inputs. This diagram illustrates the reuse of a Control Panel Type 3 to control two Alarm Modules and the engineering of a Special Fire Module specifically for a Customer. This special module is developed with special firmware and will be redirected to Generic Classes package for reuse

1. *State and activity are close concepts.* As stated in fundamental concepts, activities and states are very close concepts, state machine diagrams, or activity diagrams that can be used to study the behavior of objects. It is not uncommon to mix them.
2. *Dynamic behavior of a simple and passive object is often trivial.* Some objects are trivial so they do not need any dynamic investigation. For instance, the dynamic behavior of a switch or a button contains only *on()* and *off()* operation and *status* as attributes is not worth a study.
  - (a) A passive object was defined in Fundamental Concepts as an object that does not require service from other objects and does not take any initiative to modify its environment. Passive objects can be alive objects; in this case, they may have a very rich behavior or state evolution. So the rule is not always true.

3. *Complex objects cannot be studied dynamically.* If some objects have a trivial behavior, complex objects cannot be studied dynamically if they are not decomposed correctly. So, *dynamic studies can be conducted on sufficiently elementary objects that are not trivial.* Algorithms can be built on those objects then assembled or combined to get the algorithms of complex objects. Dynamic studies are both top-down (to identify tasks and to find out all constituting elements of a system) but the process of building algorithms, therefore object dynamic behaviors is bottom up.
  - (a) In the Alarm System, if we say that the Controller of the Control Panel will undergo three states Arming process, Wait for arming delay expired, and Disarming Process, this assertion cannot help us build the firmware as the system is still considered at its high level. A more thorough decomposition, highlighting the main door contact, the programming buttons 0–9, the sound beeper, and the LED lights will dictate in what order commands must be entered into the system. The resulting algorithm can be used to make the firmware of the Control Panel.
4. *The state of a whole system is often considered as the resulting states of all of its collaborative objects and semantics can be fuzzy.* As stated in the Fundamental Concepts chapter, the state of a whole system cannot always describe the states of all of its components. Sometimes, a synthetic term is hard to find for characterizing the state of a whole from those of its parts.
  - (a) A teacher is exposing his subject in a classroom. Saying that “the attention is moderate” cannot make any people figure out that only half of the class is listening, the rest are either somnolent, working, or gaming on their laptops.

The process of studying the dynamic behavior of a system can be split into several steps:

1. *First, identify individual tasks, simple tasks, or activities.*
  - (a) Individual tasks are those executed by a single object by opposition to a collaborative task.
  - (b) A simple task is a task executed with the collaboration of passive objects only.
  - (c) A task or an activity is not really an object operation although at low level a task is implemented by an object operation or method. An elementary operation is tied to the nature of an object, what it can do, but an operation is not goal oriented. For instance, a controller can

occasionally make a conversion because it is an intelligent object and has internal resource to realize this conversion but this operation is not considered as a task or an activity. If a controller scans buttons to detect a command, it is executing a simple task.

2. *Do not consider parallelism at the beginning of the dynamic study.* If several tasks must be executed in parallel in a real context, it is imperative to first study characteristics of each task separately. If a given object system requires simple forms of parallelism, this constraint must be inherent to the nature of the logical task but not a precocious temptation towards implementation. For instance, when activating Ctrl-Alt-Del with the keyboard, we send a command with three simultaneous actions.
3. *With an object diagram, show all objects that participate to the task execution.* Classes are used essentially in real-time systems to store attributes (structural, functional, and dynamic). The use of associations in class diagrams are not recommended since the object diagram (or a communication) is more expressive to represent collaboration scenarios for executing identified tasks. Objects will be connected with links (equivalent to associations in a class diagram).
  - (a) If a microwave oven has several buttons Start, Stop, Clock, Cook, 0–9 etc. a class diagram will represent only a class Controller connected to a single class Button, so the semantics is unclear. In an object diagram, all objects are represented with their links to the Controller.
4. *At each phase of a task, an object contributes to the task with its proper operation and its state will evolve through the execution of this operation.* If the algorithm is complex, it happens that an object may contribute to the global task with more than one operation. In this case, it is worth considering the several phases in the participation of this object, each phase will be studied separately with its own dynamic diagram.
  - (a) For instance, in the process of arming and disarming an alarm, the Controller of the Control Panel will be solicited with three operations *Arming process*, *Wait for arming delay expired*, and *Disarming Process*. It is worth considering the three separate dynamic studies as they call for three separate operations.

Figure 6.5.7.1 shows the state machine diagram of an arming process that brings an alarm system from the *unarmed state* to the *armed state* (the reverse process is not represented). The state displayed must be the state of a unique object,



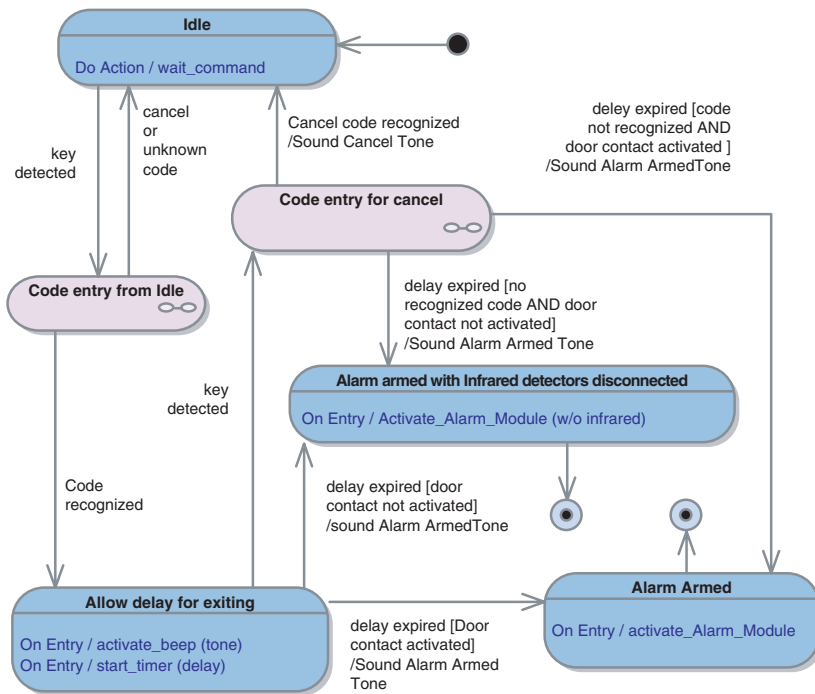


Fig. 6.5.7.1 State machine diagram of the Panel Controller showing activities accompanying the arming process. This state machine diagram supposes a better decomposition of the alarm system and therefore, it is more detailed than the state machine diagram of Figure 6.5.5.3 established at the start of the design process. Code Entry states are composite states and must be decomposed in other diagrams. From Alarm Armed states, we can come back to Idle state with Alarm disarmed, but corresponding transitions and states are not represented

in our case, the Panel Controller that is the intelligent element of the system. As said before, the way passive components (button, contact, etc.) interact with active components (controller, etc.) is not specified (polling or interruption) at this logical phase. Events sent from the passive to the active components are a valid approach with the principle of induced energy exposed in Section 6.1.3. When the owner acts on a button, he transmits induced energy to the button that closes the contact and sends an event to the controller.

States displayed in Figure 6.5.7.1 are states of the *Control Panel Controller*. Another controller in the *Alarm Module* is responsible for the monitoring of all sensors. The *Alarm Module Controller* must be studied separately. The two systems are considered as independent even if at the implementation step, we merge the two controllers into one unique active device. This example illustrates the first logical model built directly from specification. The design breaks the system into its utmost independent functions and assigns each important and intelligent task to a separate and independent “virtual” controller. No scheduling

(unless the problem is the scheduling process itself) is necessary at this time even if timing constraints are parts of the specifications. Depending on the application, timing constraint can remain as constraint at the logical phase and took into account only at the implementation phase.

One of the key concepts of achieving a good logical model is the early separation of constraints imposed intrinsically by the nature of the real-time problem itself and the constraints generated by implementation trade-offs. The second aspect will be taken into account only at the implementation phase. Controllers in the logical phase are considered as intelligent components empowered by “lightning speed” processors in order to separate *functional dependencies* from the *organizational dependencies* coming from the proposed solution.

To illustrate this important separation between timing constraints issued from functional dependencies and organizational dependencies, let us consider a student solving a problem with a computer. The problem has three dependent steps S1, S2, then S3 taken in this order. If  $S_i$  needs  $T_i$  times, the time taken to solve the problem is  $T_{123} = T_1 + T_2 + T_3$ . If  $T_1 = T_2 = T_3 = T$  and there is no functional dependencies, a group of three students with three computers (organizational arrangement) can solve the problem in  $T$  time  $T = T_{123}/3$  when there is no functional dependency. So, functional dependency cannot be optimized with organizational arrangement. If a meal needs 10 min. to be cooked, the best chef or even an army of chefs cannot reduce this time. The problem of the logical phase is to determine that the meal needs really 10 min. to be cooked and to demonstrate that it is the fastest delay.

### 6.5.8 Assigning Task Responsibility

A task requires some work, some time or can bring a small change/move of the microworld built around a given system. A task can trigger another task. The notion of unit task is dependent on the way a system is decomposed and approached. A unit task or “elementary task” will be affected to the most elementary object/agent/component identified in this system so the concept of “element” is very elastic; it varies with the context and concern.

Facing task responsibility, we have some specific recurrent problem with object modeling. First, when a task is elementary and reach sufficient low level (implementation level), it is executed by a single object and become an operation or method of the object, sometimes; it is difficult to identify the goal of the task, particularly when this operation is defined only to support a collaborative task (for instance, if an LED is flashing, without a high-level identification, it is difficult to know why it is flashing). In the opposite direction, if we can identify easily the object/agent/component responsible for elementary or individual tasks, the task analysis of high level and collaborative tasks is more complicated and may be affected by many factors:

1. *The responsibility is collective.* As the task is of collaborative nature, the part of responsibility of each participating object/agent/ component is modulated by some criteria retained for the task (execution time, frequency of implication, who is the task inventor, who finances the task, who is nominated or elected to be the task organizer, etc.) Reasons are not only technical but can bear some social, political, or human facets. If the gaming rule is well defined, they act as constraints and technically there is no difficulty of assigning tasks to objects/agents/components. The problem is more stringent when there is no clear rule or when some reasons enumerated in the next points impair the analysis process.
2. *An unclear task analysis pleads for a fuzzy responsibility assignment.* When defining a task, the decomposition is not conducted to reach individual and elementary object/agent/component, so the task is ill defined at the lowest level so it happens as if, from the development viewpoint, a flaw in the way the system is designed and decomposed. People have a very limited view on the way the task must be executed.
3. *Inefficient or inexistent dynamic study.* Tasks may be defined but the algorithms are not established. Objects/agents/components may execute tasks in a contradictory, unplanned order at run-time or worse, some people are doing things that they are not allowed to do or things that are out of their roles in the organization, etc. In the latter case, we have both a dynamic and structural flaw while designing the system. Design errors impact on responsibility attributed to objects that participate into the collaboration.

Our purpose is to point out that technically, we must build entities like a group, a bureau, a committee, etc. that constitute role aggregates to support a task or an operation. Those aggregates will have complex tasks as operations. They contain roles with a subdivision for responsibilities. In some circumstances, it may happen that individual object/agent/component cannot entirely take the responsibility of a given task even if it is the instigator, the trigger, even sometimes the author if the task calls for a collaboration of numerous stakeholders at run-time and worse, there is no clear definition of the notion of responsibility in the system. Political and social systems are generally subject to poorly designed criteria, technical products are easier to handle from this viewpoint.

Tasks are tied to structures in object technology as we must find a class to host any task. If a task is itself so important, for instance *fight\_terrorism()*, a full featured class can be built to host just this operation. A thorough decomposition will identify all subtasks needed to execute this high-level operation and all organizations inside a given country that take part in the responsibility of conducting this process. For continuation, read points 1, 2, and 3.

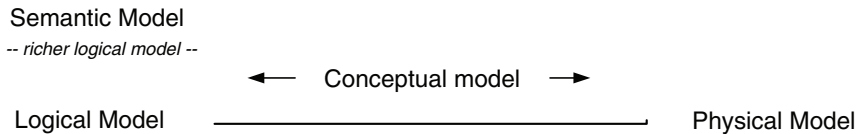


Fig. 6.6.0.1 Physical, logical, conceptual, and semantic models on a user perception axis of low or high level of data representation

## 6.6 Designing Reusable Database Models

Our goal is not explaining how to transform, with object technology, conceptual models currently used in the domain of relational database. The literature in the computer science domain is rather tremendous about modeling databases, relational or object, with entities-relationships or UML diagrams (Fig. 6.6.0.1). Commonly, a *logical model* of data exposes the logical view of data organization without any reference to performance or storage issues. The *physical model* is platform dependent, it shows access paths, files, indexes, and storage organization. Between these two poles, we find conceptual models that can fluctuate between these two poles according to its “conceptualized” degree. Too much optimization storage constraints will draw the conceptual model towards the physical side. Conceptual models provide the mapping from the logical to the physical models. A *semantic model* provides more semantic content (by authorizing relationships like inheritance and aggregation) than a common logical model. So, semantic models appear as richer logical models. It can cover the conceptual zone partly.

Object or relational explains the way data are globally organized. Roughly, relational means a spreadsheet organization with multiple tables and object means a graph-based representation of data. Some modern databases mix both object and relational organizations. Both the models are useful. If we make a description of a car, it is more natural to use a graph-based description with the whole car at the root of a tree structure. Leaves are made of most elementary mechanical, electrical pieces entering into the constitution of the car. But, if we manage a store of car spare parts, it is more natural to have a table view of all pieces, their part number, their price, etc. The application itself must dictate the kind of physical model that must be built. If we have to manage the stock of parts, a physical model based on tables is the ideal answer. But if we must make a description of the car and explain, from an engineering viewpoint, how this car must be modified to work both with electricity and gas, an object description will be the best approach. Saying that the best database system is object (or relational) from an absolute viewpoint is simply nonsense. The model used cannot be disconnected from its application. Before going further with the discussion, we suppose that “fundamental concepts” of Chapter 5 are fully assimilated.

### 6.6.1 Categorization and Classification: Reuse of Static Structures

Objects are real world. There is no class in real world. Class is an intellectual invention. A classification system is a hierarchical structure of well-defined classes nested in a series of inheritance relationships. A classification provides a powerful cognitive tool that minimizes the cognitive load on individuals by embedding information about reality through the class structure. Class membership recognition can be seen as a relatively simple pattern-matching. Relationships link classes together through *hierarchical* (inheritance) or *lateral* relationships (normal association). The overall structure of a classification system serves as a repository for the storage of information about classes, and information about their interactions. By capitalizing on the hierarchical and lateral relationships between classes, this structuring process reduces the load on our memory.

*Categorization* is the previous step before *organization*. Categorization groups entities on the basis of similarities based on some selected criteria but a system fully categorized may lack meaningful, informative relationships so it does not necessarily constitute an organization. A system is organized according to a set of criteria. If we change the criteria, we obtain another organization.

All humans are theoretically equal regardless of race, color etc. but the poor are adversely affected due to their inequality before the lawyers. The affluent have therefore more opportunities of evading the rigor of the law. So, if the initial group is split into two different groups, the organization structure is therefore different as we introduce another classification criterion. Good classification criteria give good answer to systems.

This observation of the destruction of a structure when the organization criteria are changed is a destabilizing factor. From an operational viewpoint, it is like relationships between classes are dynamic, so they can change at run-time. *Databases are built fundamentally on static relationships*. The reuse of databases schemas is possible only if the set of relationships remains static and invariable with time.

### 6.6.2 Reusability Issue of Some Conceptual Database Models

We start with an example of a database used for instance for identifying an object Person in a classical approach that shows the weaknesses of such approach regarding the reuse mechanism (those databases can be very helpful instead in regular situation).

A *Person* class (Fig. 6.6.2.1) is declared currently in the past with all attributes that allow a security agent or a policeman to control fully the identity of this person. With this design, the following situations need a database update: change of the

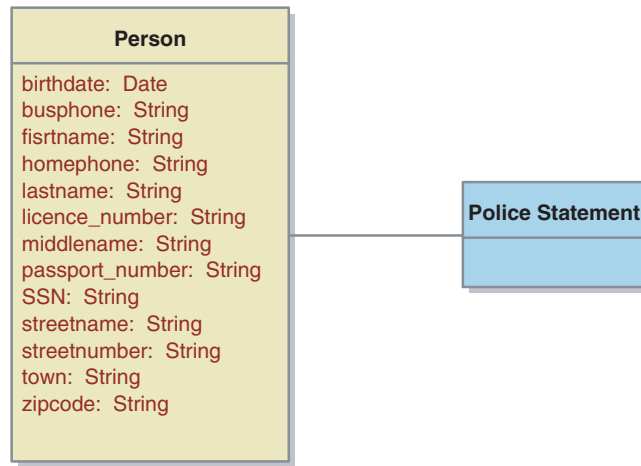


Fig. 6.6.2.1 Classic conceptual Person class often seen in database model

passport number when renewing, moving to another address, the town decides to change the street name, etc. Moreover, if the names of the person in the Passport or the Driver License do not correspond exactly (for instance abbreviations of the middle name) to the names registered in this database, the person cannot be identified correctly and he can occasionally have a lot of trouble having to justify this discrepancy.

This classic design is indeed oriented towards the final application and mimics all information that a person must fill in by hand, for instance, in a police statement. In the worst case, if the physical model has only one statement, we can put all fields inside one unique class. In Fig. 6.6.2.2, there is a slight effort of conceptualization as two classes are identified (class *Person* with all his personal information and another class *Address*). Even qualified as conceptual, this kind of model is very close to the *physical model* on the Perception Axis of Fig. 6.6.0.1.

The poor reusability comes from many facts:

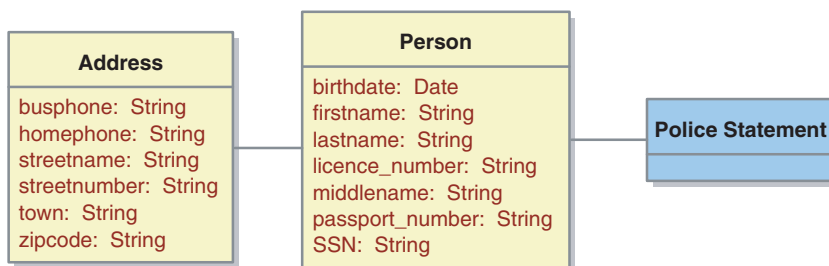


Fig. 6.6.2.2 Slight correction of the model of Figure 6.6.2.1

1. *Design is oriented towards physical support.* If the form reproduces the physical support, all small change to this support will call for a conceptual model alteration.
2. *Insufficient decomposition so information is packed into a same class.* Generally, a correction of the model of Figure 6.6.2.1 proposed is splitting the Person class into two classes *Person* and *Address*, so we now have three classes (Fig. 6.6.2.2). The social security number and the passport number are still parts of the class Person attributes. Phone numbers are parts of Address class.
3. Relationships specific to an application are not separated from more stable ones. Unfortunately, we cannot prove this point with this unsophisticated example.

### 6.6.3 Conceptualizing Domain and Modeling Reality: Database Models

The key concept of reuse is still, as in real-time systems, domain conceptualization of data instead of jumping directly to, in an application, to its final view of data. Moreover, if we mimic what really happens in reality, we can expect that the next application will be based on the same reality but with a different context. If we push the concept a little further to its final retrench, the number of objects in each domain is roughly static but the way they interact may vary greatly with time, so it would be interesting to identify classes of the domain then enrich them with attributes and operations each time we have a new problem to be solved. Possibly, new problems may call for creation of new classes, but this process is much slower than the process of enriching existing classes with new attributes and new operations.

Moreover, a uniform view of the whole application can be got by intimately connecting classes created in a database with real-time classes. Modern applications are complex and involve real-time applications coupled with some forms of data handling, so these two aspects are inseparable. For instance, in an airport, when designing registering systems for passengers and their baggage, and security systems for controlling passenger access operations, databases are intimately connected to real-time systems so we can model the whole chain of applications with the same object formalism.

To temporarily summarize previous ideas and make a step for designing reusable data models, here is the proposal:

1. *Identify classes of the domain* that match some information required by the physical model while avoiding to make classes uniquely for the current application. The validity of the class is weighted through its survival

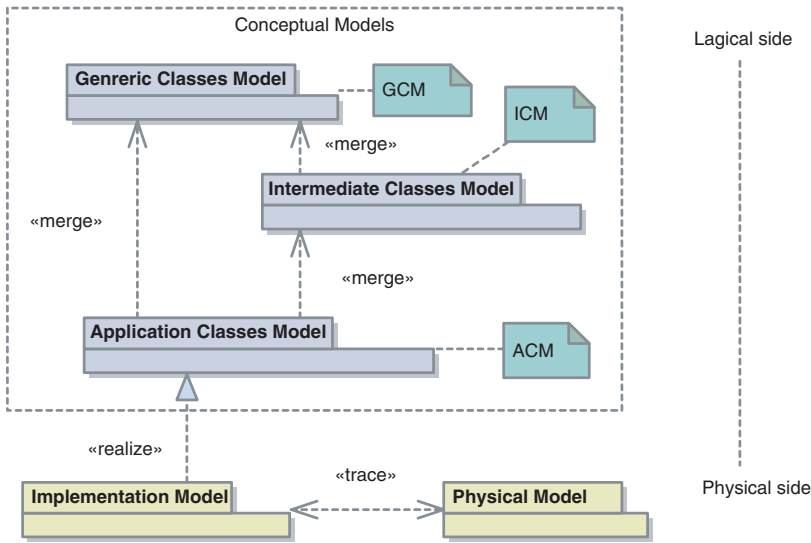


Fig. 6.6.3.1 Organization of data models, From (GCM) Generic Classes Model to the Physical Models. GCM, ICM, and ACM are all conceptual models. *Implementation Model* contains data tables, views, transactions, etc. in relational database. *Physical Model* contains files, index files, data repositories, computer nodes, geographic repartition of data, etc.

- resistance to *space* (can the database be exported to another province or another country without modification?) and *time* (is the database structure susceptible to be modified in a near future?).
2. *Have at least two packages: one for generic classes and one oriented towards the application model.* Generic classes are part of the *Generic Classes Model* (GCM) and the conceptual model designed for the application is the *Application Classes Model* (ACM). It is not excluded that *Intermediate Classes Models* (ICM) can be built to host more structured components. The ICM corresponds to Generic Architecture package in real-time systems. Figure 6.6.3.1 illustrates the package reflecting the model organization.
  3. *Use real world classes only.* Real world classes have the best chance to be there the next time another application arises. Very sophisticated or complicated classes are generally not real and thus have to be avoided.
  4. *Attributes created for GCM classes are universal and tightly tied to the intrinsic nature of objects.* This is the key factor of success in the reuse process. If an attribute is not tightly tied to the nature of the class, it is probably an attribute of another real class that must be identified as well.



5. *Use only evident inheritance or aggregation/composition relationships* for classes of the domain. These relationships must be evaluated against its resistance to space and time.
6. *Specific associations are drawn only in the ACM (or ICM) to avoid spoiling the GCM.* As said before, the ACM is application oriented and the GCM is domain oriented; hence, user defined relationships must be put in the right packages.

#### 6.6.4 Generic Classes Model

To start building a reusable library supporting the uniform methodology, we illustrate the contents of some useful and universal domains that can be reused later in our case studies. Some classes are still not broken down into their utmost elementary components but the goal is to explain the methodology.

First, the GCM top package contains four individual packages *Humans*, *Locations*, *Identifications*, and *Immobilizations* (Fig. 6.6.4.1). The number of packages is limited for explaining only the methodology. When entering the *Humans* package, we have the most important class that describes all particularities of a human that have a chance to last in space and time.

If we put identification card numbers, addresses, or phone numbers inside this class, we bring it to the physical side and the class will have less chance to be reused in other context. A person changes his address, and his phone number very often. The driver license number and the SSN numbers are more stable in time, the passport number does expire in some countries. The current practice of inserting those numbers in a *Person* class as identification strings does not hold as all identification cards of a person are separate objects, managed by independent or governmental organizations. A *Person* has an *association* with its *Passport*, his *SSN card*, or his *Driver License* instead of holding them as attributes. If those data are tied to the class *Person*, we violate the reuse concept both in space and time. Identification data are subjected to modification in time and the choice of card identification is a cultural matter. In most undeveloped countries, the notion of SSN identification does not even exist. Moreover, identification methods vary with time and circumstances. Recently, with terrorist threat, sophisticated biological signatures are proposed and used to identify humans. By constituting a separated package, we can add all identification means and connect them to a class *Person* in an application context. When dealing with multimedia attributes, we create at a logical level, attributes like *picture or image*, *soundtrack*, *videosequence*, etc. leaving all choice of implementation (LOB: Large Objects, BLOB: Binary LOB or CLOB: Character LOB, etc.) to be specified in implementation models.

Moreover, an identification card or a passport is effectively a separate object, very different from a person. A passport may have a first name and a last name

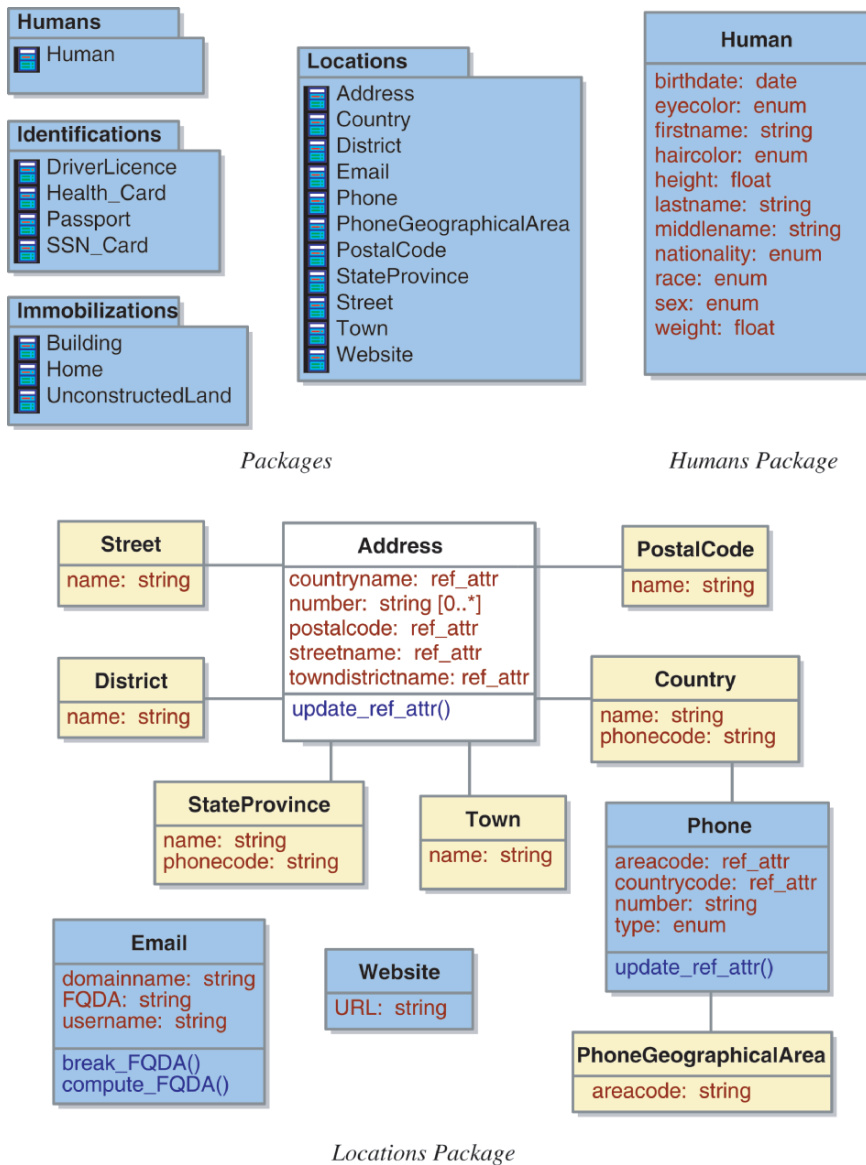
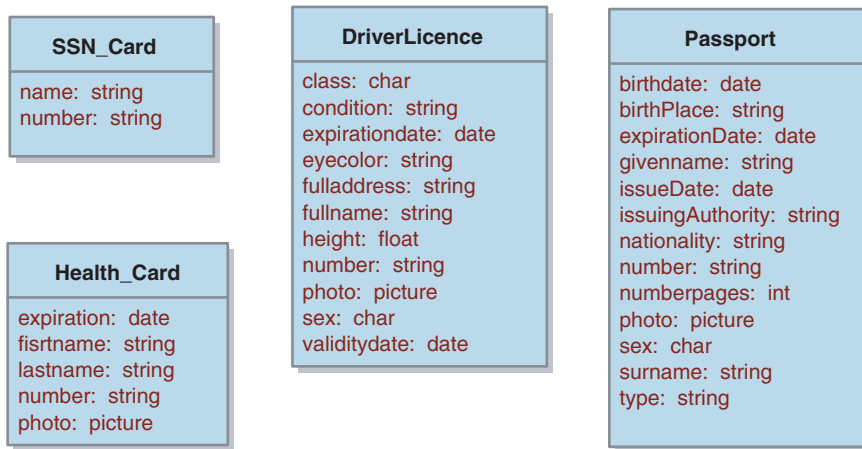
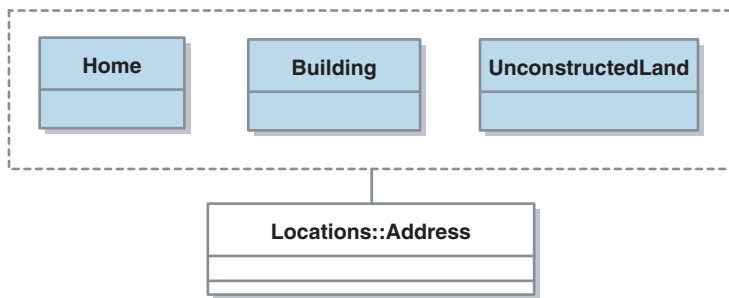


Fig. 6.6.4.1 Elementary GCM (Generic Classes Model) elements of Humans, Identifications, Locations, Immobilizations packages illustrating the contents of Generic Classes Model. The notation **Locations::Address** indicates that this class is defined in Locations package and there is an individual association between **Home**, **Building**, **UnconstructedLand** classes (contour property) and **Address** class

*Identifications Package**Immobilizations Package**Fig. 6.6.4.1 (Continued)*

on it. Their orthography may differ from first name and last name on other documents. So representing them as various documents linked to a person allow us to detect unsuspected problems as inconsistent identifications on various documents, forgery, identification stealing, etc.

The Address class is an example of complex data specification as an address is composed of many individual fields that are dependent of other objects. In many actual designs, the resident phone number is considered as an attribute of the address and this fact is inconveniencing as, in many countries, a person may ask the telephone company to keep the same phone number when moving, if the new location is inside the same area. In this case, the phone number is not a component of an address, or a component of a person. So, the Phone is considered over here as a separated class. Area code is not specifically an attribute of a phone number but of the region. Much complication may arise as area codes

usually indicate geographical areas within one country/region/province that are covered by hundreds of telephone exchange devices, not necessarily administrative frontiers. The *PhoneGeographicalArea* is therefore created and not connected to the rest because in this generic package, we do not instantiate a real situation. Nongeographical numbers, as mobile telephones (outside of the USA and Canada), do not have an area code even though they are usually written as if they do.

The same difficulty is observed for postal codes, known in various countries as a post code, postcode, or ZIP code. It is a series of letters and/or digits appended to a postal address for the purpose of sorting mail. Although postal codes are usually assigned to geographical areas, special codes may be assigned to individual addresses or to institutions with large volumes of post, such as government agencies and large commercial companies (French Cedex system).

If a real situation calls for a design for a multilingual application, cultural aspects interfere with the way the class *Address* will be made. Other classes need to be created and linked to classes derived from *Address* to add several address types.

The *ref\_attr* type used in this logical model has a very subtle role. First, it attests that *the attribute is not recognized as a proper attribute of the current class but this class needs a copy of this information* to be considered complete at higher level. *Address* is not a composite class either, because it cannot contain other components, but it “*imports a copy*” of information from other components. *Address* class is therefore an *information class* (a class that gives pieces of information in a given context) but not a *real class* (image class of a real world objects). Information classes are application dependent (ICM or ACM) and are not part of GCM, except for some classes whose use has been so generalized that they become generic (case of *Address* class). Connections between *Address* class and *Country* class, for example, allow *ref\_attr* to reach an instance of *Country* for retrieving a value of a specific attribute.

The *ref\_attr* can be mapped differently from GCM to ACM. There could be a discrepancy in the way the logical model is designed and the implementation data model as well.

At the implementation phase, if we want a simpler database system (economic consideration), a reference attribute can be mapped naturally into the type of the attribute it is referencing (for instance, *countryname* of class *Address* will take the *string* type that is the type of *name* in the class *Country*). In this case, the *ref\_attr* loses one indirection and becomes a plain attribute. We have just decided to include the country as a plain attribute instead of creating a navigational structure. In this context, if a postal code must be changed for all residents of a street, each instance of the address contains a copy of the old postal code value so the update process must be fired on all *Address* instances. If objects are separate and linked, when making a reference to an instance of the class *PostalCode*, we must change only the name of one *PostalCode* instance.

The *Phone* class is completely disconnected from the *Address* class in this design proposal. In the past, a home phone number is often associated to an address and a mobile number is associated to a person. Very often, when editing a form, most databases give a choice of one business phone number, one home number, one fax number, and one mobile number. If, occasionally, a person wants to write a second alternate home phone, it is generally impossible to accommodate this case. Users have finally no choice, they can decide to enter extra phone numbers inside *comment* fields or equivalents (if they exist), generally neither indexable nor searchable.

The old rigid model can be made more flexible by remarking that a phone number is not attached to an address nor a person but has an independent existence (this independence corresponds to reality: when moving, sometimes we cannot keep the old phone number so this number will be affected to another person) and the relationship between a phone number and a person is simply a temporary association.

Phone numbers are not considered as attributes of class *Person*. Each newly created phone number is now tied to the class *Person* through an association, we can now instantiate an object *:Person* then a multitude of phone number objects (*firstnum:Phone . . . lastnum:Phone*, etc.) for this person, and the number of phone numbers can now vary from one person to another according to their specific needs. So, a person can possess an infinite number of addresses and each property, and an infinite number of phones.

Attributes of a class characterize this class. The UML metamodel is designed with an ownership relationship between a class and its attributes. The ownership relationship is often misleading. For instance, when putting an address as attributes of a person, we declare that the property lying at this address is permanently or statically tied to this person for an undetermined time frame. Modern life tends towards very dynamic situations. A person owns a property. A home owns an address (true, in old design since *address* is attribute of *Home* and a home belongs to a person, therefore, address is an attribute of *Person*; questionable in object design since *Address* and *Home* are two objects linked by an association only and *Home* is not permanently linked to *Person*). Parents own their children or husband their spouse. A car owns all its parts (true), a company owns all its departments (true), a company owns all its employees (false, they work for the company), a person owns all his money (arguable, he has all his money and can decide how to use them; very different from the “car owns all of its parts” example because money cannot define functionalities and abilities of a person). In an unknown case, it would be more cautious to use an association instead of an ownership or aggregation/composition relationship.

To summarize, a generic class targeted to be reused is likely to exist across space, resist to time erosion, if it is designed naturally as a real class of the world. Attributes must be tightly attached to this class in all (or at least “most”)

circumstances. A class with reference attributes (e.g. *Address* class) is less generic but it is highly useful as it collects useful information into a unique *Information Class* targeted to support a given application. Reference attributes are logical and attest that the information is loosely tied to the current class. At implementation level, reference attributes can be temporarily transformed into real attributes if this complexity is not needed. In this case, we will get back into more classical database schemas. But, at the logical level, generic classes can support reuse in other contexts. While working with database system, to really catch the subtleties of all data classes and attributes, it would be interesting to break a system into its utmost elementary classes then rebuild information classes.

### 6.6.5 Intermediate and Application Classes Models

The ICM and ACM are application models. The ICM plays the role of microarchitectures package that can be imported or merged directly into an application defined by the ACM. To give an image of the role of the ICM and ACM, to make a building plan, an architect makes individual plans showing all type of elementary construction elements chosen for the entering into the composition of rooms, kitchens, parlors, bathrooms, etc. (doors, faucets, fireplace, bathtub, shower, dim switches, etc.). This inventory is equivalent to our GCM level. Later, he can assemble building components to make complete space arrangements (various bathroom types, various room arrangements, various kitchen sizes, etc.). This level corresponds to the ICM level. In the final ACM level, the whole building arrangement can be now described with components of the ICM level (and occasionally with GCM level if necessary).

In the previous example, from *Address* class that has no specific regional specificities, we can derive *AddressUS* class for USA, *AddressCA* for Canada, *AddressFR* for France, and for other countries in the world that can have different fields, and put them in an ICM package. Operations defined for each address class *AddressXX* are intended to verify data input for this country and their algorithms will be different from one country to another. Roughly, GCM can be seen as common to all applications, ICM common to a group of applications and ACM is targeted for a given application.

## 6.7 Unifying Real-Time and Database Applications

Real-time system developers and database developers adopt traditionally two different development strategies. When developing databases, we create schemas. Schemas are generally drawn with E-R tools or with class diagram of UML tools. Real-time applications are developed with functional models or object models with the UML, and implemented with development platforms based on common languages as C, C++, Basic, Assembly, C#, Java, etc. Unifying the two development platforms has been attempted in the past, mostly in

the direction of object persistence, and is considered by many as a commercial failure (Keller, OOP-2004, <http://www.objectarchitects.de>). We do not really address this issue. At the research viewpoint, the important question is “is it useful to unify real time and database applications?” At first sight, these two application kinds seem to be disconnected.

A real-time application considers a system from the object viewpoint, builds real-time classes to model real-time objects. It defines attributes and operations of classes, studies dynamic behavior of objects through the execution of individual or collaborative tasks. Attributes are of three types: structural, functional, and dynamic. Real-time objects are individual objects, have identities, work and act on the surrounding world. The static structure of a real-time object is a *forest of tree structures* based on the aggregation/composition relationship as structuring criterion. To describe the operation of a *Car* and its states, subsystems extracted from the *Car* tree structure are considered as interacting objects that requires a *Driver* object as a source of events, a *Route* object as the environment that the car acts on, etc.

Database management system considers other kinds of problems. A car retailer may hold a store of spare parts of the same car, but the problem that worries him is to maintain always a correct stock to support car repair operations and occasionally direct sale to other service stations. A complete database application includes usual sale operations (requisition, purchasing, enquiries, report, inventory functions, data import/export, etc.). It contains the same pieces of the preceding tree structure mentioned in the real time application. Attributes stored in the database are of structural and technical types found in the previous real-time application. Another set of attributes in Business Domain (part price, date of reception, date of sale, etc.) are added to handle financial operations. When manipulating business data, the best view is undoubtedly a *spreadsheet view* or a *table view*. It would be really laborious for instance to calculate a global stock price by exploring a tree structure. Conversely, it is also hard to join relational tables to get a description of a car from the engineering viewpoint.

The most important aspect is: parts in the described database have no identification. If we have 100 carburetors, they are all identical. The part number used for the inventory is not an individual identification; it allows the system to categorize the carburetor and to make sure that it will go inside a type of car for some production period. Occasionally, we can have a distinct inventory number that seems to make the part unique, but in fact, the sale person can take any carburetor and give it to anyone.

Relational databases are said to be value based and is adapted to applications that required a spreadsheet or table view of data.

In the Car example, a line (record) in a table inventories all car parts having the same price and the same part number. So, objects are not identified individually in this context. As many as 100 carburetors need only one record with a field that counts the number remaining in stock.

When a supermarket stores 100 cans of maple syrup, it does not make sense to identify them individually. A unique record is therefore needed to count and characterize all objects of the same kind. So, effectively, we can find several

applications where real-time application and database applications concern the same objects but, as the purposes of the two applications are far from each other, they can be developed as two independent applications and there is no reason to unify real-time and database projects in this context, arguing on the single fact that they refer to the same physical object.

However, there are several applications in which we need to control the evolution of an identified object, and simultaneously, store structural attributes of this object in a database in order to retrieve its description when needed. Moreover, each object needs a *separate record* in a table. The application therefore requires a real-time object (evolutional and behavioral aspects) and its description (structural aspect).

In the previous example of car parts database, if we suppose that people must make a data entry for each carburetor (100 entries for 100 parts), there is no clear evidence of the advantage of viewing all car part data with a table or spreadsheet view. We can instantiate each carburetor one by one, identify them individually. Objects are ready to work in a real-time environment and their attributes are simultaneously indexed for data management purposes. Here are some examples:

The Canadian gun registry requires every firearm in Canada to be registered or rendered in an unusable state. This was an effort to reduce crime by making every gun traceable. In this case, each gun must have a separate record.

A baggage handling service in an airport is another example of database that requires both a real-time tracing and transfer of the baggage and a database management system. The system receives information in real time from various applications that support the distribution of transfer baggage, the control of the baggage offloading points, the amount of traffic on the conveyors, etc.

The unifying process is therefore desirable when objects must be *individually registered*, and *their states monitored at real time*. Objects in this context are full-fledged and the model necessitates many kinds of attributes: *structural* for identifying the objects, *behavioral* for tracing its evolution in a real-time context. Real-time classes contain large number of real-time operations to describe the transfer, handling, distribution of the baggage, and state attributes to store states that result from those operations.

When we need to work with such an object, we instantiate a copy from its class and follow its evolution through space and time. The way we figure out the list of all objects is still a table but, at this time, the table is exceptionally crowded as each object needs one separate entry.

If the Car store of spare parts contains 2 million parts, the number of entries is limited to the number of part numbers. This is not the case with a system that requires individual tracking. The number of baggage entries is exactly that which is registered. If a security system must track all passenger states, each passenger must have its own record.

Handling baggage is a very complex scheduling problem that requires simultaneously a real-time model and a database model. Simulations are conducted to detect deadlocks, for instance, to detect the resources to be made available to



handle an airliner arrival. An interesting article describes the “Denver Airport Baggage Fiasco” in 1997 (Swartz, 1997).

To model an application that needs unification (e.g. baggage), several solutions were proposed:

1. *Handle the whole system in a real-time context with data persistence management.* Classes contain both structural (for database and real-time applications) and behavioral attributes (for real time applications). Operations embrace conventional database operations with real-time operations
  - (a) Structural attributes are for instance baggage weights, dimensions. Real time operations are those that accompany the transfer of the baggage through multiple steps, its security inspection (passed, not passed, etc.), that describe its individual states (normal, damaged, lost, etc.).
2. *Handle the whole system entirely with a relational DBMS (or eventually an ODBMS or ORDBMS) with limited real time extensions*
3. *Keep the two independent tools and make use modern techniques of accessing to databases.* Nowadays, it is very easy to connect any programming platform to a database and perform conventional database operations. Techniques like JDBC, ODBC, ADO.NET etc. are common and allow a standard programming language to pass a SQL clause to ask the xDBMS to perform a database operation. So the connection between the two apparently very different platforms is not really a problem
  - (b) The baggage system continues to handle two independent tools: standard real-time environment without persistence management and a relational database to store structural and state attributes of baggage. Synchronization processes must be defined along the transfer line of baggage from the check-in step towards its final delivery to the traveler. An *information class* can be added in the database to store, at every step of the transfer, state attributes of the baggage.

The third system is probably the most appealing method to connect those two applications. In the global system, now coexist two objects that must be synchronized, a real object, e.g. *bagRT:Baggage* in the memory of any real-time system along the transfer line and a record *bagDB:BaggageDB* in the memory of the database system. To explain the synchronization, let us suppose, for instance, that a security system detects a firearm packed inside the real baggage, it changes the state of *security* attribute from *unchecked* to *suspected* in a local *bagRT*, fires an alarm, echoes this value immediately to *bagDB* to warn other

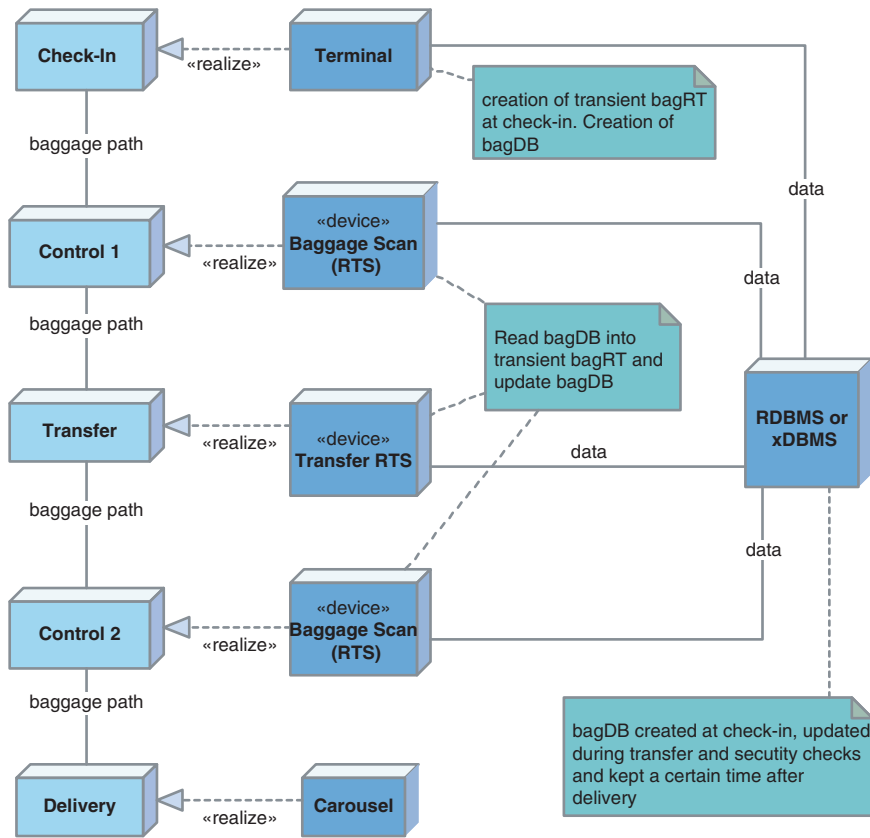


Fig. 6.7.0.1 Deployment diagram explaining that many bagRT are created temporarily (they can be made persistent locally). The object bagDB is persistent in the database and can be reached all over the network. Each real-time object is defined differently according to the nature of the device. Some of their attributes are read dynamically from bagDB if needed. bagDB is updated after visiting each node so bagDB keeps track of the current location and state of the baggage

services of the airport so that security countermeasures can be immediately deployed to intercept the real baggage corresponding to the current clone *bagRT*. Figure 6.7.0.1 represents is a deployment diagram explaining the relationship between multiple real time objects bagRT (each bagRT is created locally inside each real time application) and a database object or record bagDB (coming from an *information class BaggageDB* created in the RDBMS).

Data models used in business domains and objects in engineering worlds are generally not structured in the same way. So, the nature of the application will dictate finally what the model that must be used. This fact is certainly one of the reasons that explain why relational model with table and spreadsheet view of data still occupy a dominant position in the market. It fits perfectly to

the business domain. In engineering domains or in knowledge representation, where data are structured as networks or forest of tree structures, modeling data with object style will be the best answer. However, logical models must be dissociated from implementation models. If knowledge and engineering databases must be modeled with object style, their implementation could be easily mapped into a hybrid system of type 3 mixing an object environment to a relational database with software cement (ADO, JDBC, etc.). The baggage transfer example highlights such a solution. Several models must be developed to support such an arrangement. Real-time models are based on *real-time classes*. Database model are based on *information classes*. At run-time, corresponding objects are instantiated and they require continuous synchronization to maintain data coherence.