Chapter 8

Case Studies

8.1 Design of an Inclined Elevator or Track-Lift Tram

Our purpose aims to illustrate the Uniform Methodology by choosing a system made of objects taken from various disciplines. A simple inclined elevator going between two levels has all ingredients of such a system as it contains mechanical components (cabin, door, mechanical brakes), electromechanical components (buttons, microswitches, contact sensors, motors), intelligent components (microcontroller, electronic command for variable speed motors), biological (humans making use of the system), and computer software (program that give intelligence to the controller).

This simple and classical system is easily understood by everybody so it does not add other difficulties to the understanding of the methodology itself. Moreover, this system has inputs, outputs, close-loop control, hardware, software, and safety features (we can stop the tram with the Stop button in an emergency situation, a stalling motor triggers a breaker to go off) so we can expose all exception handling of this system and give a complete example of testable PIM model. On the other hand, we add some supplementary constraints in order to illustrate how to distribute constraints among development phases.

8.1.1 Early Requirements

A company LiftTram wants to manufacture an inclined elevator. Track-lift trams need an even slope. If the slope is not even and the terrain is rugged, a pair of cables can be used instead of the pair of tracks but fundamentally the two designs are based on the same engineering principle which includes a cabin running on wheels on either fixed tracks or static cables. The cabin is powered by a motorized drum hoist that drives the cabin up and down the tracks/cables. Two to four aerial cables (providing high safety at any charge) are used to move the cabin. Redundant limit switches are used at the high and low landings to ensure smooth stops.

The power is provided by a high efficiency three phase gear-motor with an internal brake (Motor Brake). If power is accidentally off, this brake locks *the motor shaft in place. The variable frequency motor drive allows gradual acceleration and deceleration.*

The cabin must have a large surface of glass panels to allow a clear view of the surroundings and the cabin must be designed in such a way that it can be customized to meet the client needs.

The cabin is equipped with mechanical brakes to lock it to the tracks in the event of slack aerial cables (Slack Cable Brake). If the tension on aerial cables is removed, tension detectors activate the Slack Cable Brake, lock the cabin to the tracks, and stop the motor.

The cabin is supported on the tracks with castors; two to four of them have each a centrifugal overspeed safety brake (Overspeed Brake) that will be activated if the speed reaches 15% over the rated speed.

A control unit is available inside of the cabin. This allows the operator to stop anywhere, besides automatic stops at the starting and the arriving points. The cabin has safety switches at the front and back to stop the cabin in the event of an obstruction on the tracks (Track Jam Brake).

If for an unknown reason, the battery of limit switches fail to stop the cabin at the high landing, a detection of a stalling motor (Current Surge Detector) stops the motor. At the low landing, the Slack Cable Brake has the same effect by activating the Slack Cable Brake.

Manual commands can be issued inside the cabin and the same commands are duplicated outside the cabin at each landing (three sets). Essentially, each security brake category has its corresponding lights red and green (the red light means "brake activated" and the green light means "brake released"). Each brake has a corresponding set and release button. If a brake is set manually, a signal is sent to the motor to stop it. We have also general "all brakes activated" and "all brakes released" buttons with their corresponding lights. The button "all brake activated can serve as emergency stop while the cabin is in motion."

A Cabin Door Lock is a magnetic switch used to prevent the cabin from moving in the event that the door is not completely closed. This lock also prevents the door from being opened while the cabin is moving. The door itself has a pair of monitoring lights.

To move the cabin, the operator glides a cursor up or down proportionally with the speed. When arriving at landings, the motor brake stops the cabin. To move the cabin in the opposite direction, the operator moves the cursor in the neutral zone before sending a release brake. "Release Brake" signal cannot be activated if the cursor is not in the neutral zone.

The following free form diagrams (Figs. 8.1.1.1 and 8.1.1.2) locate main components of the system.

Complementary information can be obtained at ASME (American Society of Mechanical Engineers). This organization (*http://www.asme.org*) has published

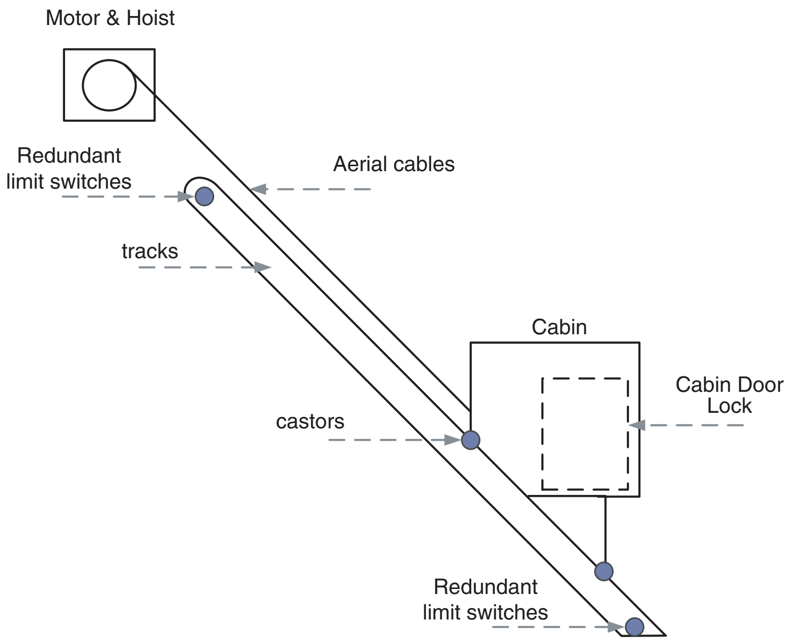


Fig. 8.1.1.1 Free form diagram showing main components of the inclined elevator

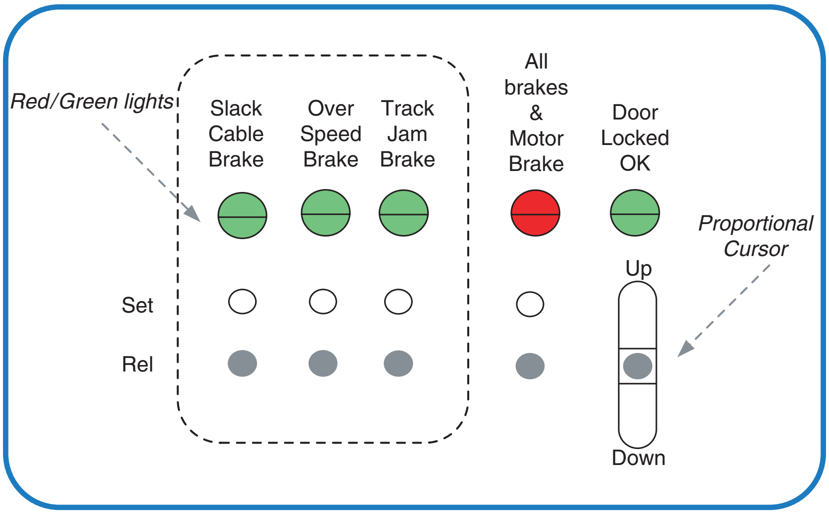


Fig. 8.1.1.2 Control Panel of the inclined elevator

Safety Code for Elevators and Escalators under the reference 17.1. Our purpose is to approach a real case to show how to model a system but we have not verified any conformance of our logical PIM to this standard.

8.1.2 Requirements Engineering: Text Analysis Phase

Section 6.2 has already exposed the goals of this preliminary phase that identifies all actors/objects, use cases, activities, relationships, attributes, and all concepts that allow us to build the SRS in a graphical form, extract and analyze information embedded in the text. Simple spreadsheet software can be used for this text analysis. All UML diagrams are eligible for specification, but the topmost diagrams recommended for this phase are: UC diagram, class diagram, object diagram, and BPD. Dynamic diagrams can be used if some important sequences are needed to be exposed.

It is important not to confuse this phase with the design phase. All we need are identifying activities that we can put as CRT parameters and so, the main parameters of the whole project can be estimated. Some activities must be decomposed to have a good estimation. The utmost goal of this phase is to show, from a technical viewpoint, that the company involved in the elaboration of the SRS document is a serious contender, it knows the technology, can realize the project, and has given a good estimation of CRT parameters, without necessarily exposing all its secret know-hows (design knowledge).

Table 8.1.2.1 Text analysis of Early Requirements (Inclined Elevator). First step of Requirements Engineering phase

Step	*Initial text*	*Concepts*	*Reformulation*
1	A company LiftTram wants to	Actor/Object	LiftTram Company
1a	manufacture an inclined elevator.	UC/Activity	Manufacture
1b	Track-lift trams need an even slope. If the slope is not even and the terrain is rugged,	Actor/Object	Inclined Elevator System or Track-Lift Tram
1c	a pair of cables can be used instead of the pair of tracks but fundamentally the two	Constraint	Even slope -> model with tracks
1d	designs are based on the same engineering principle which includes a cabin running	Constraint	Rugged slope -> model with cables
1e	on wheels on either fixed tracks or static cables	Goal	Reuse of common components
2	The cabin is powered by a motorized drum	Actor/Object	Cabin
2a	hoist that drives the cabin up and down the	Actor/Object	Drum Hoist
2b	tracks/cables	Actor/Object	Motor (and its Shaft will be merged together as they are indivisible)
2c		Actor/Object	Guiding Tracks/Cables
2d		UC/Activity	Power (Motor power Drum Hoist)

(*Cont.*)

Table 8.1.2.1 (*Continued*)

Step	*Initial text*	*Concepts*	*Reformulation*
3	Two to four aerial cables (providing high safety at any charge) are used to move the cabin. Redundant limit switches are used at the high and low landings to ensure smooth stops	Actor/Object	Aerial Cable
3a		Multiplicity	2..4 (Aerial Cable)
3b		UC/Activity	Pull (Cable pulls Cabin)
3c		Actor/Object	Limit Switches Low
3d		Actor/Object	Limit Switches High
3e		Actor/Object	Low Landing Position
3f		Actor/Object	High Landing Position
3g		Goal	Smooth stopping
4	The power is provided by a high efficiency three-phase gear-motor with an internal brake (Motor Brake). If power is accidentally off, this brake locks the motor shaft in place	Attribute	3-phase (attribute of Motor)
4a		Actor/Object	Internal Motor Brake
4b		UC/Activity	Lock if power off
4c		Actor/Object	Motor Shaft (merged into Motor)
5	The variable frequency motor drive allows gradual acceleration and deceleration	Actor/Object	Variable Frequency Drive
5a		UC/Activity	Accelerate motor gradually
5b		UC/Activity	Decelerate motor gradually
5c		UC/Activity	Keep speed
6	The cabin must have a large surface of glass panels to allow a clear view of the surroundings and the cabin must be designed in such a way that it can be customized to meet the client needs	Actor/Object	Glass Panel
6a		Multiplicity	* (many)
6b		Relationship	Contain (Cabin contains Glass Panels)
6c		Goal	Clear view of the surroundings
6d		Goal	Customization to meet Client needs
7	The cabin is equipped with mechanical brakes to lock the cabin to the tracks in the event of slack aerial cables (Slack Cable Brake). If the tension on aerial cables is removed, tension detectors activate the Slack Cable Brake, lock the cabin to the tracks, and stop the motor	Actor/Object	Slack Cable Brake
7a		Multiplicity	* (many)
7b		UC/Activity	Lock Cabin to Tracks
7c		Event	Slack tension (Aerial Cable)
7d		Actor/Object	Tension Detector
7e		UC/Activity	Stop Motor
8	The cabin is supported on the tracks with castors; two to four of them have each a centrifugal over-speed safety brake (Overspeed Brake) that will be activated if the speed reaches 15% over the rated speed	Actor/Object	Castor
8a		Multiplicity	4 (Castor on Tracks)
8b		Actor/Object	Overspeed Brake
8c		Relationship	Integrate (to Castor)
8d		Multiplicity	2..4
8e		Event	Overspeed detected
8f		Actor/Object	Overspeed Detector

(*Cont.*)

Table 8.1.2.1 (*Continued*)

Step	*Initial text*	*Concepts*	*Reformulation*
8g		UC/Activity	Activate (Overspeed Detector Activate Brake)
8h		Condition	Speed 15% over rated speed
9	A control unit is available inside of the cabin. This allows the operator to stop anywhere, besides automatic stops at the starting and the arriving points. The cabin has safety switches at the front and back to stop the cabin in the event of an obstruction on the tracks (Track Jam Brake)	Actor/Object	Control Unit
9a		UC/Activity	Stop (Cabin) anywhere
9b		UC/Activity	Stop at Low Landing
9c		UC/Activity	Stop at High Landing
9d		Actor/Object	Front Safety Switch
9e		Actor/Object	Back Safety Switch
9f		UC/Activity	Stop if obstruction
9g		Actor/Object	Obstruction Object
9h		Actor/Object	Track Jam Brake
9i		UC/Activity	Activate (Safety Switch activates Track Jam Brake)
9j		Relationship	Contain (Cabin has Safety Switches)
10	If for an unknown reason, the battery of limit switches fail to stop the cabin at the high landing, a detection of a stalling motor (Current Surge Detector) stops the motor. At the low landing, the Slack Cable Brake has the same effect by activating the Slack Cable Brake	Condition	Limit Switches fail
10a		Actor/Object	Current Surge Detector
10b		UC/Activity	Detect surge
10c		UC/Activity	Stop Motor at high landing
		UC/Activity	(Slack Cable Brake) stop Motor at low landing
11	Manual commands can be issued inside the cabin and the same commands are duplicated outside the cabin at each landing (three sets). Essentially, each security brake category has its corresponding lights red and green (the red light means "brake activated" and the green light means "brake released"). Each brake has a corresponding set and release button. If a brake is set manually, a signal is sent to the motor to stop it. We also in general have "all brakes activated" and "all brakes released" buttons with their corresponding lights. The button "all brake activated" can serve as an emergency stop while the cabin is in motion.	Multiplicity	3 (Control Unit mentioned at step 9)
11a		Actor/Object	Red/Green Lights (for each category of brakes and all brakes)
		UC/Activity	Set lights on/off
11b		Actor/Object	Set/Release Buttons (one set for each category of brake)
11c		UC/Activity	Set brake on/off (one set for each category of brake)
11d		Actor/Object	"Set all brakes" Button (Emergency Stop interpretation)
11e		Actor/Object	"Release all brakes" Button
11f		UC/Activity	Set all brakes on/off

(*Cont.*)

Table 8.1.2.1 (*Continued*)

Step	*Initial text*	*Concepts*	*Reformulation*
11g		UC/Activity	Set motor off (when one kind of brake is on)
12	A Cabin Door Lock is a magnetic switch	Actor/Object	Cabin Door Lock
12a	used to prevent the cabin from moving in	Actor/Object	Door
12b	the event that the door is not completely closed. This lock also prevents the door	Relationship	Contain (Cabin Door Lock attached to Door)
12c	from being opened while the cabin is moving. The door also has a pair of	Condition	Movement enable if Door Completely Closed
12d	monitoring lights	UC/Activity	Open Door during movement
12e		Actor/Object	Red/Green Lights (for Cabin Door Lock)
12f		UC/Activity	Set lights on/off (Cabin Door Lock)
13	To move the cabin, the operator glides a	Actor/Object	Cursor for speed control
13a	cursor up or down proportionally with the speed. When arriving at landings, the	UC/Activity	Move up/down proportionally
13b	motor brake stops the cabin. To move the	UC/Activity	Move to neutral zone
13c	cabin in the opposite direction, the operator moves the cursor in the neutral zone before sending a release brake.	Condition	Cursor must be in Neutral zone for releasing any brake
13d	"Release Brake" signal cannot be activated if the cursor is not in the neutral zone	Actor/Object	Operator of the Cabin

8.1.3 Building Classes for Instantiating Actors in Use Case Diagrams

As said in Section 6.3, our choice was to dissociate the requirements engineering objects completely from the design phase. The correspondence between objects, components, or functions will be established by a *mapping process*. Objects identified in a very short time frame (requirements analysis) are used to estimate the project importance and cannot be kept "as is" when addressing the complex design phase. Reuse considerations (classes/components will be merged from the organization reuse database), industrial secret cannot be unveiled at requirement analysis.

Moreover, a class identified at this phase generally has just identification. They serve to instantiate numerous objects needed to explain use cases and BPDs (or occasionally other dynamic diagrams). Attributes are absent except when their presence is unavoidable. Important operations/activities are translated into uses cases and more elementary ones are not relevant in this context. Figure 8.1.3.1 gives a list of classes identified by the text analysis process and is grouped according to their technological domains.

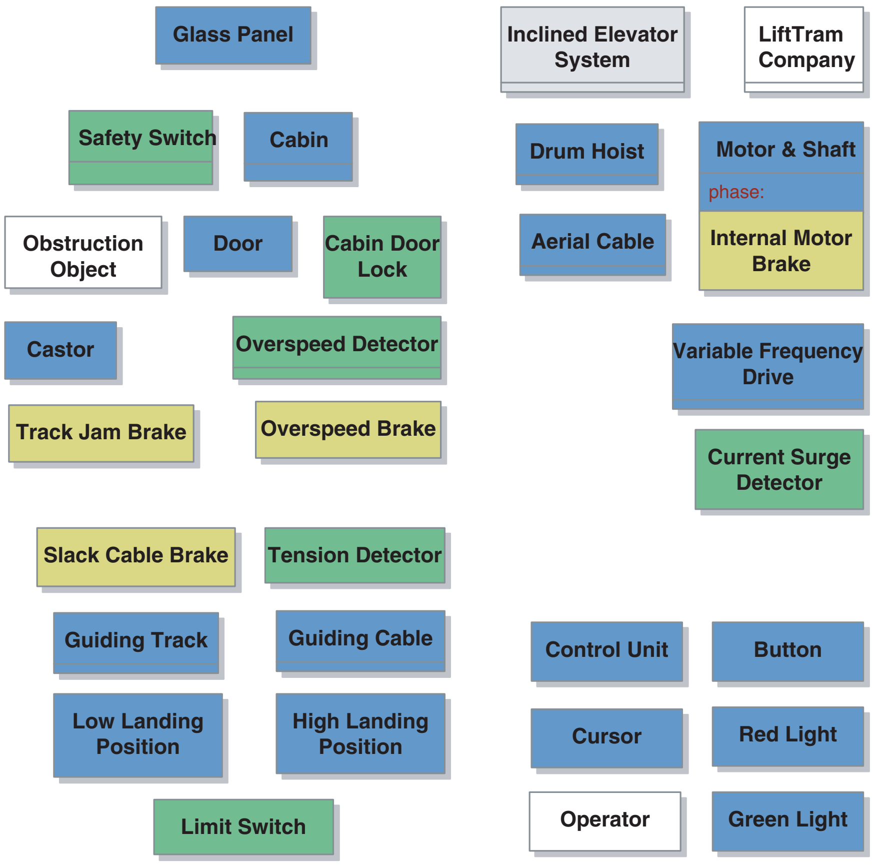


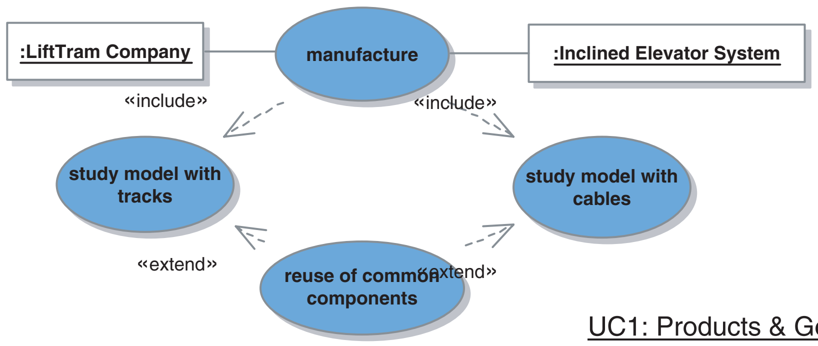
Fig. 8.1.3.1 Classes of the Inclined Elevator identified by Text Analysis of Early Requirements Specification

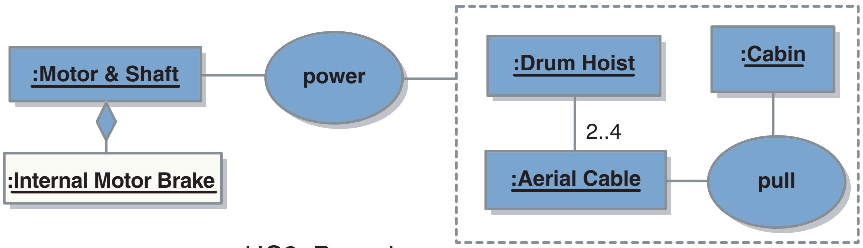
Classes of Figure 8.1.3.1 are generic and afford a first “evident” clustering of objects; for instance, *Safety Switch* will later give rise to *Front: Safety Switch* and *Front: Safety Switch* objects, *Limit Switch* will later give rise to *Low: Limit Switch* and *High: Limit Switch* objects. All buttons will be instantiated from a unique class *Button*. For *Red Light* and *Green Light*, we have the choice between making a unique class LED Light and putting an attribute *color* or making two different classes. We have chosen to show design variations. We make two classes as they correspond to two different electronic components (some LED integrate two colors into one light). For *Track Jam Brake*, *Overspeed Brake*, and *Slack Cable Brake*, we know from engineering experience that they are “brake type” and can be implemented later as a same brake system activated from several sensors. But, at the requirements, it is not mandatory to reveal early any technological solution. So, from a perspective of specification, it would be more interesting to specify functions (so why use cases are used) instead of specifying objects. Many functions can be implemented later by the same object and so project costs can be lowered accordingly. As a consequence of this discussion, some solutions at the Requirements Engineering phase may appear as not based on solid design criteria from a "designer perspective." But, as this preliminary phase is not a rigorous design phase and must be done in a very limited time frame, the impact of this "inconsistency" would be minor.

8.1.3.1 Requirements Diagrams. The previous class diagram was established without any relationships. The class diagram can be, at the limit, not joined to the SRS document as objects/actors are only relevant in UC diagrams that follow. The following diagrams reproduce rigorously clauses of Table 8.1.2.1 but mix clauses together and reorganize them in a more understandable way. If diagrams are named UC1, UC2, UC3, etc. (Fig. 8.1.3.1.1). these names can be registered in a supplementary column of this table allowing cross referencing. Each UC diagram exposes a simple or complex clause, and a new idea of the specification and we must avoid the temptation to mix all clauses together. Comments are put immediately after each use case to explain the techniques.

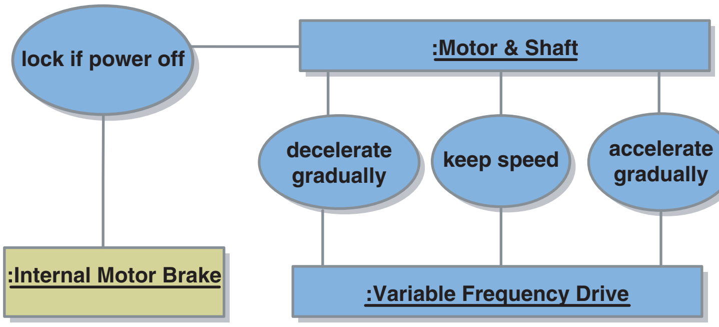
A use case often involves many actors or many objects. The "simple line" connection means, as said in the theoretical part, "concern" or "imply." A use case involves several actors/objects and allows us to identify the underlying activity. It is not mandatory to represent all involved actors or objects, just the main ones. The "extend" relationship shows that the reuse is optional (in our case, it will be deployed whenever possible). Each diagram tells a "scenario in a story."

The motor contains its internal brake. It powers the whole system inside the contour. The contour is a "white box" showing a drum hoist associating with 2–4 aerial cables pulling the cabin. So, a UC diagram in our case mixes intimately a classical UC diagram to an object diagram. The link between Motor/Brake is of aggregation type, the link between Hoist and Cable objects is not necessary to be defined.





UC2: Powering

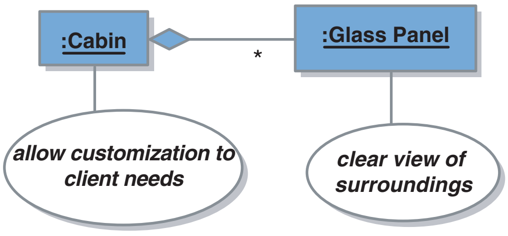


UC3: Motor, Command and Integrated Security

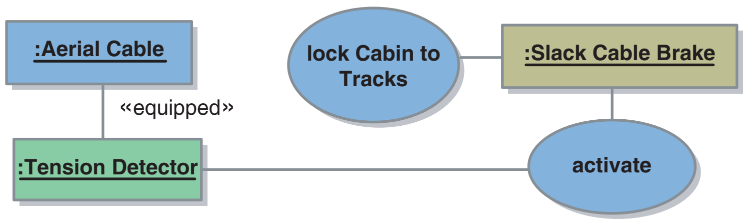
The internal motor brake locks the shaft (that is inseparable from the motor). A variable frequency drive powers the motor, accelerates, decelerates gradually, or keeps a constant speed.

The comfort elements normally will not enter into our logical design PIM but will be taken into account at the implementation phase. As said in the theoretical part, all constraints are disseminated among development models, some in PIM, others in PSM. In this diagram, we have an example of use case deployed to represent goals. In this case, we recommend making them abstract and transparent to distinguish them from regular UC that we must compute CRT parameters.

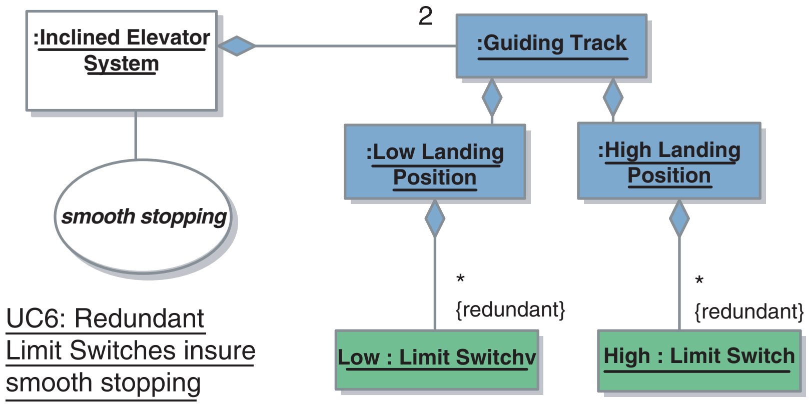
The stereotyped relationship <<equipped>> is drawn between two objects and, by this way, saves a use case. When saying for instance that "an aerial cable has a tension detector," we think of the "light" aggregation form. If mechanically, this connection between an aerial cable and a tension detector needs a complete process from which CRT parameters must be evaluated (not this case), it would be more appropriate to choose a representation that evidences the use case (activity).


:Cabin
*
:Glass Panel
allow customization to client needs
clear view of surroundings

UC4: Comfort elements



UC5: Slack Cable Protection and Tension Detector

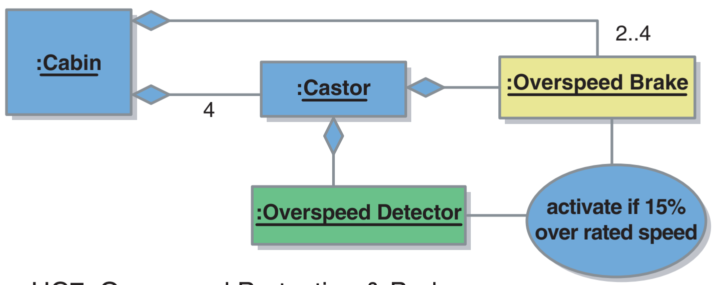


UC6: Redundant Limit Switches insure smooth stopping

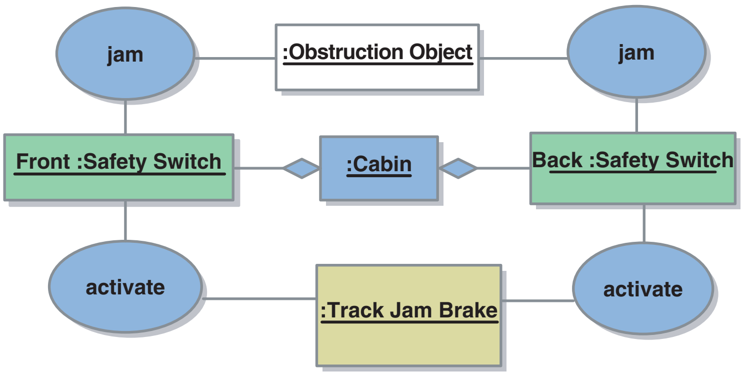
Most aggregation/composition relationships can be interpreted as "is equipped with". They serve mainly to pack several components inside one logical description (See full discussion of this relationship in Section 5.5.5).

In a UC diagram, it would be difficult to represent a condition and make it visible as there is only use case and actor. We can choose to write down the condition "15% over rated speed" as comment, or in the constraint field of the use case (not visible in some tools). The simplest way is (if possible) entering it in the use case itself.

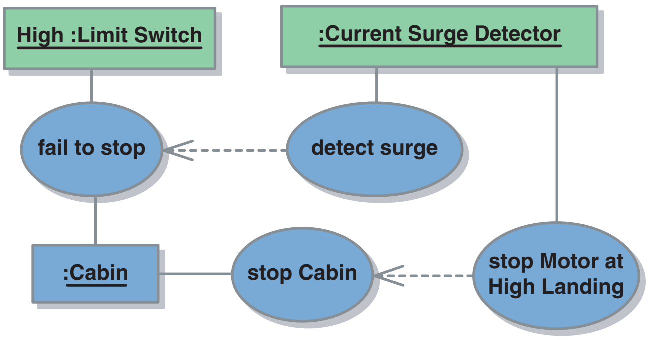
As said before, we create an object "Track Jam Brake" because at this phase, it would be irrelevant to discuss about solutions. So, objects of this kind may be introduced at requirements engineering phase to materialize a function that must be realized. "Obstruction Object" is not part of the system (it explains only the function that must be implemented) so its representation is alleviated.



UC7: Overspeed Protection & Brake

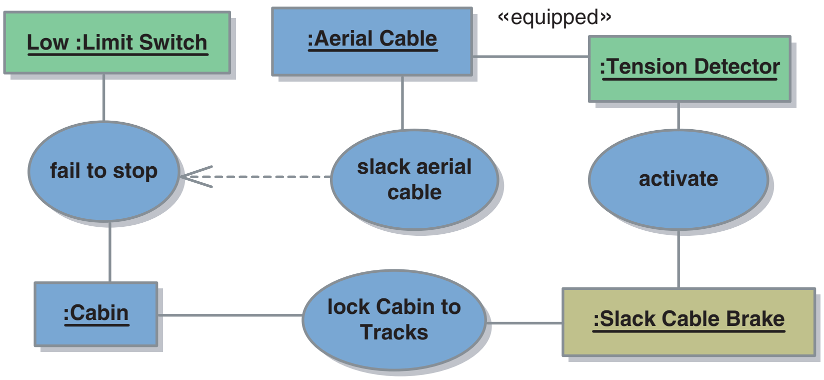


UC8: Track Obstruction Protection & Brake

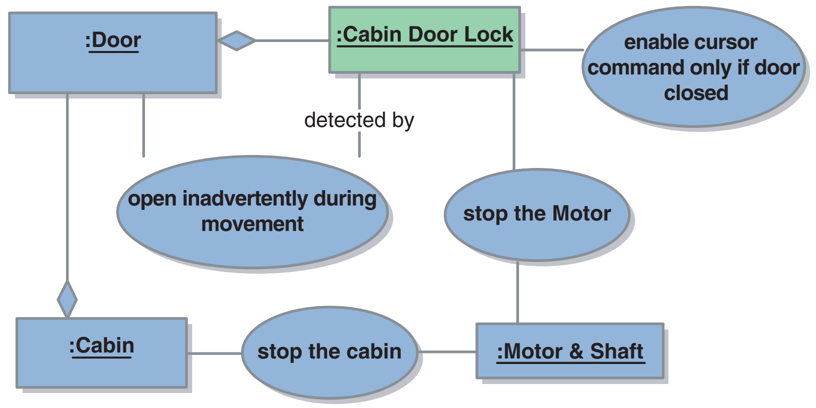


UC9: Fail to stop at High Limit & Protection

We have seen that the connection between two use cases can be <<include>> or <<extend>>, a series of actions in a cause-effect chain can be represented with the general <<dependency>> relationship generally not stereotyped. If the cabin fails to stop, a surge will occur. If the surge detector stops the motor, the



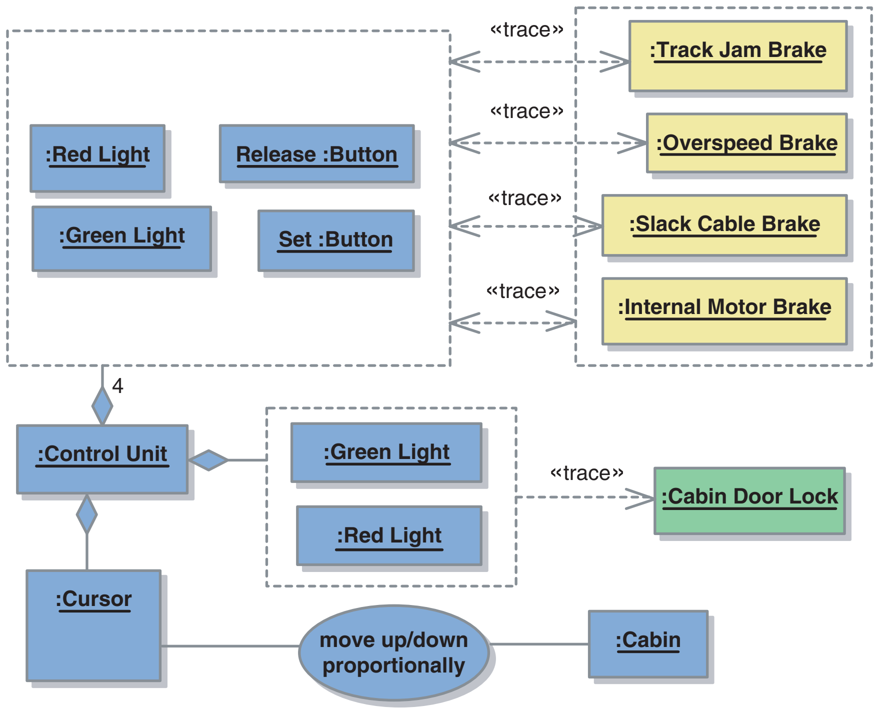
UC10: Fail to stop at Low Limit Protection



UC11: Door Lock Protection during movement. Cursor enabled only if Door locked

cabin will be stopped accordingly. The "stop Cabin" use case is tied to "stop Motor at High Landing" in a dependency relationship.

The UC12 diagram makes use of contour properties to save representation. The Control Unit has four sets of Red/Green Lights, Release/Set Buttons. Each set



UC12: Control Unit

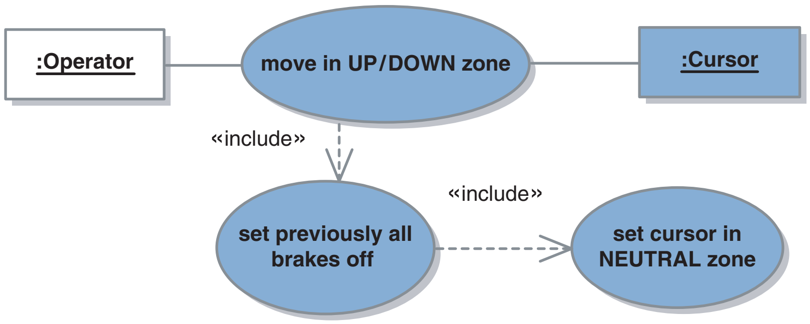


Fig. 8.1.3.1.1 Thirteen use case diagrams showing activities of the Inclined Elevator System from which CRT parameters must be evaluated

is attached to its corresponding protection. A <<trace>> relationship of bidirectional type is bound to each set as buttons are inputs and lights are outputs. Generally, this relationships is used when two actors/objects must evolve together with some dependence, but, for simplifying the representation, we cannot explicit the whole cause-effect chain.

The "include" relationship means "constraint" imposed to the use case. To move the cursor up and down, all brakes must be off. Brakes cannot be set to off if the cursor is not set in its neutral zone, so when all brakes are released, the Cabin cannot move before the Operator acts progressively on the cursor.

8.1.4 PIM of the Inclined Elevator System: Creating the Generic Classes Package

Normally, a re-spec phase mentioned in theoretical part must precede the design phase. We integrate these changes into the design discussion.

Hereafter, we simulate the reuse mechanism by creating a Generic Classes package containing several levels of packages. The domain is oriented towards the domain of the LiftTram company, but, a close examination of classes declared in those packages shows that, except for mechanical classes of the Generic Mechanical Classes package targeted to solve specifically the problem of the LiftTram company, all other packages can be reused in several domains of electrical and computer engineering. We restrict the declared generic classes to the context of this problem only but, in real situations, the number of packages is not limited.

If the Generic Classes package exists, just import it into the current project. If it exists but the number of classes is not sufficient, just create new classes if they can be easily identified at this step (for instance, the project has identified in the SRS document the presence of a double cursor with two elementary cursors Up and Down with a neutral zone in the middle. This class can be created immediately in the Generic Electromechanical Classes package). Otherwise, they can be added later when they appear in the project.

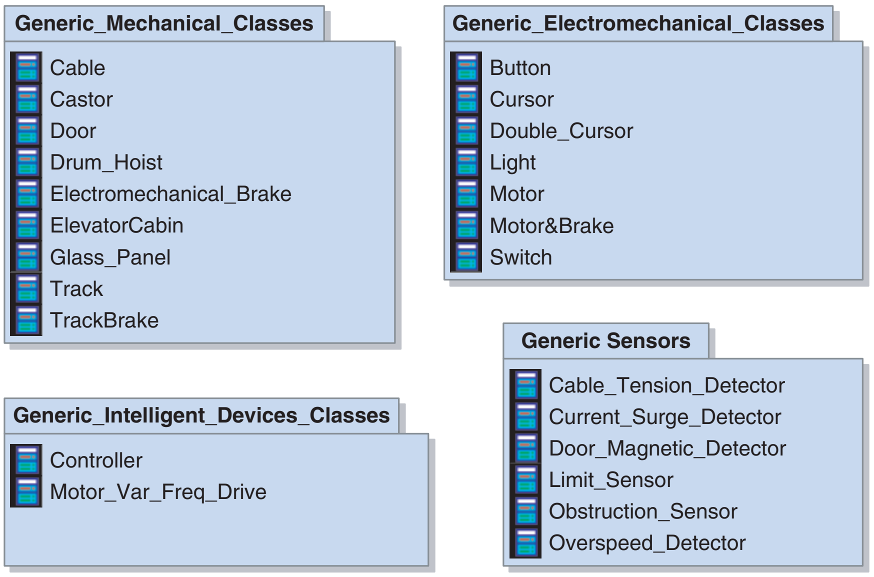


Fig. 8.1.4.1 Generic Classes package in use in LiftTram Company

The Generic Mechanical Classes package shows classes in use in the Lift-Tram Company (Fig. 8.1.4.1). All other packages are general and can be reused outside the Elevator domain.

Generic Classes are stuffed with their logical operations to support the PIM study (Fig. 8.1.4.2). They are not yet mapped into technological components

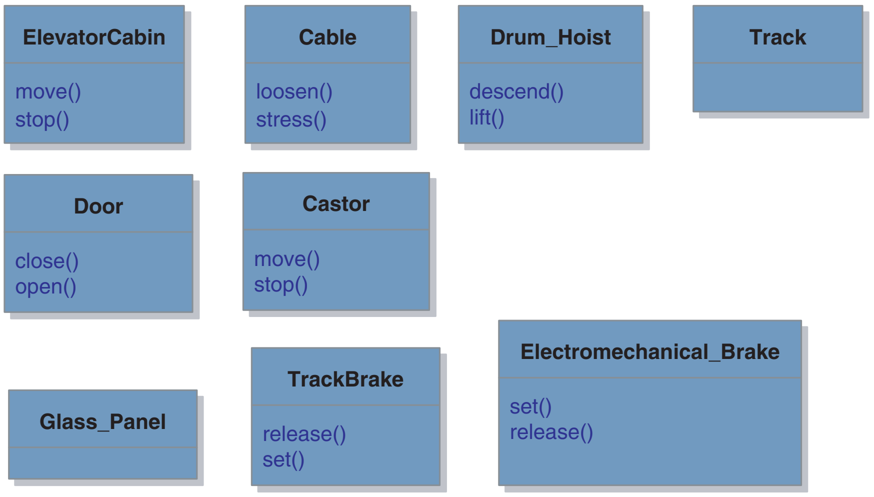


Fig. 8.1.4.2 Details of the Generic Mechanical Classes package. These classes are specific to the Elevator Domain

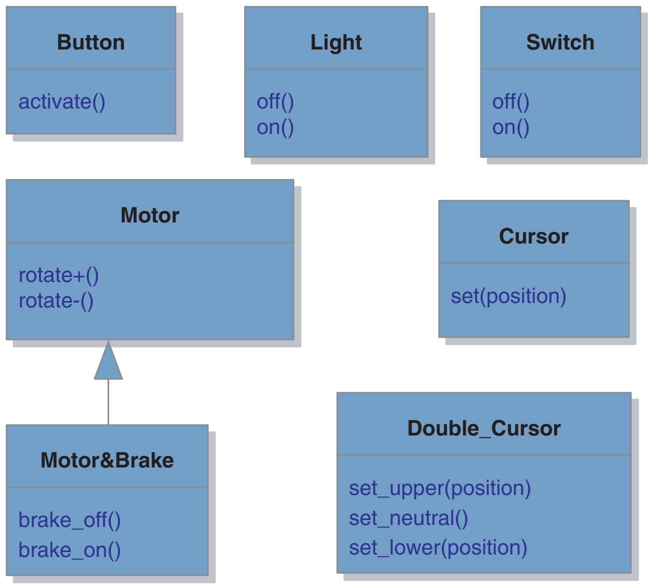


Fig. 8.1.4.3 Details of the Generic Electromechanical Classes package

so all common attributes (for instance, strength of the cable, dimensions of the Elevator Cabin, etc.) are irrelevant at this design step (of making reusable PIM).

One of the most important corrections being added to the original project and the SRS document at the re-spec phase was the creation of an electromechanical brake. This change comes from the fact that we cannot use the internal motor brake (designed to immobilize the shaft when the power is accidentally switched off) to stop the motor shaft in normal operation mode (at high or low landings). Moreover, the brake must be allowed only when the speed of the motor is very low, without hampering the performance of the VFD at low creep speeds (some of the main reasons for having VFD control). So, the electromechanical brake must operate only when the motor speed is very low and drops under a preset threshold. Therefore, either we need an input of the motor speed to the controller, or, for economic reasons, we can use a simple delay adjusted to the mechanical configuration. For the moment, in the logical design, we cannot say where this electromechanical brake must be installed (for instance, on the motor or on the hoist; this decision will be postponed to the PSM phase), but we suppose that the Controller can send a logical command to set/release the brake (Fig. 8.1.4.3).

Operations in generic classes are reused in several contexts. Operations must be identified with appropriate names that reflect the nature of the class but NEVER name them in the context of a given application. For instance, a motor has a shaft that can rotate clockwise (+) or counterclockwise (−). If, in the context of the elevator, we name operations of the motor *up()* and *down()*, we just destroy the reuse mechanism, since we interpret these operations in the context of the elevator domain. If the same kind of motor is used later to open or close gliding doors, the naming scheme, *up()* and *down()*, becomes obsolete quickly.

The Motor&Brake class is derived from the Motor class. This derivation adds two operations *brake_off()* and *brake_on()* to existing operations *rotate+()* and *rotate–()* of the Motor. We do not derive the Double_Cursor class from the Cursor class as they are very different electronic components. The double cursor rather aggregates two cursors and adds a mechanical neutral zone to the overall design. This modeling difficulty (aggregation or inheritance) has already been discussed in Section 5.5.7.

In this PIM design, the Light class is not identified as a LED Light class; despite the fact that all electronic engineers think that it is the way to implement this visual component. The same consideration has already been found in the mechanical package (Figure 8.1.4.2) in which we have not distinguished a steel Cable class used to guide the tram or a cable used to power the cabin, contrary to what was been announced in the SRS document. This genericity holds for all classes of the PIM.

Classes like Cursor and Switch are not used in this project but are drawn to show that this package always has more classes than needed.

The same genericity has been observed in the way operation are conceived as attributes and operations (Fig. 8.1.4.4). There is no specification of the implementation types (float, Boolean, integer, etc.). Operations have parameters but no implementation types are specified. As discussed in the theoretical part,

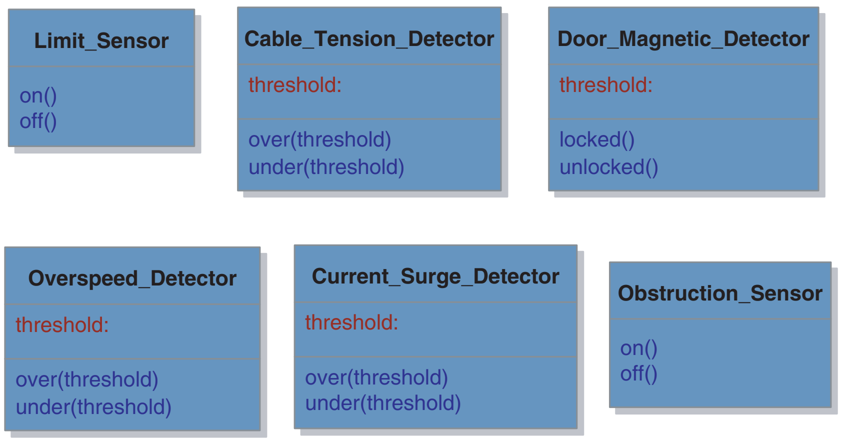


Fig. 8.1.4.4 Details of the Generic Sensor Classes package

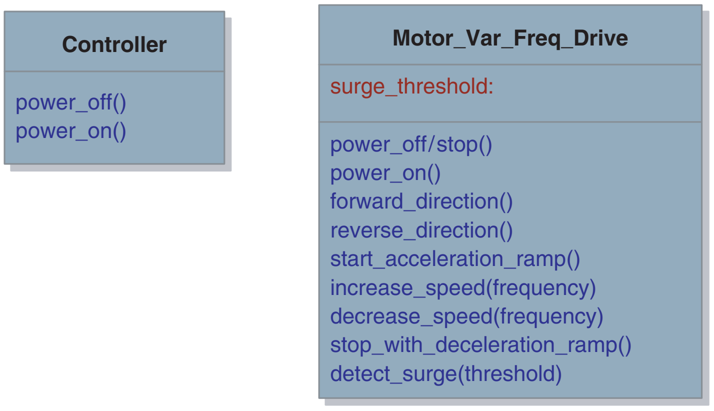


Fig. 8.1.4.5 Details of the General Intelligent Devices Classes package

to support logical models, we think that the low-level specification currently available must be enhanced in the future to meet MDA needs.

In a VFD application, the motor reacts to the varying voltage and frequency from the drive and develops a torque for the load. So doing, it draws current from the VFD. The current increases with the load. The direction of rotation can be set with *forward_direction()* or *reverse_direction()*. Stopping the motor does not mean powering it off. If the drive is currently set at some speed, it will undergo a fastest deceleration ramp to immobilize the motor shaft. We can put a power off that produces a brutal stop but this feature is not a regular command of the VFD.

The detection of the surge current has been integrated to the VFD. If such a feature is not available at implementation phase, we have the possibility of designing this feature as an add-in device but the correction can be made in the PSM.

For the intelligent controller (Fig. 8.1.4.5), as it is a very complex and versatile device, there is nothing specific or generic for this device. We can practically create any operation for it; this fact explains why there is no specific operation defined for this component. In real situation, we can define many types of controllers (microprocessors or microcontroller, 8-16-32 bits to account for their power and possibilities).

8.1.5 PIM of the Inclined Elevator System: Creating the Generic Architectures Package

The Generic Architectures package is a second layer or level of reuse concept. It allows developers to build over time structural concepts based on classes declared in previous Generic Classes package. They can be new classes

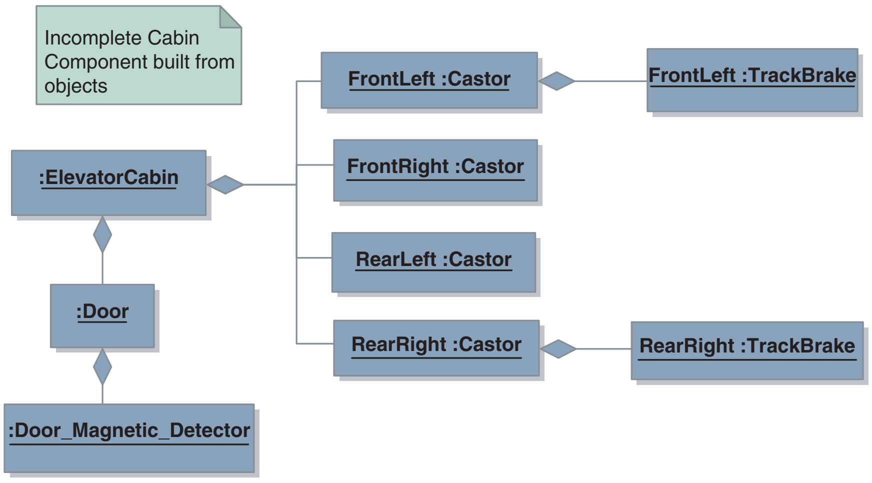


Fig. 8.1.5.1 For one shot deal project with small chance of recurrence (small scale reuse), components can be built directly from objects instantiated from the Generic Classes package. This diagram is incomplete and is targeted to show small-scale reuse process

obtained by aggregating/composing classes together, objects derived from classes and linked together into a generic component. Interfaces can be defined for components.

In UML 2, a component can be now considered as a class and thus can be instantiated. Therefore, there exist two approaches for handling real-time systems, one, with small-scale reuse and another with large-scale reuse. The first one consists of building real objects/components just after the Generic Classes package. Objects/components built under their instantiated form are "object based components" by opposition to "class based components." For instance, to build a fully equipped cabin ready to be used in the elevator, an object cabin will be instantiated, attached to an object door, an abject magnetic sensor, four objects castors, etc. If accidentally, we need duplicated components in the object form, they can be replicated. There is no inconvenience associated with this method of developing projects except that we cannot reuse microarchitectures.

Figure 8.1.5.1 shows this attitude that can be adopted for one shot deal projects with very small chance of recurrence.

For large-scale reuse, microarchitectures can now be built from classes present in Generic Classes package, and stored in Generic Architectures package (Fig. 8.1.5.2). In this package, we store composite classes and components under their "class forms," not "object forms." 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

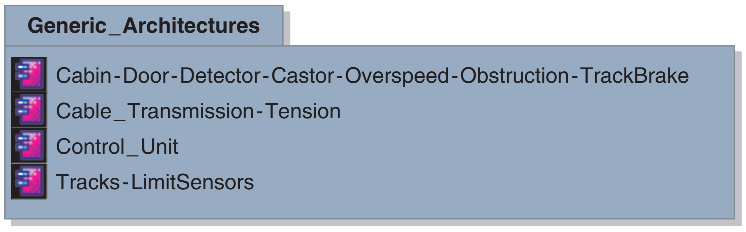

Generic_Architectures
Cabin-Door-Detector-Castor-Overspeed-Obstruction-TrackBrake
Cable_Transmission-Tension
Control_Unit
Tracks-LimitSensors

Fig. 8.1.5.2 Four components are defined in Generic Architectures package

developed. We illustrate its contents by creating some generic microarchitectural constructs for the hypothetic LiftTram Company. This process considers components as classes. At the next step only, components will be instantiated and used to build PIM prototypes.

Components defined at this stage are "class based components" (Fig. 8.1.5.3). They are built from classes. They are composite (sign "o-o" at the right low corner) and multidisciplinary as they mix inside a same entity several classes belonging to several domains. Interfaces can be defined for each component, but it is not always mandatory to do so (only when we want to afford a "black box" view of a component). In our case, components are transparent and in their "white box form." Links between components can go through the frontier of a component and reach internal classes. The constitution into components allows us to handle complexity only, not necessarily to encapsulate them all of the time.

The Drum Hoist contains 2–4 cables. Each pair of cable is controlled by a tension detector. We can build other components with other configurations then name them differently, for instance assemblies CTT1, CTT2, etc. to account for various versions of the Cable-Transmission-Tension assembly (Fig. 8.1.5.4).

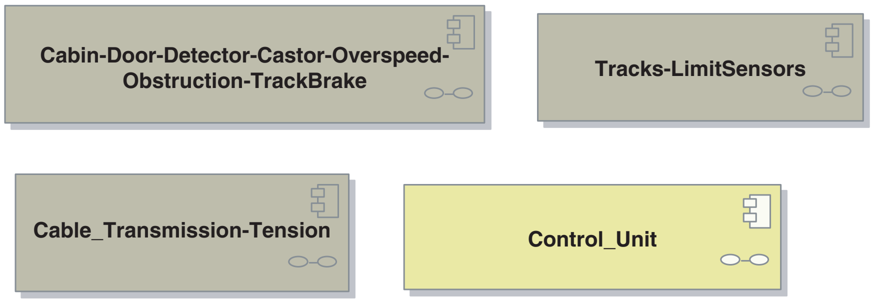

Cabin-Door-Detector-Castor-Overspeed-Obstruction-TrackBrake
Tracks-LimitSensors
Cable_Transmission-Tension
Control_Unit

Fig. 8.1.5.3 Generic Architectures package contains the definition of four composite components

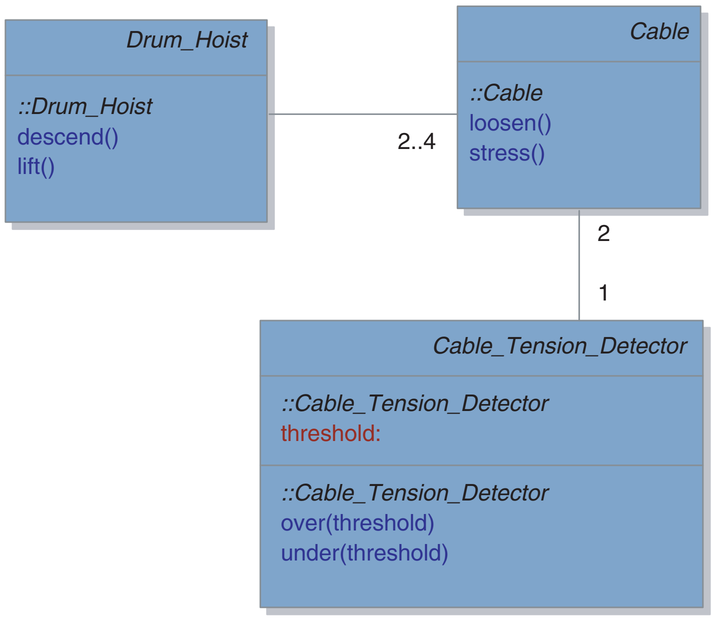


Fig. 8.1.5.4 Composition of the Cable-Transmission-Tension component

The Track-LimitSensors component (Fig. 8.1.5.5) packs inside a same set two tracks (Left, Right tracks facing the slope) and each track is equipped with two limit sensors. This aspect shows that a component may contain classes or objects/components that are not connected together.

We have the choice of merging once the Castor class and then affecting a multiplicity 4 to this class as often done in database system. In Figure 8.1.5.6, the Castor class has been merged four times instead, to generate four different classes whose names come from their positions relative to the Cabin (we suppose

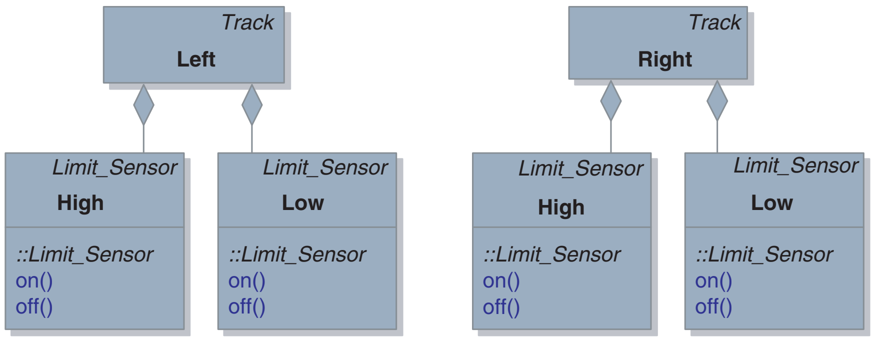


Fig. 8.1.5.5 Composition of the Track-Limit_Sensors component

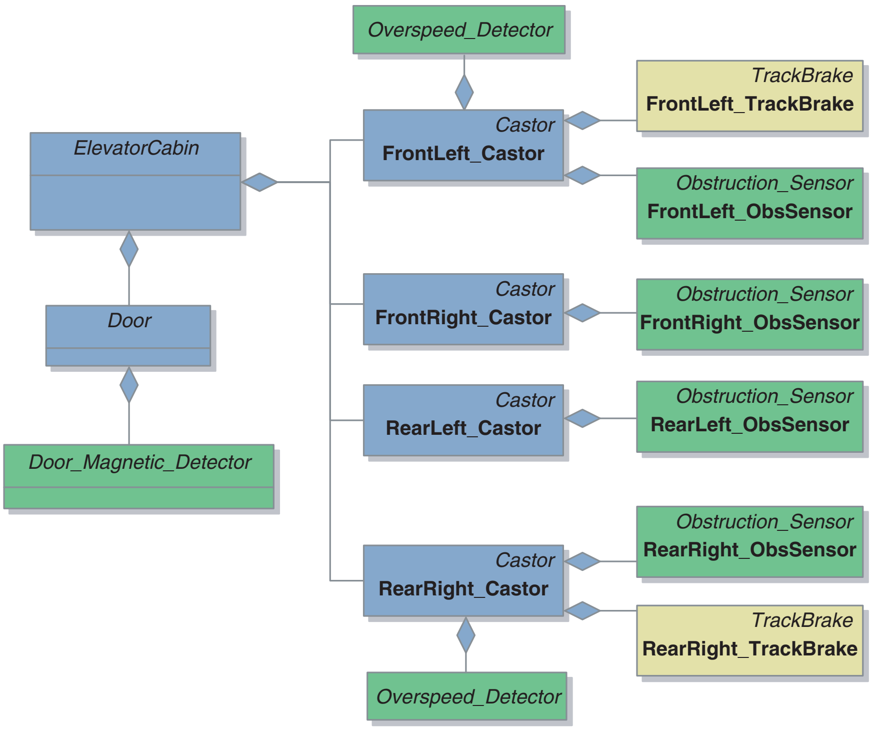


Fig. 8.1.5.6 Composition of the Cabin-Door-Detector-Castor-Overspeed-Obstruction-TrackBrake component

that we have defined a front side for the cabin, for instance the side facing the slope). In real-time system, using a multiplicity specification is not always appropriate as this process bypasses the identity of each individual object. In our case, as sensors and brakes must be mounted on specific castors, making four different classes is a better approach as this process shows visually how the system is really built.

In the solution proposed (other solutions is welcomed), brakes are present only on the front left and the rear right sides. Obstruction detectors are present in all track sides as tree branches may obstruct any side of the tracks in the up/down directions.

The aggregation/composition relationship is used with a restrictive interpretation "each time we instantiate an elevator cabin, we must instantiate all of its aggregated components." This attitude shows all objects packed within the component. From the functional viewpoint, castor is mounted into the cabin and a track brake is attached selectively to a castor to implement protection scheme. If a developer chooses to represent the aggregation with a simple association, he must maintain only his proper inventory.

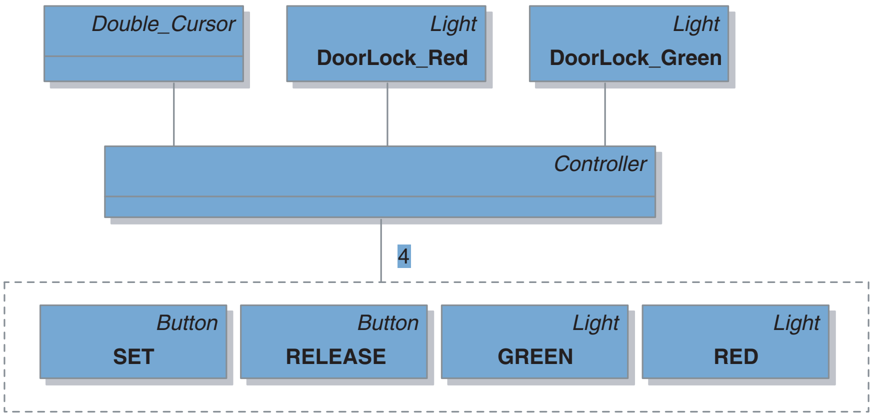


Fig. 8.1.5.7 Composition of the Control Unit component

The Control Unit component is shown with its accompanying elements. When connecting this unit in a system, connections will be drawn directly to the Controller. For instance, if the Controller must monitor the Door Lock Sensor, this signal goes through the frontier of the Control Unit component (Fig. 8.1.5.7) to join the controller directly. From the electronic interfacing viewpoint, possibly, we need more elements (parallel port, adaptation electronics, even analog–digital or digital–analog converter, etc.). These elements can be integrated in a modern microprocessor-based controller. These details are part of the PSM, not the PIM, unless algorithms to be studied involved these devices themselves. Therefore, in this logical model, we focus only on the control logic. If necessary, electronic interface classes (do not confuse the electronic interface with the "UML interface" concept) can be inserted to represent these elements. Ports, required interfaces, and provided interfaces are available embedded elements that can be added to any component to define interfacing structure (see Section 4.10).

8.1.6 PIM of the Inclined Elevator System: Instantiating a Working System

A working system can be built by instantiating objects from classes (belonging to Generic Classes package), object-based components from class-based components (belonging to Generic Architectures package) and objects directly from Application Specific Classes/Components package. The last package (Fig. 8.1.6.2) is needed because some information classes are specific to the current application, cannot be reused in any context, and must be created on the flight to support the application. Figure 8.1.6.1 explains the building process.

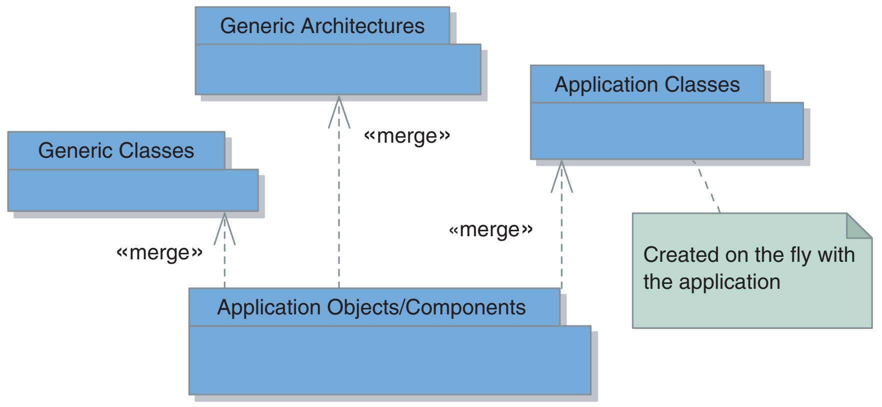
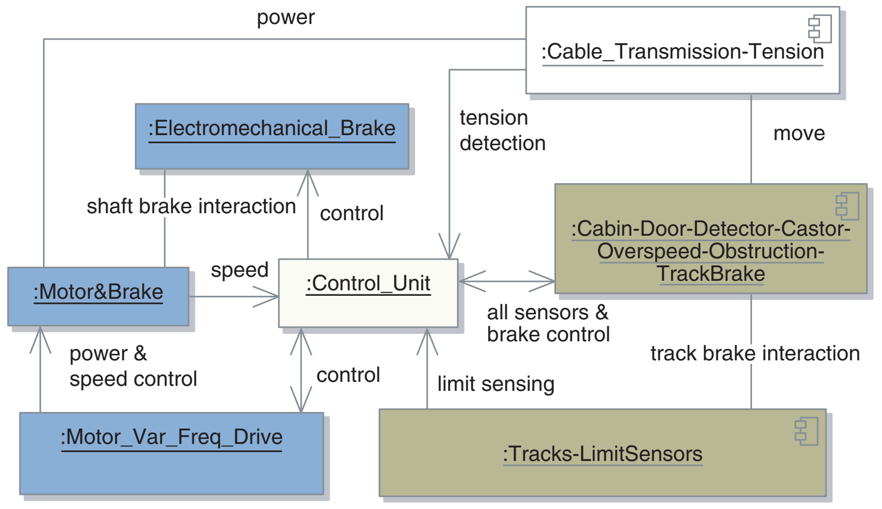


Fig. 8.1.6.1 The Package Application Objects/Components contains the PIM built from merging classes and components from other packages. Only Application Classes Package is not reusable



Final System composed of Objects and Components instantiated from Reusable Packages at various levels

Fig. 8.1.6.2 The final system is composed of objects/components instantiated from reusable Generic Classes Package, reusable Generic Architecture Package. Links drawn between objects/components are high level identification. They show potential communication between objects/components

8.1.7 PIM of the Inclined Elevator System: Behavioral Study of the Control Unit

Each object or component inside a system has a behavior. As stated in the theoretical part, the behavior of simple and passive objects/components is trivial and so does not need any dynamical study. The behavior of the complex and/or insufficiently decomposed objects/components is difficult (even impossible) to reach. The presence of a study does not necessarily mean that it is a valid and secure design. So, the key of success for obtaining complete, reliable, and valid design passes through a thorough decomposition and identification of all elementary and nonelementary operations of objects/components (see Sections 6.5.7 and 6.5.8)

In our case, most objects of the system are passive, mechanical, or electromechanical objects, so their behaviors are trivial. For instance, a cable has two states, stressed or loosened; the door has two states, opened or closed; the Motor can rotate clockwise or counterclockwise, etc. Some components have a complex behavior, e.g. the VFD. As they are black box components, their interfaces are fully defined so there must be no problem if we observe scrupulously operating modes indicated in the instruction manual. We cannot change or modify anything when dealing with black box components.

The only device that needs a full dynamic study would be the control software implemented in the Controller. The first step is establishing a complete inventory of all inputs and outputs, possible values they take in their respective domains, as well as thresholds and actions that must be performed around those thresholds. A link in an object diagram can pack inside a unique connection a set of inputs and outputs. They must be separated individually to reach a complete investigation. A new object diagram can be drawn to identify visually all the inputs/outputs of the Controller. Table 8.1.7.1 lists all the variables of the Controller and Figure 8.1.7.1 shows the corresponding object diagram with all objects that interact with the controller. We stop the interaction at the first neighbors of the Controller.

Normally, we can connect directly the Double_Cursor to the VFD with some signal adjustment. We have chosen a full software solution that shields the operator from the real command injected to the system. This concept relies heavily upon the “intelligence” of the Controller. We do not pretend that the solution is optimal for the elevator domain. The purpose of this case study is using an imaginary system that allows us to explain a design methodology. So, specialist advices for future releases of this book are welcomed.

This shielding effect is necessary in most complex real-time systems as human operators are not always reliable, particularly in emergency situations. For instance, normally, we expect that the operator adjusts the cursor to accelerate then decelerate manually when approaching the landing zones. If for some

Table 8.1.7.1 This table lists all objects/components constituting the first neighbors of the Controller and identifies the nature of communication channels exchanged with the Controller. Signals 10, 24, 25 are analog

N	*Variable*	*Hosting object or component*	*In/out*	*Comments*
Inputs and Outputs OUTSIDE the Control Unit				
1	EM_brake	*:Electromechanical_Brake*	Boolean Output	0: No brake; 1: Brake
2	slack_cable	*:Cable-Transmission-Tension*	Boolean Input	1: Slack Cable detected
3	door_closed	*:Door_Magnetic_Detector*	Boolean Input	1: Door closed 0: Door opened
4	Overspeed	:Overspeed_Detector	Boolean Input	1: Overspeed 0: Speed under threshold value
5	Obstruction	*:Obstruction_Sensor*	Boolean Input	1: At least one of the four obstruction sensors is on 0: No obstruction
6	Trackbrake	*:TrackBrake*	Boolean Output	1: Set trackbrakes on 0: Set trackbrakes off If the power is off, normally brakes is on by design for security
7	Highlimit	*High:Limit_Sensor*	Boolean Input	1: High limit redundant sensors reached
8	Lowlimit	*Low:Limit_Sensor*	Boolean Input	1: Low limit redundant sensors reached
9	Direction	*:Motor_Var_Freq_Drive*	Boolean Output	0: Forward 1: Reverse
10	ramp_speed		Output Value	Set ramp speed (valid for two directions)
11	stop_decelerate		Boolean Output	1: Stop with decelerated ramp
12	Surge		Boolean Input	1: Current surge detected
13	power_on_off		Boolean Output	1: Power on (Motor) 0: Power off (Motor)
Inputs and Outputs INSIDE the Control Unit				
14	slack_light	*Slack:Light*	Boolean Output	1: Red 0: Green
15	slack_set	*Slack_Set:* Button	Boolean Input	1: Test slack brake security
16	slack_ release	*Slack_Release:Button*	Boolean Input	1: Release slack brake

(*Cont.*)

Table 8.1.7.1 (*Continued*)

N	*Variable*	*Hosting object or component*	*In/out*	*Comments*
17	OV_light	*OV:Light*	Boolean Output	1: Red 0: Green
18	OV_set	*OV_Set:* Button	Boolean Input	1: Test overspeed brake security
19	OV_Release	*OV_Release:* Button	Boolean Input	1: Release overspeed brake
20	TJ_light	*TJ:Light*	Boolean Output	1: Red 0: Green
21	TJ_set	*TJ_Set: Button*	Boolean Input	1: Test track jam brake security
22	TJ_Release	*TJ_Release: Button*	Boolean Input	1: Release track jam brake
23	DoorLock_Light	*DoorLock: Light*	Boolean Output	1: Red (door unlocked) 0: Green (door locked)
24	UP_value	*:Double_Cursor*	*Input Value*	Set UP speed
25	DN_Value		*Input Value*	Set DOWN speed
26	neutral_zone		Boolean Input	1: In Neutral Zone 0: In UP or DOWN zone

reasons, he forgets to decelerate manually when arriving, the limit sensors take over the landing operation (decelerate then apply the EM brake) even if the cursor is still at the maximum position. The only operation the operator must do is to bring back the cursor towards the neutral zone before being able to start a new cycle in other direction. Other solution like a fully automatic operation with two buttons “UP” and “Down” starting fully programmed cycles with a cursor adjusting the maximum speed is also an interesting solution as extended exercises.

Another problem detected is the potential conflict not mentioned in the SRS document. We have three sets of commands, one in the cabin and each landing zone has also an identical Control Unit. What really happened when several operators try to adjust their cursors at the same time? This hardware problem with switch logic is not solved at this PIM phase and will be postponed to the PSM phase. Voluntarily, we have introduced this point to show the difference and the role of models when evolving from the first PIM to the last PSM.

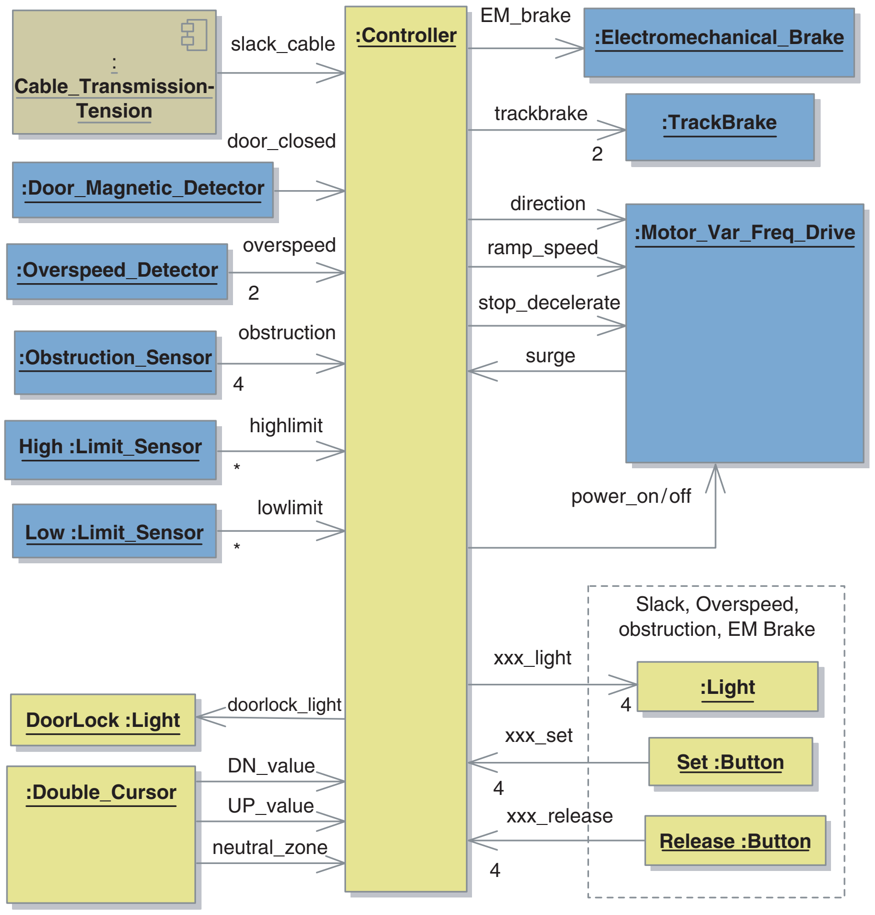
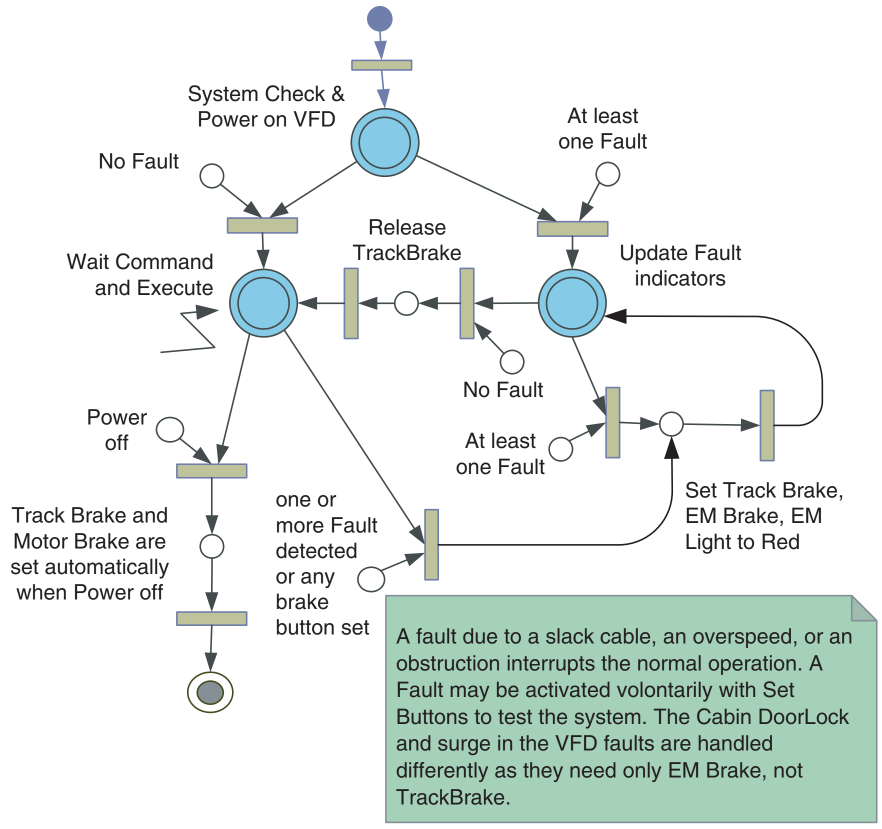


Fig. 8.1.7.1 Object diagram of the Inclined Elevator Controller with all inputs/outputs of its first neighbors

As a technological note, the Controller is able to read and issue both digital and analog signals. Microcontrollers or microprocessors boards with appropriate parallel ports, A/D and D/A converters currently support this feature.

Figure 8.1.7.1 shows a relative complex application. The best approach is top down with a full description of the natural sequence to identify in the first round high-level tasks. Progressively, tasks are decomposed to reach more elementary operations. The following figures algorithms of the controller studied with a SEN diagram. Those dynamic diagrams are first attempts and must be confirmed by more systematical tools like "images attribute methodology" exposed in Section 7.5.1.

The top-level SEN diagram of Figure 8.1.7.2 explains the overall software organization and represents a specific design of this elevator. Other designs are


System Check & Power on VFD
At least one Fault
No Fault
Release TrackBrake
Wait Command and Execute
Update Fault indicators
No Fault
Power off
At least one Fault
one or more Fault detected or any brake button set
Track Brake and Motor Brake are set automatically when Power off
Set Track Brake, EM Brake, EM Light to Red

A fault due to a slack cable, an overspeed, or an obstruction interrupts the normal operation. A Fault may be activated volontarily with Set Buttons to test the system. The Cabin DoorLock and surge in the VFD faults are handled differently as they need only EM Brake, not TrackBrake.

Fig. 8.1.7.2 When the power is on, the Controller undergoes a system check routine then powers the VFD. If faults are declared, the Controller requests that all faults must be cleared before enabling any movement

also possible. Our goal is to show how to model, draw hierarchical dynamic diagrams, represent interruptions, subdiagrams, not really to solve the elevator problem.

When entering a dynamic diagram, first locate the “black point” that represents the pseudoinitial state where interpretation starts. The “black point circumscribed with a circle” is the exit point representing the end of the current diagram. At the top level, this pseudofinal state marks the end of the software. At any other levels, this final state returns the control to the parent level. We can have more than one final state, in this case, they exit with different conditions. For instance, while exiting “System Check and Power on VFD” (Fig. 8.1.7.3), the subroutine exits with “No fault” or “at least one fault.” Composite and decomposable nodes are double circled and normally must be detailed elsewhere. We make use of two sizes of subroutines and the decomposition of “small size” subroutines are delayed to the PSM as they can be easily understood.

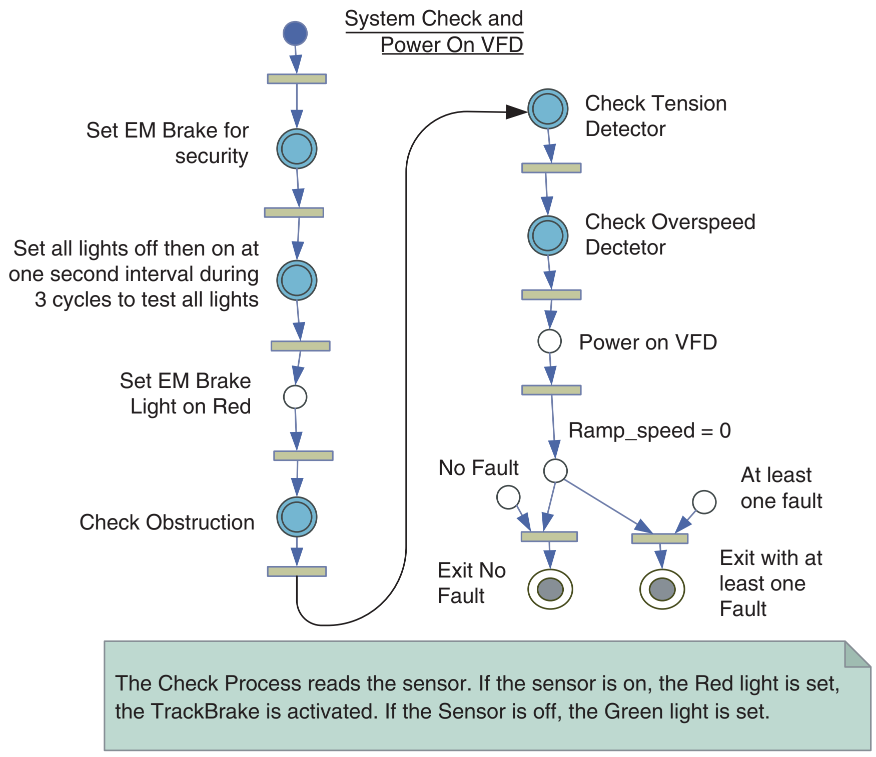

System Check and Power On VFD
Set EM Brake for security
Set all lights off then on at one second interval during 3 cycles to test all lights
Set EM Brake Light on Red
Check Obstruction
Check Tension Detector
Check Overspeed Dectetor
Power on VFD
Ramp_speed = 0
No Fault
At least one fault
Exit No Fault
Exit with at least one Fault
The Check Process reads the sensor. If the sensor is on, the Red light is set, the TrackBrake is activated. If the Sensor is off, the Green light is set.

Fig. 8.1.7.3 This routine details "System Check and Power on VFD" subroutine. It sets the main EM brake on, verifies track jam, slack cable and overspeed sensors, and set lights accordingly

Normally, by default, we write as a constraint at project level that all activities, elementary or composite (subroutine), will execute to their natural end. So, they keep or "drown" the token as long as needed (see Section 7.1 for rules). Interruptible activities are marked with lightning arrows (e.g. "Wait Command and Execute"). So when the power is off (the current can be switched off by other mechanical means and not with power on/off on the Control Unit), the Trackbrake and the fail–safe internal Motor Brake (not EM Brake) are by default on and they immobilize the cabin in this circumstance.

When the Controller is powered on, it sets the EM brake immediately to stop all movements to undergo a system check. At the end of the check, the VFD is powered on. At this point, the subroutine exits with two possible conditions "with at least one fault" or "no fault". If "no fault," the system is ready to read commands from operator and execute them. In the presence of faults, all faults must be cleared (details in Fig. 8.1.7.4) before returning to normal operation. Tracks must be cleared if jammed by obstructive objects, the cable must not

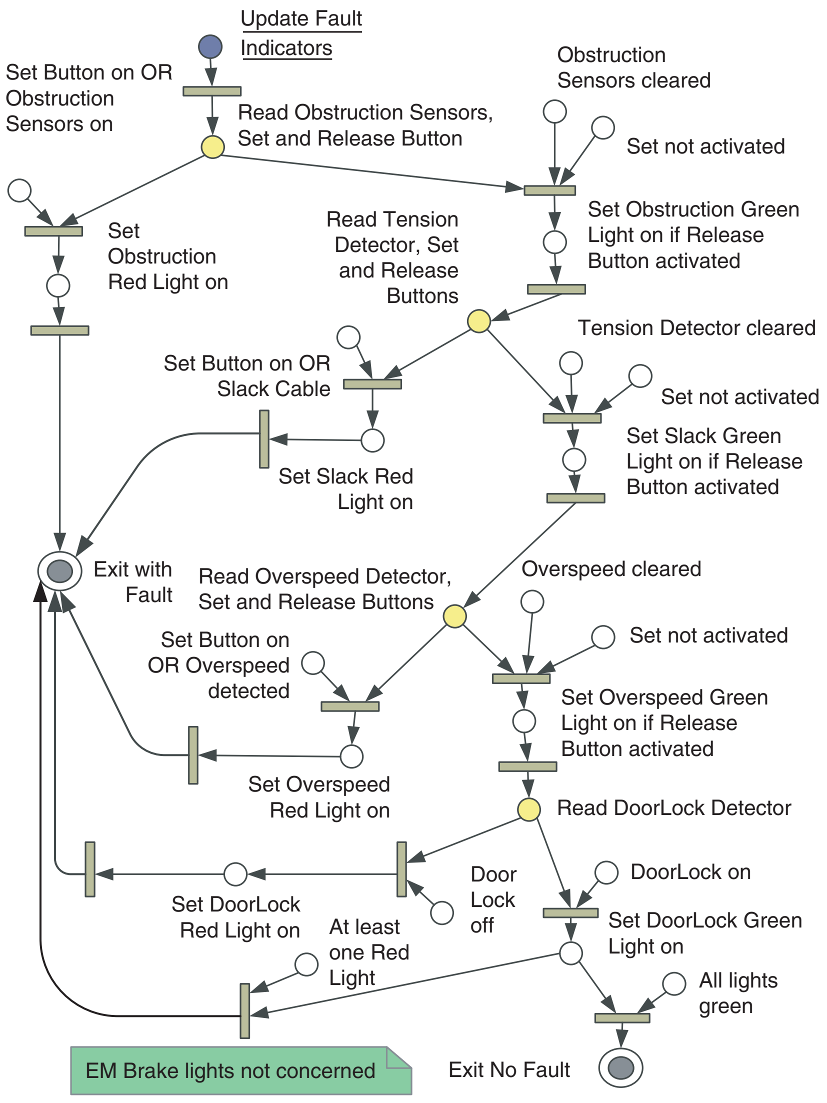

Update Fault Indicators
Set Button on OR Obstruction Sensors on
Read Obstruction Sensors, Set and Release Button
Obstruction Sensors cleared
Set not activated
Set Obstruction Red Light on
Read Tension Detector, Set and Release Buttons
Set Obstruction Green Light on if Release Button activated
Tension Detector cleared
Set Button on OR Slack Cable
Set not activated
Set Slack Green Light on if Release Button activated
Set Slack Red Light on
Exit with Fault
Read Overspeed Detector, Set and Release Buttons
Overspeed cleared
Set not activated
Set Button on OR Overspeed detected
Set Overspeed Green Light on if Release Button activated
Set Overspeed Red Light on
Read DoorLock Detector
Door Lock off
DoorLock on
Set DoorLock Red Light on
At least one Red Light
Set DoorLock Green Light on
All lights green
EM Brake lights not concerned
Exit No Fault

Fig. 8.1.7.4 The subroutine “Update Fault Indicators” reads all sensors and buttons then updates all signal lights (Lights are exclusive, Red or Green only)

be slack, the overspeed detector must be “under,” and the cabin door must be closed before being able to issue any command. At the end of the routine, the VFD is powered on by the Controller despite the presence of faults or not.

The way the subroutine “Update Fault Indicators” is designed, the system test must be performed by maintaining the Set buttons constantly at their “on” position. If the set button is released, the set is cleared immediately as there is no

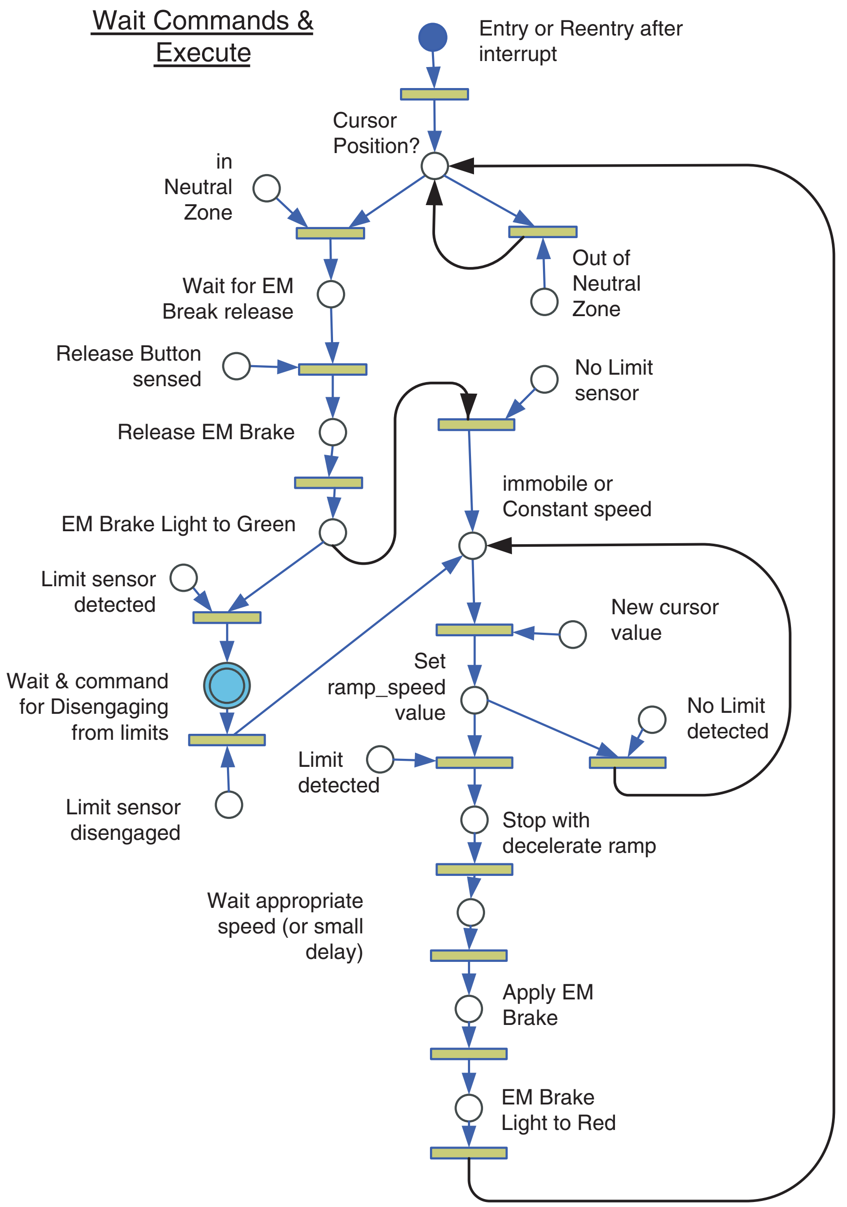


Fig. 8.1.7.5 The main subroutine supports the normal operation mode of the cabin

provision for latching the signal. Even if the sensor is in its normal state, the light does not pass to green immediately but requires a pressure on the corresponding release button, otherwise the fault condition remains. An instruction like "Set Slack Green Light on if Release Button activated" is not elementary from the programming viewpoint. But, at the design phase, we admit that it can be easily understood by any implementer.

In Figure 8.1.7.2, the "Wait command and Execute" subroutine can be interrupted anywhere, even immediately when entering the subroutine in presence of fault. Each fault arises an interrupt, triggers the trackbrake and the EM brake. So, all categories of brakes mentioned in the SRS document are designed with the combined action of two brakes: EM brake for normal operation of the cabin and EM brake combined with trackbrake for emergency situations.

The next subroutine of Figure 8.1.7.5 explains the normal operation of the cabin. When entering the "Wait command and Execute" subroutine, the EM brake avoids all cabin movement due to the incorrect position of the cursor. The trackbrake has been already released by clearing all exceptions conditions. Roughly, the operator must push on the release Motor Brake Button (EM Brake) to suppress this last brake. But this action can be realized only if the Double_Cursor is in its neutral zone.

When all brakes are cancelled, the operator can now operate the cabin by gliding the cursor in the UP and DOWN zones. At this time, it is possible that the redundant limit sensors are still present and detected by the controller. As these sensors stop the movement of the cabin when arriving at the landings, we must distinguish the disengaging procedure for quitting the landings from the brake produced when arriving at the landings; otherwise, if the limit sensors inhibit any movement, we will be not able to leave the landing zones, the batteries of limit sensors, constantly switched on.

When the cabin is moving with a constant speed, the controller continuously reads the cursor information and updates the ramp speed of the VFD at real time.

8.2 Emergency Service in a Hospital and Design of a Database Coupled with a Real-Time System

The following case is interesting as we simultaneously address two problems, real time and database, that are intimately related to each other. We are in the situation described in Section 6.7 in which objects must be *individually registered*, and *their states monitored at real time*. The model necessitates many kinds of attributes: *structural* for identifying the objects, *behavioral* for tracing its evolution in a real-time context, etc.. Real-time classes contain large number of real-time operations to describe medical operations, and state attributes to store patient states and, at the same time, the real-time system needs information in the database about patients and eventually needs to update them.

When working with such a system, the database must be built first, and then classes can be derived from the database in order to get all descriptive information. Dynamic attributes and operations are then added to make workable classes. As said in the theoretical part, when a supermarket stores 100 cans of maple syrup, it does not make sense to identify them individually. In a hospital, patients have multiple views. They are indistinguishable when we need make medical statistics but this "cans of maple syrup" view or "columns of numbers' view disappears when addressing to the patient that the practitioner must study medical information and find a good health-care plan for her or him. They must be individually monitored and treated with care and humanity. Canadian gun registry program or baggage handling service in an airport, are, for instance, problems of the same category. Owing to the limited format of this book, we cannot treat the example completely so we focus more on the database part and give very useful information to approach this double problem from the logical view. The PIM will stay valid even if at the next phase, the PSM calls for two different but communicating platforms.

8.2.1 Early Requirements

A hospital X wants to improve its emergency service and wishes to model its system to build a patient database and a decision-making system to classify patients arriving at emergency service in order to decide which patient must be examined first.

The hospital contains:

1. *Receptionists taking phone calls.*
2. *Ambulance men transporting patients to emergency or transferring them towards other destinations.*
3. *Patients who come by their own means, generally accompanied by members of their family.*
4. *Receptionists performing patient registering with an admission form. The mean time spent to register is about 2 min for a patient with an existing record and 8 min for a new patient. If a patient is unable to register, his or her family member can help for registering.*
5. *Medical assistants who verify patient records, collect, scan, and classify official certificates, letters of reference, medical prescriptions, radiographies, laboratory test results, etc. Their tasks can be performed anytime, while patients are waiting in the waiting room or later.*
6. *Chief nurses who examine patients and determine the urgency with which each patient should be examined by a practitioner. Their algorithm can be described roughly as follows. After taking the pulse and the tension, filling data in the database, they estimate the maximum waiting time allowed in minutes. The patient queue is therefore updated after each data entry (rescheduling). If a general practitioner is free before (or after)*

the expiration of the maximum delay, the patient at the top of the list will pass the examination immediately. The average duration of examination of a patient is about 12 min by the chief nurse.

7. *General practitioners are those who carry out diagnoses, prescribe drugs, radiography, blood test, or more specific tests. Medical data are collected at each step and are accessible via the medical database. Each medical examination lasts between 7 and 15 min. On the average, 11 min was retained as a mean value. Thereafter, the patient, on foot or on a stretcher, can be put under observation in a rest area between 30 min and 8 h. In this case, the general practitioner puts the patient back in the waiting queue exactly as if the later comes from the reception area. In the rest area, other practitioners can reexamine the patient for the 2nd time or the 3rd time, etc. Subsequent examinations last around 6 min. After each examination, the practitioner must write down what is the maximum waiting time allowed before the next examination (same procedure executed by chief nurses).*

8. *Nurses , for the period of observation in the rest area, take measurements on the patient like blood pressure, pulse, pain scale every 60 min, give drugs to patients, check/install medical apparatus, and update the database. This takes about 10 min for a nurse to realize this act for each visit. At night time, nurses can take over tasks of receptionists and take phone calls if needed. A chief nurse may execute all medical or administrative tasks of a nurse or of a medical assistant if necessary.*

9. *Surgeons.*

10. *Other conditions are, no patient can stay in the rest area more than 8 h. If there is an extension beyond this limit, practitioners must choose between prescribing a hospitalization in the current hospital (if beds or individual rooms are still available), transferring the patient to another hospital with a preliminary availability check, sending the patient in surgery, or sending the patient to home with a drug prescription. The practitioner can decide on a hospitalization immediately when examining the patient for the 1st time or after the 2nd, 3rd examination, etc. when he has seen laboratory or test results, or simply just by seeing that a patient is getting worse. In those cases, the patient leaves the emergency queue.*

The emergency staff counts 10 general practitioners, 3 surgeons in the daytime (7a.m. to 8 p.m.). In the evening, two general practitioners are always present and a surgeon works on call only. Initial hypotheses state that beds, nurses, chief nurses, receptionists, laboratory personnel, etc. are largely sufficient (technically speaking, they are "infinite resources"). The hospital X wants to know *what is the maximum number of patients that can be admitted in a day within this medical staff, what is the maximum arrival flow manageable according to the time staying in the rest area, and finally what are important dynamic parameters of the whole process?*

8.2.2 Use Case Diagrams for System Requirements Specifications Document

Hereafter, we bypass the text analysis phase to expose directly a set of UC diagrams that normally structures concepts revealed during the text analysis phase. This task affords a first attempt of structuring the description. Concepts in Requirements Analysis phase are exposed as those perceived by the hospital Client, but, at the design phase, they can be represented in a completely different manner. As said in the theoretical part, even if the use case analysis is not very accurate, it has several benefits:

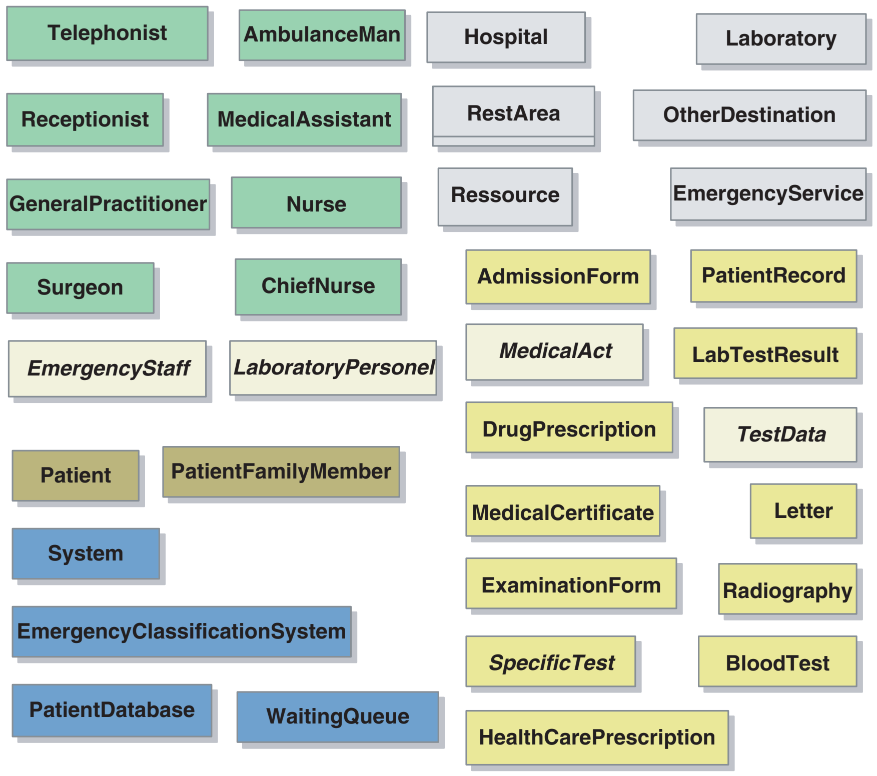
1. It helps conceptualizing the problem
2. It forces to read “correctly” the project to be developed
3. It classify ideas
4. It identifies main actors and main activities
5. It organizes activities into a network of dependent activities. By laying a network of “include” and “extend” relationships, it first apprehends dynamic structures
6. It highlights goals to be satisfied
7. It puts all requirements under a graphical form
8. It tends to structure the system even if this structure is not very rigorous.

All those benefits will prepare correctly the design phase.

8.2.3 Designing the Patient Database Coupled with a Real-Time Emergency System

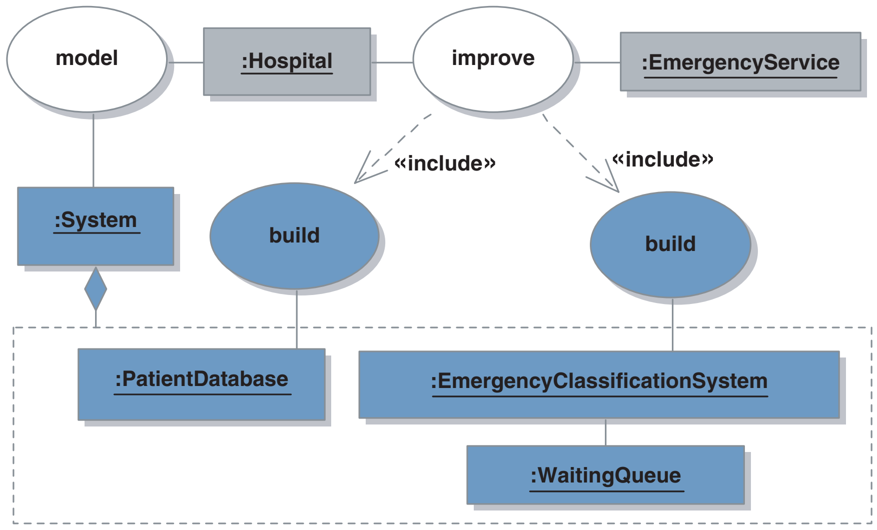
The challenge is to deploy reuse principle early in the PIM, despite what will be decided at the PSM level. As said before, there is two systems intermixed together in the problem of this hospital.

First, we must construct a database to store information about patients, record all medical acts performed by medical staff at every step, and store multimedia data (radiography images, letters, various documents, etc). Firstly, this database is of a special type in which all records must be identified individually. Secondly,

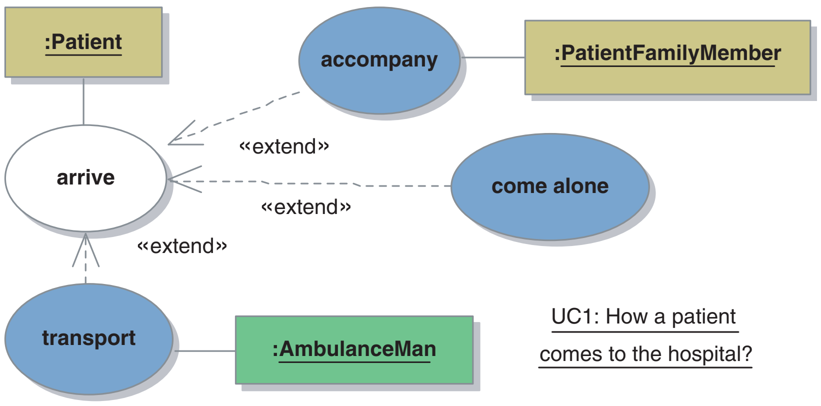


Classes of the Hospital for use with Requirements Engineering Only

Fig. 8.2.2.1 Classes of the Hospital with the Emergency Classification Problem. They are used to instantiate objects for specification purpose only, not for design

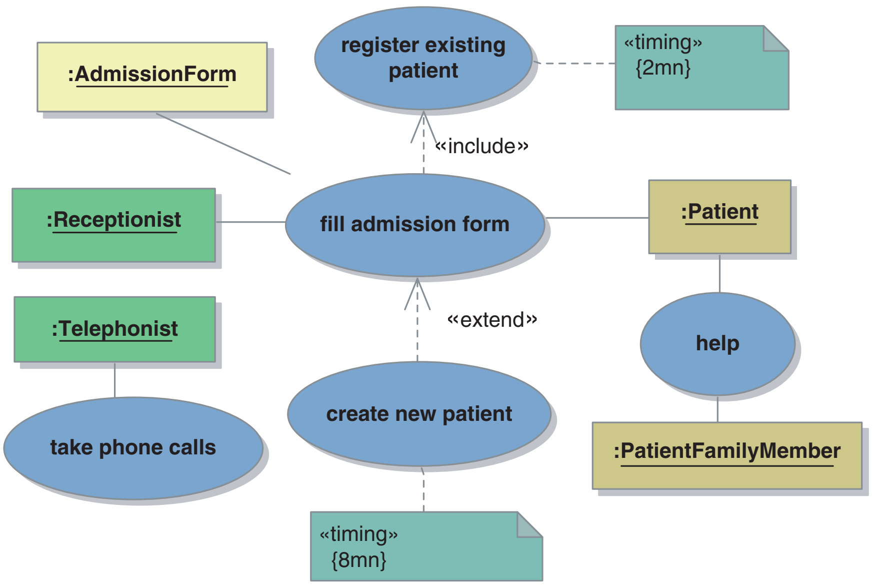


UC0: Problems to be solved

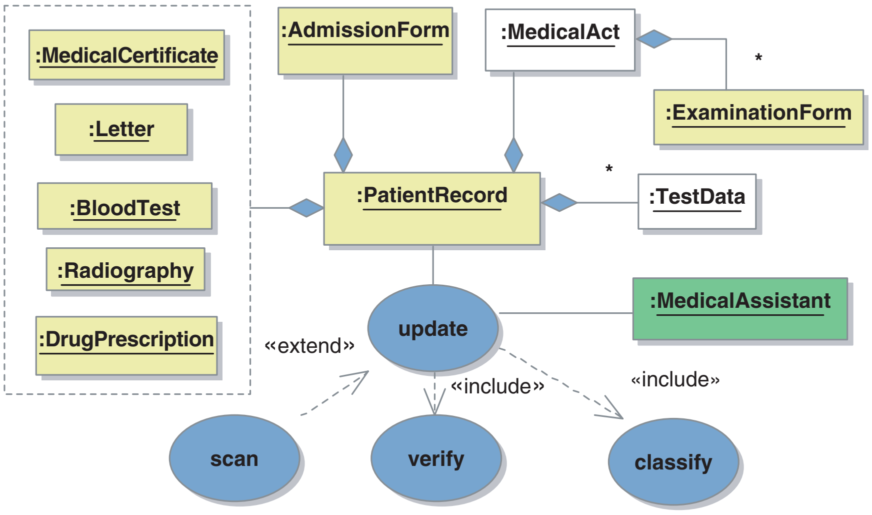


the emergency system shares some structural data of the database. The decision of the general practitioner or the chief nurse about the maximum waiting time is dependent upon the current medical state of the patient that in turn, depends on current medical treatment, and patient history (for instance, if the patient has a heart attack in the past, it is likely that this patient will receive a high priority in the waiting queue).

At the limit, we can separate the hospital problem into two completely unconnected projects: the management of the emergency system and the

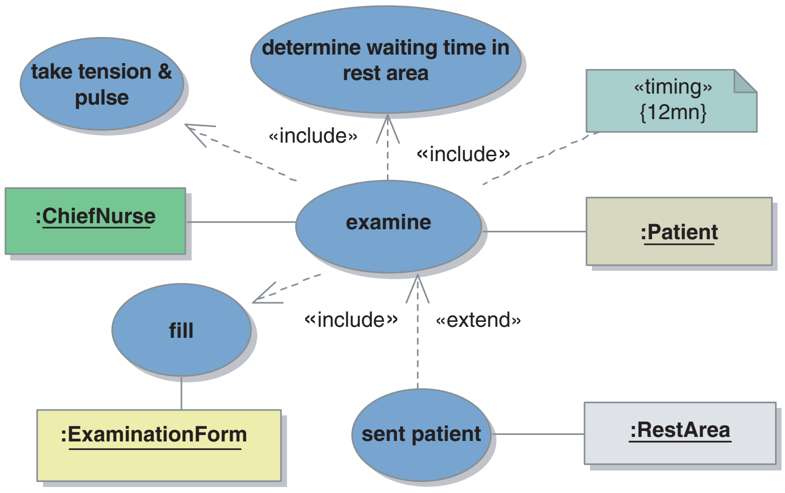


UC2: Reception & Patient Registering

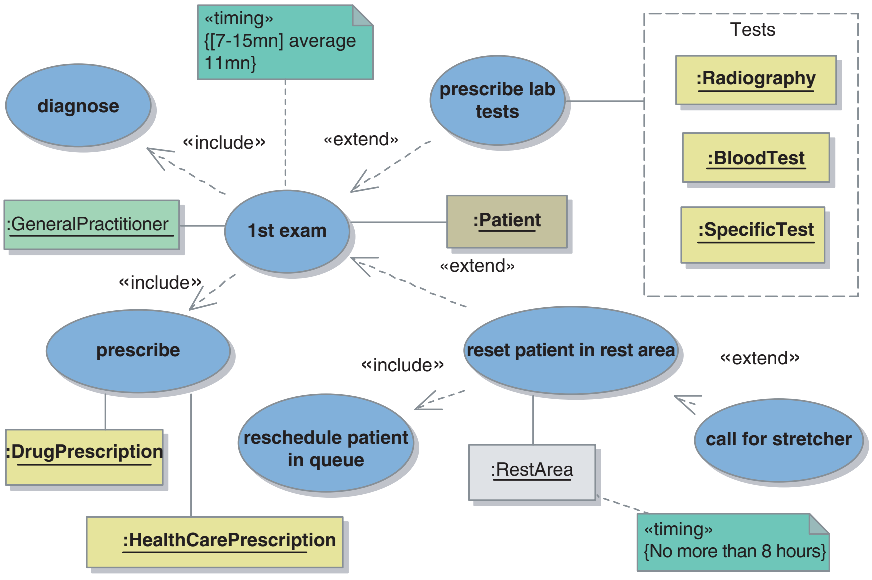


UC3: Tasks of a Medical Assistant

administrative database. In this case, we have very few information that can explain why a given patient has stayed, for instance, 5 h in the rest area, has been examined many times by the medical staff. To corroborate facts, a person must open the two applications at the same time and display information in such a way that relationships become explicit and help him in inferring new information. As this condition is very restrictive, it would be more interesting to connect systems together if, naturally, they are interrelated. Nowadays, even in

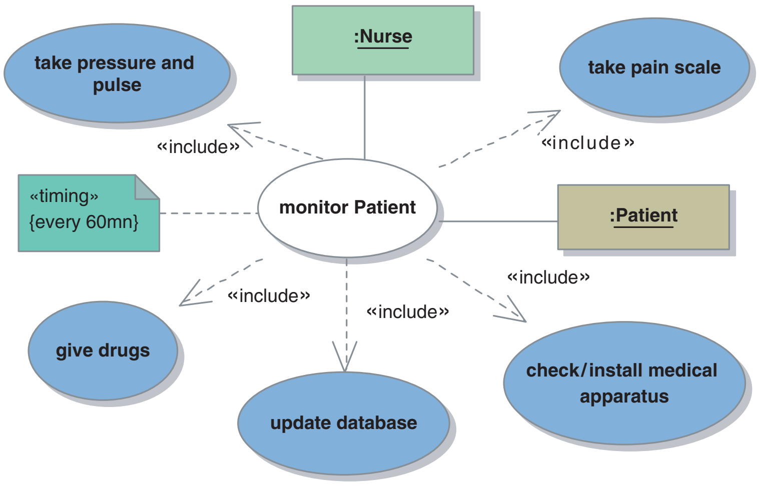


UC4: Tasks of a Chief Nurse

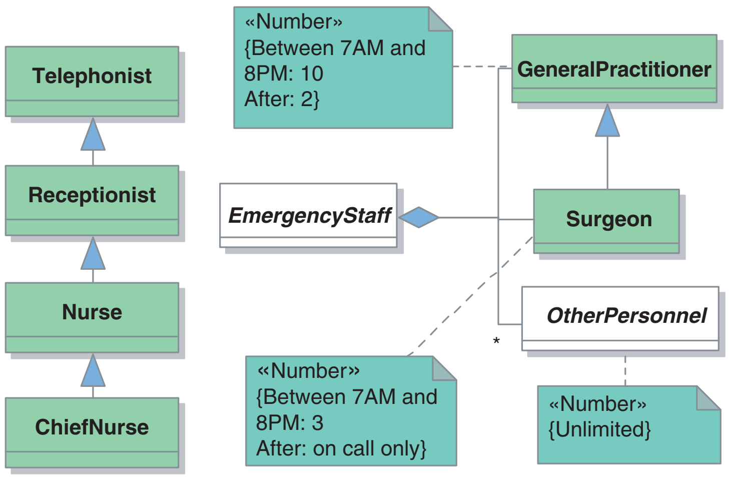


UC5: Tasks of a General Practitioner

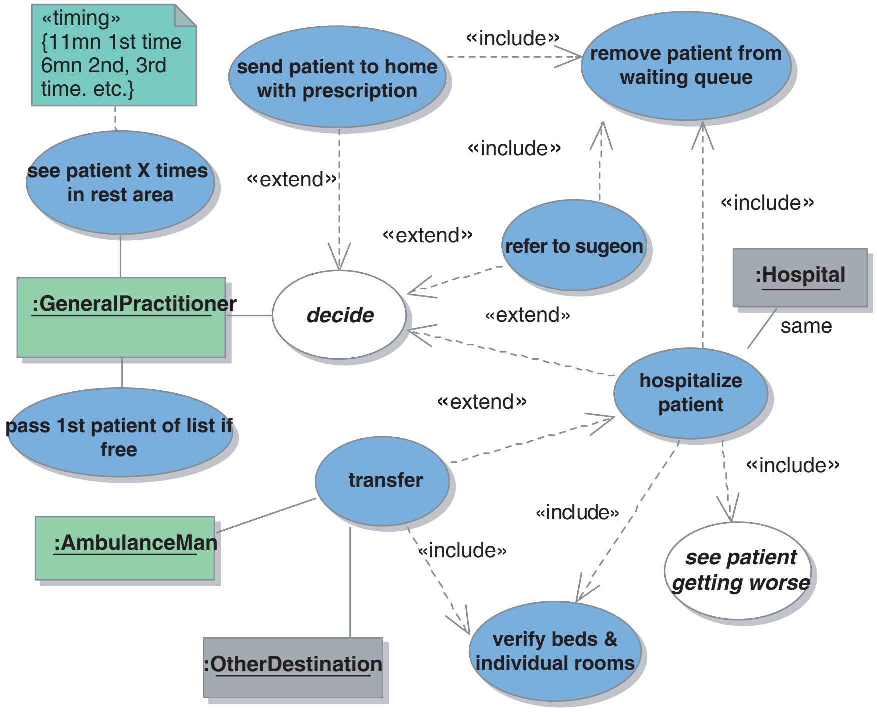
the presence of many independent platforms, from any software environment, we can connect to any database, and gain access to its data. The problem is administrative, not technological. Some technical difficulties still remain when accessing huge amount of multimedia data through controlled channels. From a logical view, a large amount of real-time classes of this hospital are the same



UC6: Tasks of Nurses



UC7: Staff & Role Inheritance


«timing» {11mn 1st time 6mn 2nd, 3rd time. etc.}
send patient to home with prescription
«include»
remove patient from waiting queue
see patient X times in rest area
«extend»
«include»
«include»
«extend»
refer to sugeon
:Hospital
:GeneralPractitioner
decide
«extend»
same
hospitalize patient
pass 1st patient of list if free
«extend»
transfer
«include»
:AmbulanceMan
«include»
«include»
see patient getting worse
verify beds & individual rooms
:OtherDestination

UC8: Emergency Queue Management

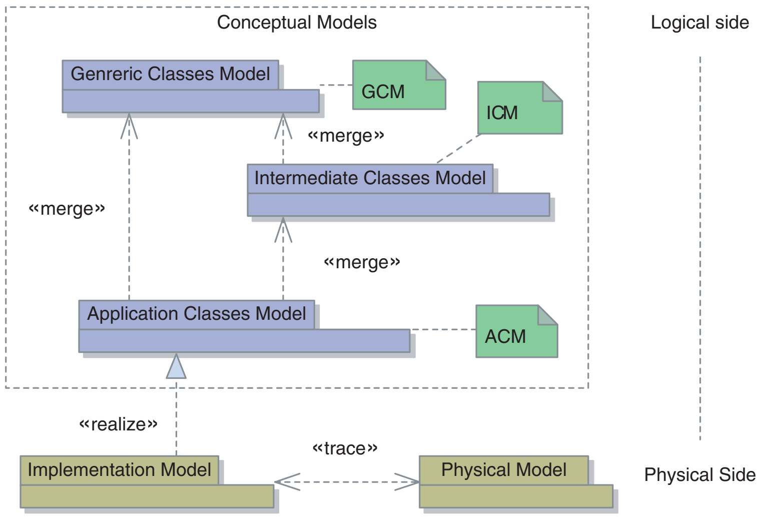
Fig. 8.2.2.2 UC0–UC8 specifying the hospital emergency problem

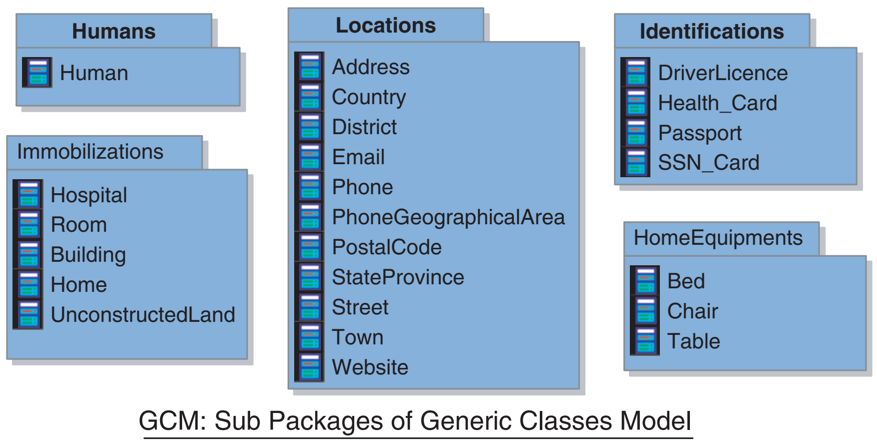
classes found in the database so real-time classes can simply be derived from database classes then enriched with operations and state variables. Even if the PSM is based on two different platforms, the PIM model must reflect this reality. Classes that are not concerned by this derivation are generally "information classes" (they are tied to an application and are not generally reusable). Information classes are created to support particularities of an application. For instance, a form that contains synthetic information about a patient is an information class (forms requested are different among hospitals). Transient query classes are mostly information classes.

For designing the database system, Figure 6.6.3.1 (reproduced hereafter) has already proposed a method for organizing data packages in a reuse context. This method divided the project into three packages called models. From GCM to the Physical Model, we pass through GCM, ICM, and ACM that 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.

Figures 8.2.3.1 to 8.2.3.3 show packages and classes defined in GCM, ICM, and ACM packages with their respective comments. The GCM is reusable in any context. ICM is reusable in any hospital environment on the world, and ACM is specific to this specific hospital X.

Humans, Locations, Immobilizations, Identifications were already defined in Chapter 6. We have enriched Immobilizations with a generic Room description and added a HomeEquipments package to create a Bed class to be used later in the ICM package to create a HospitalBed class. Chair and Table classes are unnecessary in this context but are displayed to illustrate the contents of this new package.





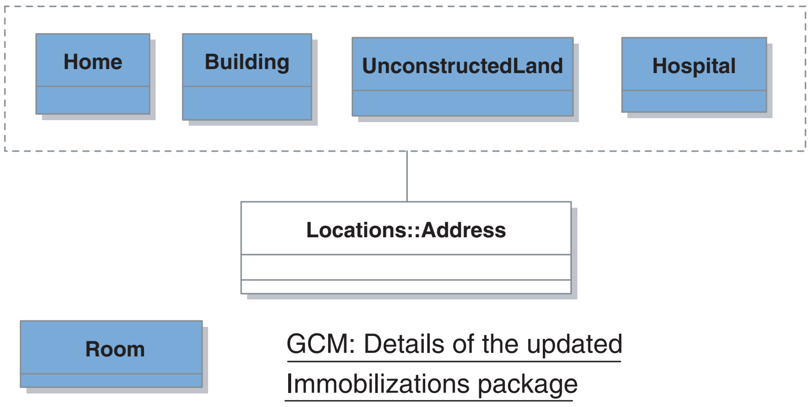
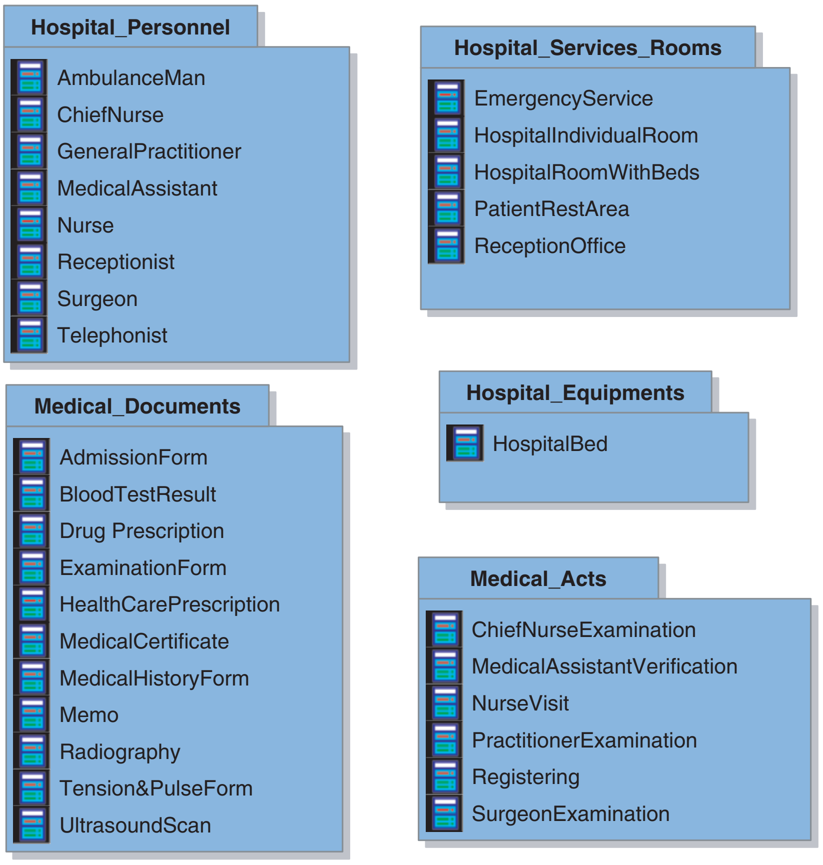


Fig. 8.2.3.1 Contents of the Generic Classes Model (GCM). Figures of this GCM suite must be completed with Figure 6.6.4.1 of Chapter 6

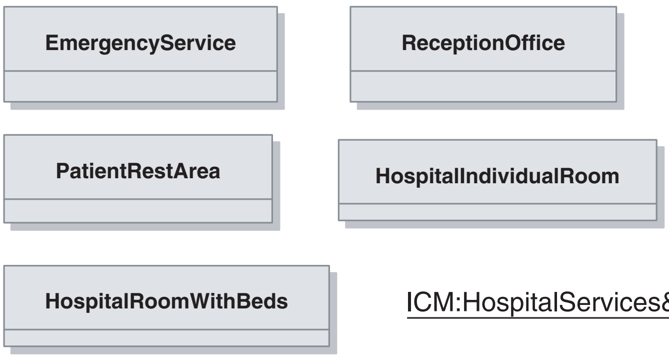
The immobilizations package is the same as that shown in Figure 6.6.4.1. A Room does not bear any address so it is created outside the association. Contents of other packages are already shown in Figure 6.6.4.1. We illustrate therefore the surprising fact that a GCM created in a completely different context can be reused in the context of this Hospital. Practically, we have exported the design of Figure 6.6.4.1 to an XMI format (defined in UML) and reimported this format inside this Hospital project.

It would be interesting to note that we have not used the title "GCM package of the Hospital." Owing to its high potential of reusability, the same package GCM can be redeployed "as is" for modeling a company, a university, or a governmental organization.

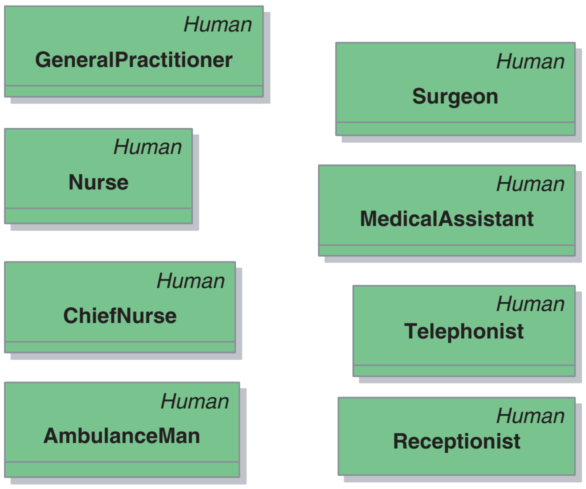
The next ICM package will be a step closer towards solving the Hospital problem. There can be many ICMs before reaching the final ACM. The number of ICMs is dependent upon the complexity of the final ACM itself and how



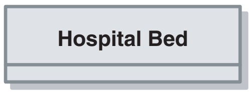
ICM: Top Level package



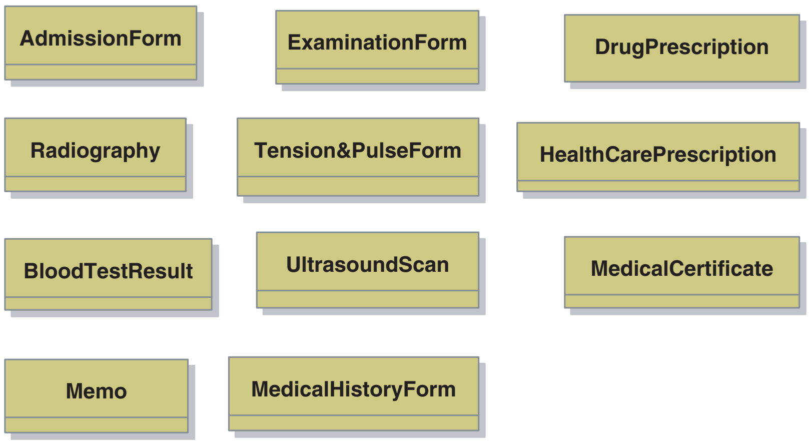
ICM:HospitalServices&Rooms



ICM: Hospital Personnel



ICM: Hospital Equipments



ICM: Medical Documents

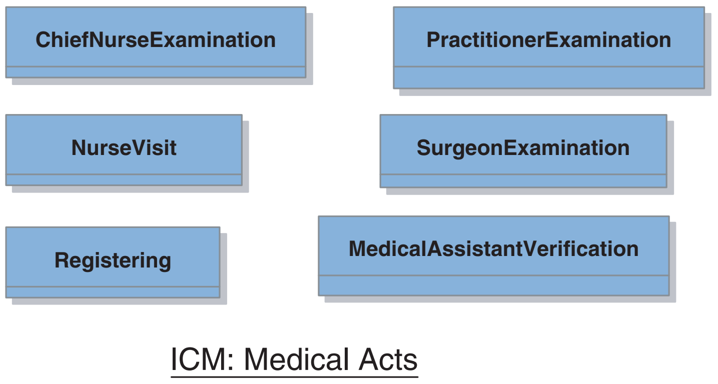
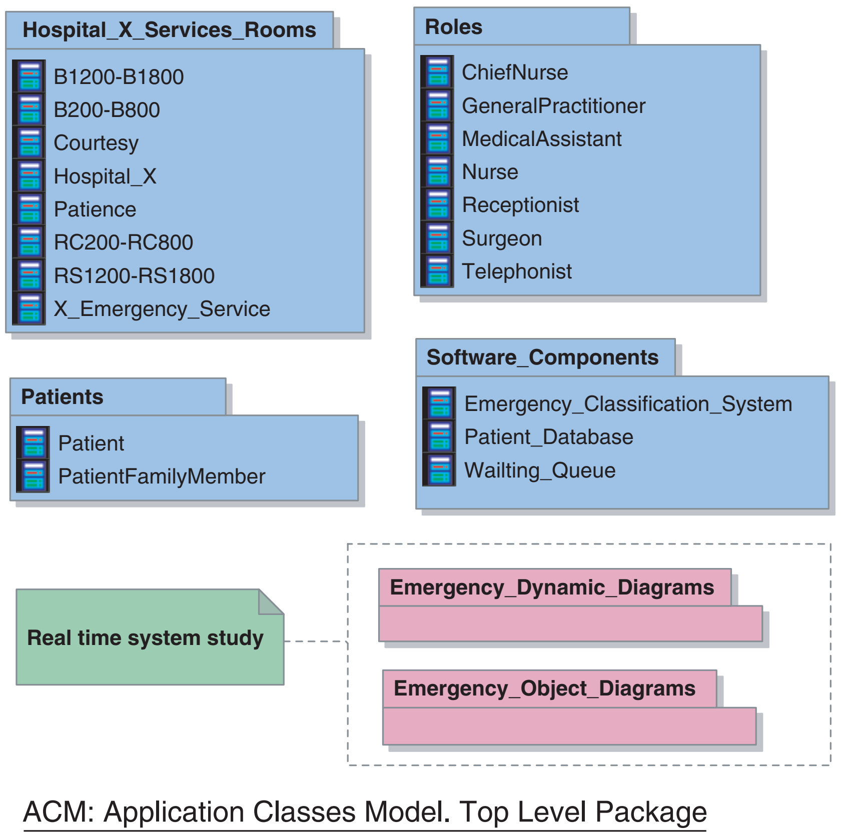
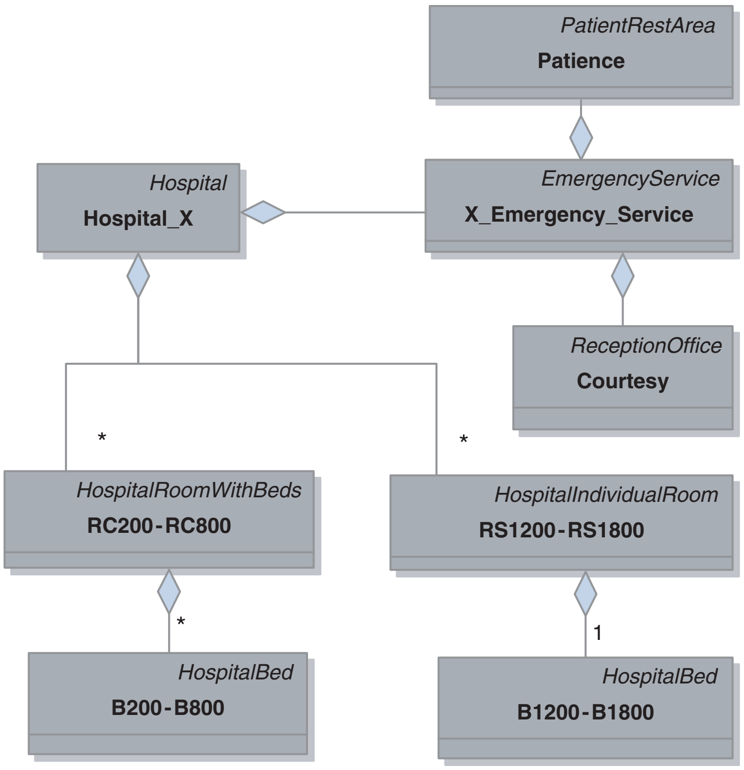
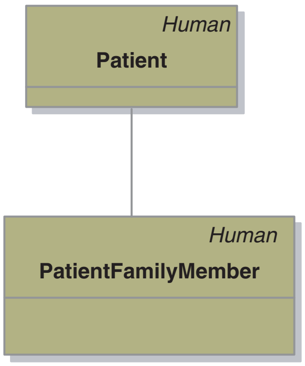


Fig. 8.2.3.2 Contents of the ICM package that contains classes belonging to the Health Care System





ACM:This package shows how the Emergency Service of the Hospital X is located inside this Hospital.



ACM: Patients and their family members

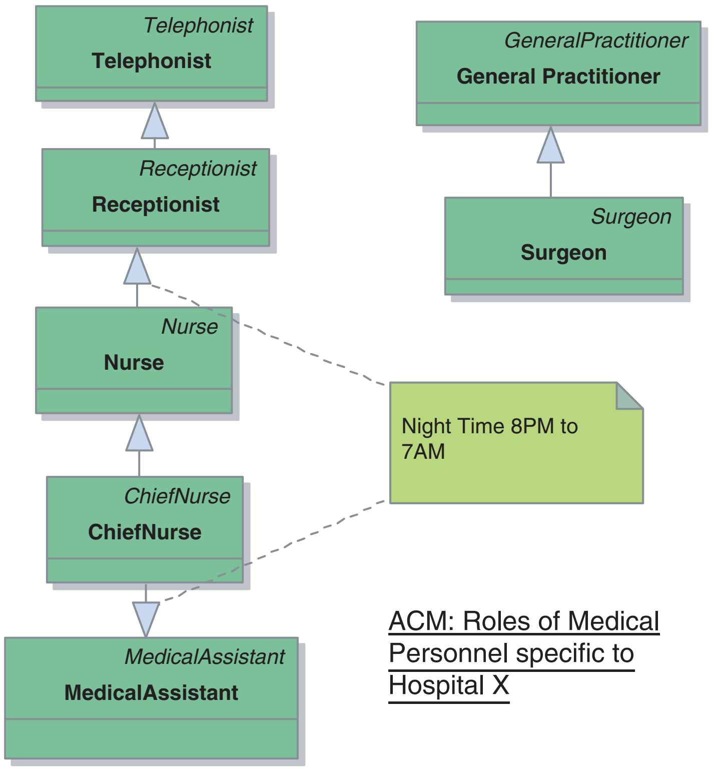


Fig. 8.2.3.3 Contents of the ACM (Application Classes Model) package. This package relates all relationships specific to the Hospital X

far concepts in the ACM are, relative to elementary concepts of the first GCM. In our case, we have created five packages: Hospital_Services_Rooms, Hospital_Equipments, Hospital_Personnel, Medical_Acts, and Medical_Documents to support the Hospital application. But, we anticipate that all classes created in the ICM can be reused in Health Care domain. Classes are not still connected together through relationships for not to bind those into a rigid structure that tends to destroy the reusability.

The subpackages "Dynamic Diagrams," "Object diagrams," and Software Components" can now be filled with diagrams similar to what we have developed for a real-time system and the Emergency problem can now be approached with all classes ready for the new derivation. If we take, for instance, a class like Surgeon, it derives from a generic Surgeon class in the ICM that packs operations very generic to a Surgeon all over the world. In the ACM packages, we put differential operations allowed in the context of the Hospital X, in country Y. The Surgeon class in ICM derives from Human class in GCM, so as a human; a surgeon will have all regular attributes common to a patient or any other humans.

With this example, we have demonstrated that databases and real-time applications can coexist in a same continuum and they are closely related in a PIM. The uniform methodology shows a high degree of reuse and creates models that defy space and time. New applications will be built from now on a common knowledge base that spares us so much energy and time. Each new problem instantiates new scenarios with new operations (occasionally new classes) but we have never to reinvent the wheel again.