

Designing Real-Time Systems with UML-Part II

By Bruce Powel Douglass

In this second in a series of three articles on how the Unified Modeling Language can be used to develop real-time and embedded systems, the author shows how the UML applies to a real-time system development problem: an anesthesia patient ventilator.

In part II of this series on the Unified Modeling Language, We shall employ many of the important features of the UML, including use cases, scenarios, class and object models, and finite state machines. The notational elements we'll use are the Use Case, Sequence and Class diagrams, and the Statechart.

The Problem

Breathing is important for two primary reasons.¹ First of all, it is the means by which oxygen required for metabolic processes is acquired (250ml/min O₂ for your standard-issue 70kg adult). Secondly, it is the primary means for the removal of carbon dioxide, a waste product of that very same metabolic process (200ml/min for the same hypothetical adult). In most deep-anesthesia surgical procedures, the patient is anesthetized and paralyzed, which facilitates the surgery but raises an issue of some concern to the patient. A paralyzed patient cannot breathe without assistance. Some means must be provided to ventilate the patient's lungs.^{2,3}

Such a machine is called a patient ventilator. A ventilator controls the delivery of fresh mixed gases to the patient and the removal of waste gases, as illustrated in [Figure 1](#). Fresh mixed gas (containing between 21% and 100% O₂) enters the breathing circuit on the inspiratory side and mixes with "scrubbed" gas returning within the circuit to the patient. This gas is ultimately exhaled by the patient, thus entering the expiratory limb of the breathing circuit. The bellows assembly, under the control of the patient ventilator, powers the movement of gas pneumatically by pushing the bellows down to force inhalation and reducing the pressure in the drive gas to allow a passive exhalation and a refilling of the bellows.

The space internal to the bellows is filled with pre-mixed oxygenated patient gas, while the space between the bellows and its enclosing chamber is filled with drive gas. The drive gas provides the pneumatic power to compress the bellows, driving the patient gases through the CO₂ absorber and into the patient.

The details of the function of the bellows are provided in [Figure 2](#). During inspiration, the ventilator increases the pressure of the drive gas. In the bellows assembly shown in the figure, this pressure forces the top of the bellows down, increasing the pressure in the breathing circuit. The patient's elastic lungs expand, filling with oxygenated mixed gas. During expiration, the ventilator decreases the pressure of the drive gas. Once it falls below the pressure in the lungs, the lungs passively deflate, equalizing pressure with the drive gas.

In this closed breathing circuit, the expired gases are pushed through a soda lime canister that removes CO₂ from the mixture. This scrubbed gas is then mixed with some amount of new, fresh gas replacing the removed oxygen.

In addition to the movement of gases, ventilators usually perform some machine monitoring to enhance patient safety. The common parameters monitored are:

- O₂ concentration in the inspired limb of the breathing circuit (fiO₂)
- CO₂ concentration in the expired limb of the breathing circuit (etCO₂)
- Volume flow through the patient
- Breathing circuit pressure sensor

These monitored parameters are part of the safety function of the ventilator. Although the physician may set the O₂ concentration in the incoming mixture to high value, some means must be provided to detect whether or not inspired gas is hypoxic due to some system fault. Although other means for detecting hypoxic breathing mixtures are available,⁴ a ventilator can raise an alarm when it detects a hypoxic gas mixture, and actively bring it to the anesthesiologist's attention. The parameter is called fractional inspired oxygen concentration, or simply fiO₂.

One of the common problems with controlled or assisted ventilation is the insertion of the endotracheal tube, a piece of plastic tubing that is inserted into the trachea of the patient and attached to the "Y" piece of the breathing circuit (see [Figure 1](#)). The tube conducts gas between the patient's lungs and the breathing circuit. This arrangement works pretty well as long as it is, in fact, the trachea that is intubated. Occasionally, the endotracheal tube is instead inserted into the esophagus. A pressure sensor will detect a reasonable pressure wave, but what is really being ventilated is the patient's stomach. The only thing in a correctly-connected breathing circuit that creates CO₂ is the patient's metabolic system, and the CO₂'s only conduit to the breathing circuit is via the endotracheal tube. Therefore, a CO₂ monitor on the expiratory limb can detect the lack of CO₂ during expiration, allowing the ventilator to raise an alarm if the expiratory limb has insufficient CO₂.

A volume flow sensor permits the ventilator to monitor the actual volume of gas exiting the patient's lungs. This is useful because the ventilator must check that the amount of gas being delivered is (more or less) what was set by the user (some loss typically occurs due to compliance of the tubes in the breathing circuit and small gas leaks).

The pressure sensor provides additional information, including a pressure waveform. This allows the determination of a measure I:E ratio (see [Table 1](#)), as well as the end-expiratory pressure.

Additionally, blocked or kinked hoses will result in high inspiratory pressure that can be detected by a pressure sensor. The type of alarm raised helps the user identify the source of the fault so that it can be expediently corrected.

The Gasp-o-matic

The Gasp-o-matic ventilator includes both set (controlled) and measured parameters. These parameters are: inspiratory flow rate, respiration rate, I:E ratio, tidal volume, minute volume, peak pressure, expiratory CO₂ concentration, expiratory pressure, and inspiratory O₂ concentration.

Obviously there is some redundancy here. For example, minute volume may be directly set on some ventilators, but on others it is derived from tidal volume and respiration rate.

The Gasp-o-matic not only allows the user to control the particulars of the delivery of anesthetic gas to the patient, it also allows the monitoring of the delivery to ensure that it matches the user settings. Additionally, the ventilator alarms if something untoward occurs during the surgical session.

Alarming for most hazards is an appropriate safety measure for our anesthesia ventilator because (1) the user is a trained professional and in attendance,⁵ and (2) the most common hazard, asphyxiation, has a fault tolerance time of five minutes (10 minutes if the patient is breathing 100% O₂). For safety hazards with short fault tolerance times, automatic intervention of the machine must be performed. For example, over-inflation of the lungs is a serious hazard with a fault tolerance time of about 250ms. In this case, the ventilator will not rely on the user to correct a fault but instead will provide a secondary pressure relief valve (done mechanically) to protect the patient's lungs.

Alarms on the Gasp-o-matic are classified into three groups: informational (low criticality), caution (severe injury or death will result eventually if no corrective action is taken), and critical (immediate impending injury or death if no corrective action is taken). Each of these different alarms has slightly different behavior.

Informational alarms are displayed for a brief period-no more than two minutes (less if they are replaced by higher-priority alarms). Informational alarms are displayed in green. Caution alarms are displayed until acknowledged by the user (the user presses the alarm silence button). Caution alarms are displayed in yellow. Critical alarms will be annunciated until acknowledged, but they will re-annunciate every two minutes until the condition clears. Critical alarms are displayed in red. If a critical alarm condition disappears before being acknowledged, then the active annunciation via the speaker will cease and the alarm message will be grayed out. If the alarm condition reappears before being acknowledged, then it will maintain its current position within the alarm window, become "ungrayed" and be re-annunciated. If the alarm condition doesn't reappear before being acknowledged, then the alarm will disappear when it's acknowledged. Some typical alarms are shown in [Table 2](#).

The alarms are displayed in the alarm window, shown in [Figure 3](#). The alarm window holds up to five alarms, displayed in order of type (critical first, followed by caution, and then informational). Alarms of the same type are ordered chronologically, showing the most recent alarm of that priority first. A graphical indicator shows when not all alarms are displayed because of lack of space, so that the user knows to scroll the alarm window when not all active alarms are shown. Alarms are acknowledged by the Alarm Silence button. This only affects alarms that have been displayed because we don't want to remove an alarm that has never been seen by the physician.

The front panel shown in [Figure 3](#) provides all the user control of the ventilator. Note that all controlled parameters have both a set and a measured value. The set value displays the value commanded by the user. The measured value displays what the sensors detect is actually happening. Parameters that are measured only, such as O₂ concentration, only have a single value.

Modes of Operation

The ventilator operates in three modes: ventilation mode, configuration mode, and service mode. The ventilation mode is the normal operating mode for the machine. Configuration mode allows the user to configure the machine, set alarm limits, and so on. Service mode is used by the service personnel to update software, calibrate or replace the sensors, and so forth. The mode button on the front panel

is a three-way toggle, selecting the operational mode. For the purpose of our discussion here, we won't consider designing the configuration or service modes of operation, but in a real system, their design would require as much care as normal operational mode.

Requirements Analysis

In requirements analysis, we take a problem statement such as that above and try to uncover and rigorously define what the full set of requirements are. One of the reasons this is necessary is because the original problem definition is usually constructed by marketing folks unaccustomed to the rigorous thinking required of engineers. In complex systems, omitting important requirements is all too common, either because these requirements are "so obvious even an engineer would see it" or because they haven't bubbled up to the consciousness of the marketer. Also, requirements may conflict with each other and some requirements may just be plain wrong. The purpose of requirements analysis is to sort all this out.

The view taken by requirements analysis is of a black box. The concern is what the system does within its contextual universe and how it interacts with external actors. The basic tools of requirements analysis are use cases and scenarios. The astute reader will note that this approach is essentially a functional decomposition of the system, which is still the most common approach. Avionics systems engineers, for example, often use products like Statemate to do an initial functional decomposition of the requirements for an aircraft, complete with executable state models, before deciding which aspects will be done in hardware and which will be done in software. Once that initial work is done, software requirements analysis can begin.

Use Cases

A use case is a primary purpose or function of a system. Use cases communicate with some set of the identified actors, and are shown as named ovals on the use case diagram. Actors are represented using stick figures. Use cases may also extend (specialize) other use cases or use the facilities provided by another.

The use cases for the Gasp-o-matic are shown in [Figure 4](#). [Table 3](#) provides a short description of the identified use cases.

Scenarios

As I discussed last month, a scenario is an instance of a use case. It shows a particular interactive sequence of message exchanges between the use case and the participant actors. Each use case represents an essentially infinite set of different scenarios. Fortunately, only a few importantly different scenarios must be explicitly modeled because the others will be trivial variants of this small set. I call the set of important scenarios the orthogonal scenario set. In most systems, anywhere from a dozen to a few dozen use cases are identified. Each of these use cases will have from one to a few dozen scenarios of interest.

In this article, we'll use the message sequence diagram, as we previously discussed, to represent the scenario. Consider the setting of tidal volume from 400ml to 500ml, as shown in [Figure 5](#). Through the marketing person we uncover that there is a hidden requirement for the user to confirm his action. This, our customer explains, is so that inadvertently dragging a sleeve or hose over the knob won't unexpectedly change our delivery settings.

A typical scenario analysis with a marketer will go something like this:

"That's reasonable," we say. "How would you like to see that work?"

"Well, I think we should use a turn knob that you have to push in to confirm."

"Okay, so what do you want to do if the user doesn't confirm?"

"If the user doesn't confirm, then the setting should never be applied. After, oh, a minute or so, the setting should revert back to what it was."

"How is the user supposed to know if a setting is confirmed or not?"

"The setting value should blink until it is confirmed."

"Are you sure you want two user actions for each user function? What about alarm silence?"

"All except alarm silence. That should work as a single action."

And so on. This process continues until the scenarios are fully explored and a complete set of requirements emerges. The result of each of the sessions is a scenario like the one in [Figure 5](#).

The reason that the technique of scenario analysis is so valuable is that a complex state diagram is more likely to confuse most marketers than help them, and enough scenarios can illustrate all interesting behavioral variations. Consider the following exchange:

"So what happens if the user turns another knob before confirming the first," asks the marketer.

"Well in that case, the setting should revert to its previous value as soon as another knob is turned."

Now we have another scenario, as shown in [Figure 6](#).

Repeat this process enough and you'll have a description of the marketer's view of what the system should do from an external viewpoint. Armed with a more-or-less complete set of requirements for what the system must do, we are ready to "open up the box" and begin the process of identifying and defining the objects inside the system. This process is called object analysis.

Object Analysis

The purpose of object analysis is to identify the objects, classes, and relationships that are inherent in the problem. Put another way, object analysis is an act of discovery of all properties of a system that must be present in any acceptable solution. Design, as we will see next month, is the process of elaborating the analysis model for a particular implementation.

The primary tools in object analysis are the class diagram, scenarios, and statecharts. Class diagrams capture the object structure of the system, including the classes, objects, and their relationships. Dynamic interaction among sets of objects are captured on sequence diagrams, and the behavioral space for reactive objects is captured on statecharts logically bound to the individual reactive classes.

Class Diagrams

The natural place to start object analysis is with the class diagram. The major activities are the identification of the objects and classes, clarification of the responsibilities of the identified objects (and the primary attributes and operations of the objects that will be used to achieve those responsibilities), and identification of the relationships and the properties of those relationships.

Object Identification

The objects inherent in the system aren't always obvious. I use several different strategies on any given system. These strategies usually overlap to some degree but the use of several different strategies maximizes the probability that you will find all the essential objects. The strategies we'll use here are:

- Physical devices
- Visual elements
- Transactions

Naturally, there are dozens more strategies that I could use in lieu of or in addition to these, but let's limit our focus a little.^{6,7}

What are some of the physical devices? The primary ones from the problem statement are:

- Ventilator
- O2 sensor
- CO2 sensor
- Pressure sensor
- Display
- Push knob
- Button
- CO2 absorber
- Drive gas control valve

Now just because a strategy identifies a potential object, that doesn't mean that you should always use it. The word "ventilator" appears all over the problem statement, but it may not be a reasonable object to have. Of the objects suggested by this strategy, I would be inclined to throw out the ventilator and CO2 absorber because it isn't obvious that the software will have anything to do with them. The last object in the list, the drive gas control valve, is implied by the necessity to control the flow of drive gas into the bellows to power the flow of breathing circuit gas into the lungs.

The visual elements strategy is very common in programs that have a significant GUI component. In the Gasp-o-matic ventilator, the GUI identifies not only GUI-type things, but also suggests data values that must be measured, controlled, or calculated. A list of visual elements is:

Visual GUI elements

- Label string
- Value string
- Alarm string
- Hidden element indicator
- Push knob

Data elements (measured or controlled)

- Tidal volume

- Minute volume
- Inspiratory flow rate
- I:E
- Respiration rate
- Airway pressure
- fiO₂
- etCO₂
- Alarm

Transactions, in this context, are messages or events that must persist for a nontrivial amount of time. Alarms are an obvious choice for transactional objects within the Gasp-o-matic, but other possibilities exist as well. We've already identified that the machine must track service calls so it can notify the user when the next one is getting close, so a service call may be a transactional object. Also, the process of setting a controlled parameter involves transactions because it must be stored until it is confirmed. Looking forward, one could imagine a service requirement emerging that would require an error log to store machine faults to aid the service technician. In such a system, errors are transactional objects.

Transactions that are grouped together often have a transaction manager. It is common, for example, to create an alarm manager object to manage the set of active alarms.

Responsibilities, Attributes, and Behaviors

For many of the objects in the system, the responsibilities, attributes, and behaviors are obvious. For an example, see [Table 4](#).

Some properties of objects may not be so clear. For example, the problem statement says that a number of different hazardous conditions must be identified, such as low-inspired O₂ concentration, low end-tidal CO₂, high airway pressure, and so on. Whose job is it to identify the hazardous condition of low-inspired O₂? It would be possible to have the O₂ sensor do it, but if we want the low alarm limit to be programmable, this would require a smart (read "expensive") sensor when a dumb (read "cheap") one might be more cost-effective. In this example, we'll create a parameter class that contains a low alarm limit responsible for checking. The parameter class will act as a kind of proxy for the ultimate source of the information, and also identify whether alarm limits are exceeded.

Looking at some other objects, we can build a repository of information about potential classes, as seen in [Table 5](#).

Relationships

Objects never stand alone. An object is a fairly small level of decomposition, and several of them must collaborate together to achieve a system-wide function (use case). The two kinds of associations we'll use most in this problem are association and generalization.

An association is required for objects to send messages to each other at run-time. For example, if we have an O₂ sensor and an O₂ parameter object, an association is required for the O₂ sensor to send the current O₂ concentration value to the O₂ parameter. We might (and will, in this case) also say

that each parameter has a "view" object that presents the value of the parameter on the display for the user. This requires another association between the O2 parameter object and the O2 view object. This is shown in [Figure 7](#).

Controlled parameters have somewhat more complex lives. For example, they must also associate with a push knob. Further, they must also know what their high and low setting limits are, and must track measured versus controlled values. This is shown in the next figure. Note that while [Figure 7](#) was an object diagram (a colon separates the name of the object from the name of the class), [Figure 8](#) is a class diagram. Thus, it applies to all controlled parameter objects.

Notice that we've included the role names with the two associated view classes to clarify the associations. We've also included the role multiplicities on the associations to identify the number of instances of the classes that participate in the association at run time.

There are several obvious opportunities for generalization in the Gasp-o-matic. Parameter is one. Perhaps a controlled parameter is a specialized version of parameter that extends the functionality of the base class. It may be that all the sensors derive from a common base class. A taxonomical hierarchy of alarm types is another possibility.

Let's consider the control of the ventilation itself. There are several partially redundant controlled parameters: tidal volume, minute volume, respiration rate, inspiratory flow rate, and I:E ratio. The first three are mathematically related, where:

$$Mv = Tv * R$$

Mv is minute volume

Tv is tidal volume

R is respiration rate

The inspiratory flow rate and the I:E ratio also enter into the picture. Let's suppose a minute volume of 5,000ml/min, a tidal volume of 500ml, and a respiration rate of 10 breaths/min. Each breath can be divided into three phases: inspiratory, expiratory, and pause between breaths. The total time for the sum of these phases must equal the time allotted for each breath:

$$I + E + P = 60/R$$

In this case, if the I:E ratio is 1:2, and the pause is set to one second, then the time for inspiration is 1.667s, and expiration takes twice as long: 3.333s. The time available for inspiration (I) and the required volume to be inspired (Tv) must define the inspiratory flow rate (IFR):

$$IFR = \frac{Tv}{I}$$

In this case, the inspiratory flow rate is 500ml/1.667s, or about 300ml/sec. If the inspiratory flow rate is changed, then either the I:E, tidal volume, or respiration rate must change.

The management of the breathing control is the responsibility of the BreathControl class. [Figure 9](#) shows a first-cut analysis object model for the Gasp-o-matic ventilator.

Note that we decided to add a few classes. For example, the alarm manager associates with an AlarmListView class, which in turn associates with the AlarmView class (one to many). Meanwhile, the alarm manager also associates with many Alarms, each of which has an alarm view.

Scenarios

Is the class diagram in [Figure 9](#) a reasonable arrangement of classes to handle alarms? Can they all get the required information to achieve their collaborative goal? Somehow we must make sure that the alarm views associated with the AlarmListView are the same ones that are associated with the currently active alarms that are (indirectly) associated with the alarm manager. How does this system work exactly? We can "drill down" and explore the interaction of the class by analyzing a scenario. [Figure 10](#) illustrates this procedure. The hatched line indicates the "System Boundary" and represents all objects in the universe not explicitly shown on the message sequence diagram.

Remember that scenarios show objects, not classes. The scenario in [Figure 10](#) shows the two active alarms as well as their associated views as different objects on the diagram.

[Figure 11](#)

[Figure 12](#)

What about the setting of tidal volume, minute volume, and respiration rate? It's possible to set the three to incompatible settings. Let's suppose that the machine will make adjustments in the parameters that aren't set to optimize the others. Suppose that first we set tidal volume to 500ml and then respiration rate to 10 breaths/min. Now we set minute volume to 6 liters/min. What should the system do?

One approach would be to remember which parameter was set last and keep that one the same and vary the oldest parameter. For example, in the above scenario, we would adjust tidal volume to 600ml because it was set first. On the other hand, really low minute volumes are best achieved by a relatively large tidal volume but a low respiration rate. This is because of the physiological dead space—that is, volume in the patient's breathing system that is not involved in gas exchange. If the tidal volume is too low, only the dead space will get fresh gas and so the patient will not be able to exchange O₂ and CO₂. The physiological dead space includes the volume of the trachea and the bronchioles. However, to achieve really large minute volumes, a high respiration rate in combination with a large tidal volume is probably best. This is further complicated by the necessity to support a wide range of patient sizes, from, say, 2kg up to 200kg. Neonates cannot tolerate large tidal volumes and have a much smaller physiological dead space. One solution for this is to allow the physician to control the set limits, but an easier one is to permit the physician to select from among three sets of limits: neonate (1kg to 6kg), pediatric (6kg to 30kg), and adult (more than 30kg). Thus we must add a control to the display to set the patient mode (shown in [Figure 9](#)).

Statecharts

A number of classes in this example show state behavior. Remember that such classes are called reactive because they react to events. Let's look at three such classes: PatientMode, BreathControl, and Alarm.

The PatientMode class is easy. It has three states: Neonate, Pediatric, and Adult. The entry into these states is set by the associated PushKnob. The actions are taken upon entry to the states. All the

actions are similar-first the patient limits are updated, and then the BreathControl is sent a ChangeLimits event to signal that it must handle new limits.

The BreathControl class has two orthogonal state components. The main one of these controls the phasing of the breath via setting the drive gas flow to the bellows. The second component controls the settings. This component has a single state called Waiting_for_Update. This state accepts two events: ChangeLimits, which sets the limits based on patient size (from an event sent by the PatientMode class); the other event is sent from any of the ControlledParameters whenever their value is changed.

The Alarm class is used to handle caution alarms that must be explicitly acknowledged. The state machine for this class is shown in [Figure 13](#). The Active state has two substates: Waiting_For_Ack and Acknowledged. The former state has two substates: NotViewed and Viewed. When the associated AlarmView has been displayed by the AlarmListView, a WasSeen event is sent by the AlarmView back to the Alarm. The danger avoided is that an unviewed alarm shouldn't be removed without the user having an opportunity to see it. Explicit acknowledgement should only affect an alarm if it has been viewed.

The Alarm class has two subclasses. The first is the InfoAlarm class. This is a subclass because it has all the behavior of the Alarm class plus more-it also has an automatic timeout, as shown in [Figure 14](#). This is an example of state inheritance. Because the superclass (Alarm) is reactive and its subclass is a part of its parent, it follows that the subclass must also be reactive. Further, it should include all the states and event transitions of its parent class.

The second subclass is the CriticalAlarm subclass. Its behavior is slightly more elaborate. In this case, if the alarm condition ceases, it must be "remembered" so that if it reappears before a certain time, the alarm will retain its position in the alarm list. If it does not reappear within that time, the alarm is removed.

Once more from the top

We have seen, through the vehicle of the Gasp-o-matic ventilator, the process of analysis of a real-time system using the standard UML syntax and semantics. We broke the analysis into two phases: requirements analysis and object analysis. The UML provides use cases and scenario modeling via message sequence diagrams for the identification and extraction of requirements. We identified eight uses cases in this particular example. In a real development cycle, we would probably have identified and drawn at least a dozen scenarios for each of these.

Once we "open up the box" and peer inside, the UML class diagram, message sequence diagram, and Statechart are used to identify objects, their relationships, and their reactive behavior. We used three object identification strategies and ultimately identified 19 classes and arranged associations that appear to support the required messaging passing among them. We then looked at a scenario of usage of the alarm system as a quick "test" that the class model was adequate. Finally, we elaborated some of the class state models to capture their required behavior.

In the next (and final) installment, we'll explore how the UML supports the design of real-time systems. We will do this with the elaboration of the diagram types we've used so far, but also with the inclusion of design patterns and deployment diagrams. We will proceed in each of the three phases of design: architectural design (allocation to processors, definition of thread management

policies, and allocation of objects to threads), mechanistic design (application of design patterns consisting of a few to several objects), and detailed design (design of object's internal structure).

Bruce Powel Douglass has almost 20 years' experience designing safety-critical real-time applications in a variety of hard real-time environments. He is an advisory board member for the Embedded Systems Conference. He also worked with methodologists at Rational and other companies on the UML specification. He is currently employed as the chief evangelist at i-Logix, and can be reached at bpd@ilogix.com.

REFERENCES

1. I try to breathe every chance I get.
2. Ehrenwerth, Jan and James Eisenkraft. Anesthesia Equipment: Principles and Applications. New York: Mosby-Year, 1993.
3. Dorsch, Jerry A. and Susan E. Dorsch. Understanding Anesthesia Equipment: Construction, Care and Complications. Baltimore, MD: Williams & Wilkins, 1994.
4. The patient typically turns blue.
5. Don't try this at home!
6. Douglass, Bruce Powel. Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications. Reading, MA: Addison-Wesley-Longman, 1998.
7. Douglass, Bruce Powel. Real-Time UML: Efficient Objects for Embedded Systems, Reading, MA: Addison-Wesley-Longman, 1998.

Other Sources

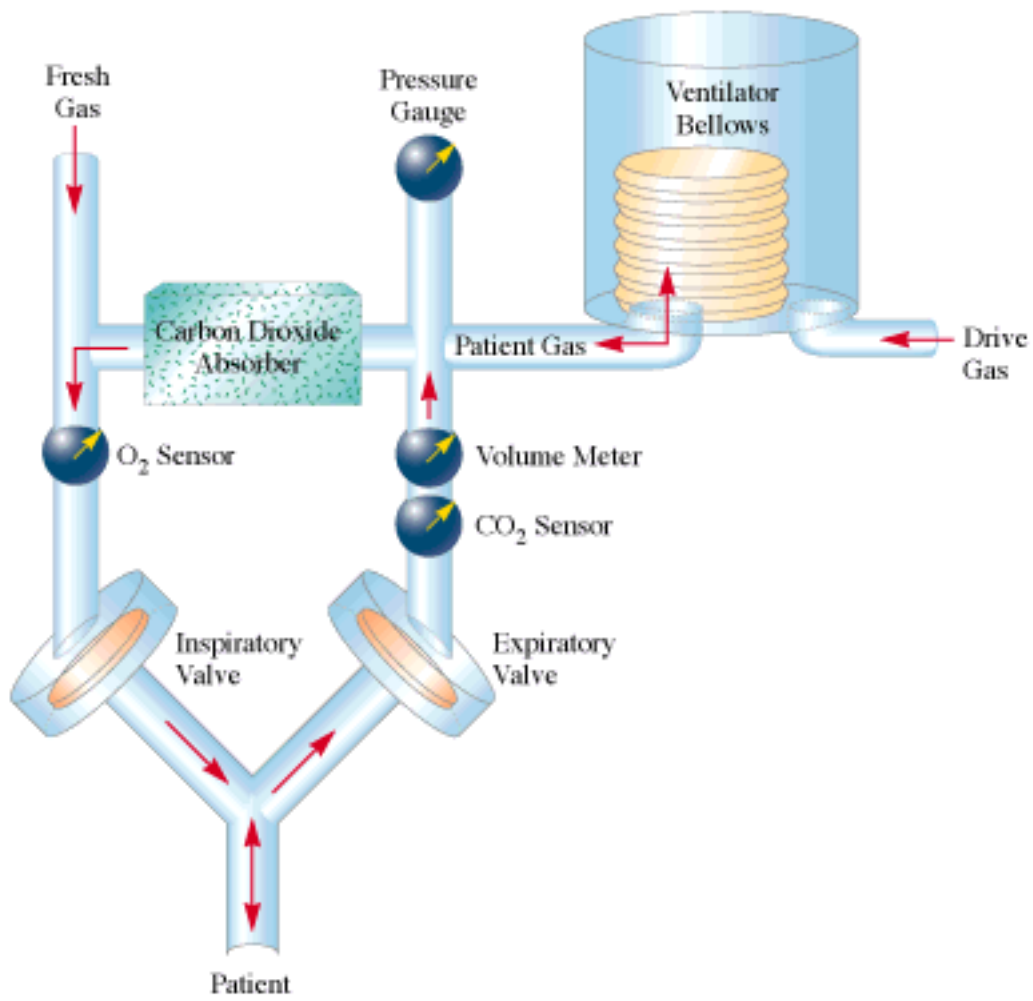
Fowler, Martin and Kendall Scott. UML Distilled: Applying the Standard Object Modeling Language. Reading, MA: Addison-Wesley-Longman, 1997.

Harel, Naamad, Pnueli, Politi, Sherman, Shtull-Trauring, and Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," IEEE Transactions on Software Engineering (16), 1990, p. 403.

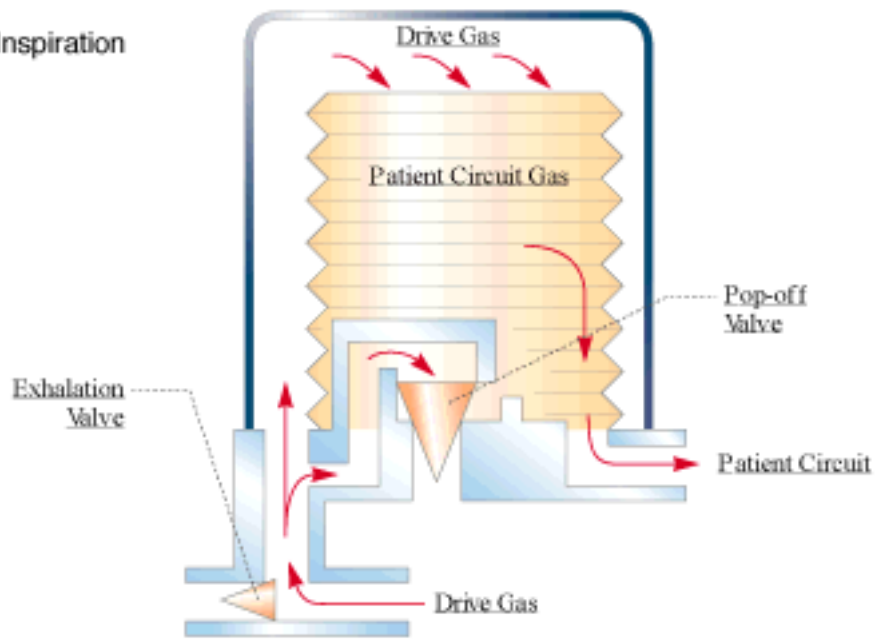
Rational Corp, et. al., UML Notation Guide Version 1.1, September, 1997. (As submitted to the OMG.)

Rational Corp, et. al., UML Semantics Version 1.1, September, 1997. (As submitted to the OMG.)

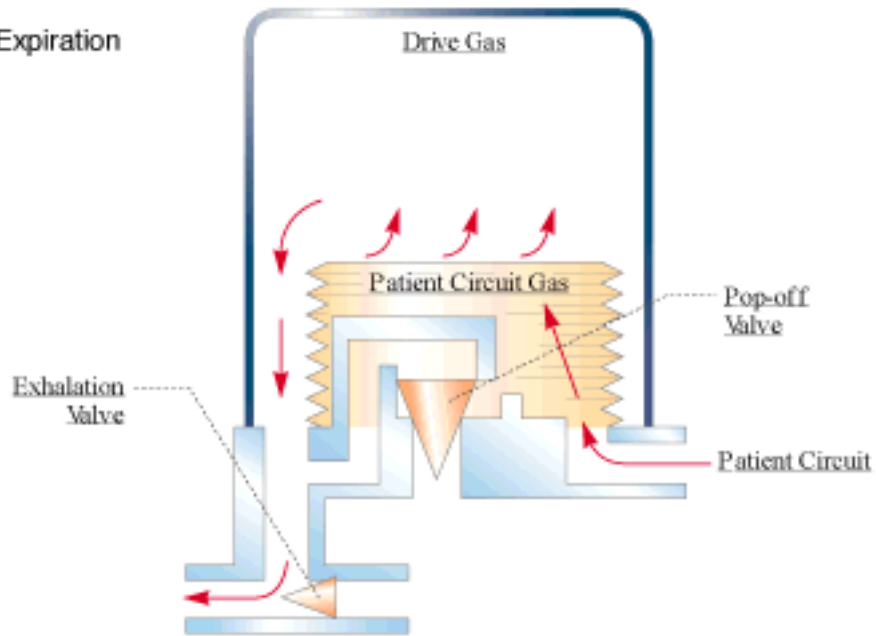
Rational Corp, et. al., UML Summary Version 1.1. September, 1997. (As submitted to the OMG.)



Inspiration



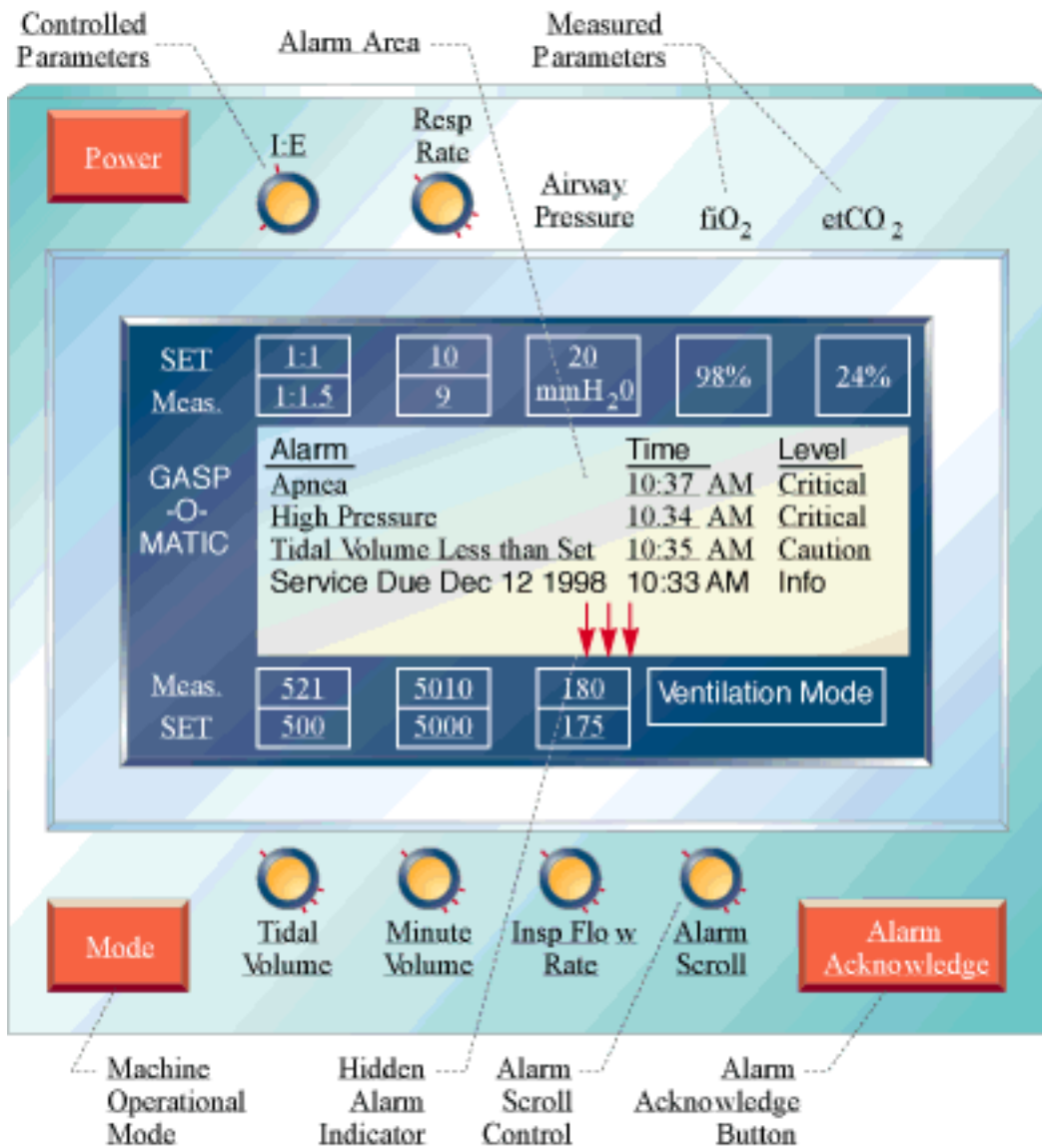
Expiration



Parameter	Measured	Set	Description
Inspiratory flow rate	y	y	The rate of gas flow during the inspiratory cycle
Respiration rate	y	y	Breaths per min. (normally 250ml to 1,750ml)
I:E ratio	y	y	Ratio of time per inspiration and time per expiration
Tidal volume	y	y	Volume delivered per breath
Minute volume	y	y	Volume delivered per minute
Pressure	y	y	Delivered or expired pressure
et CO ₂	y	n	End-tidal (expiratory) CO ₂
Airway pressure	y	n	Pressure in the airway
in O ₂	y	n	O ₂ concentration (inspired)

TABLE 2
Typical alarms.

Alarm	Criticality	Description
Apnea	Critical	Patient isn't breathing.
Low O ₂	Critical	O ₂ concentration is below atmospheric (21%)
High Airway Pressure	Critical	Airway pressure is high, suggestive of an airway blockage or kinked airway hose.
Low Airway Pressure	Critical	Airway pressure is low during inspiration, suggestive of an airway disconnect or severe leak.
Low etCO ₂	Critical	Patient's expired CO ₂ is significantly below expected, suggestive of an esophageal intubation.
Internal Error	Critical	An internal, uncorrectable software error has been identified.
Minute Volume Less than Set	Caution	Commanded minute volume wasn't delivered.
Tidal Volume Less than Set	Caution	Commanded tidal volume wasn't delivered.
Tidal Volume Cannot Be Delivered	Caution	User has set a tidal volume that cannot be delivered due to other settings.
Minute Volume Cannot Be Delivered	Caution	User has set a minute volume that cannot be delivered due to other settings.
Device Version	Info	Message displayed on startup identifying the version and revision numbers of the machine
Service Due <date>	Info	Message displayed on start up when the service date is within 30 days or is overdue.



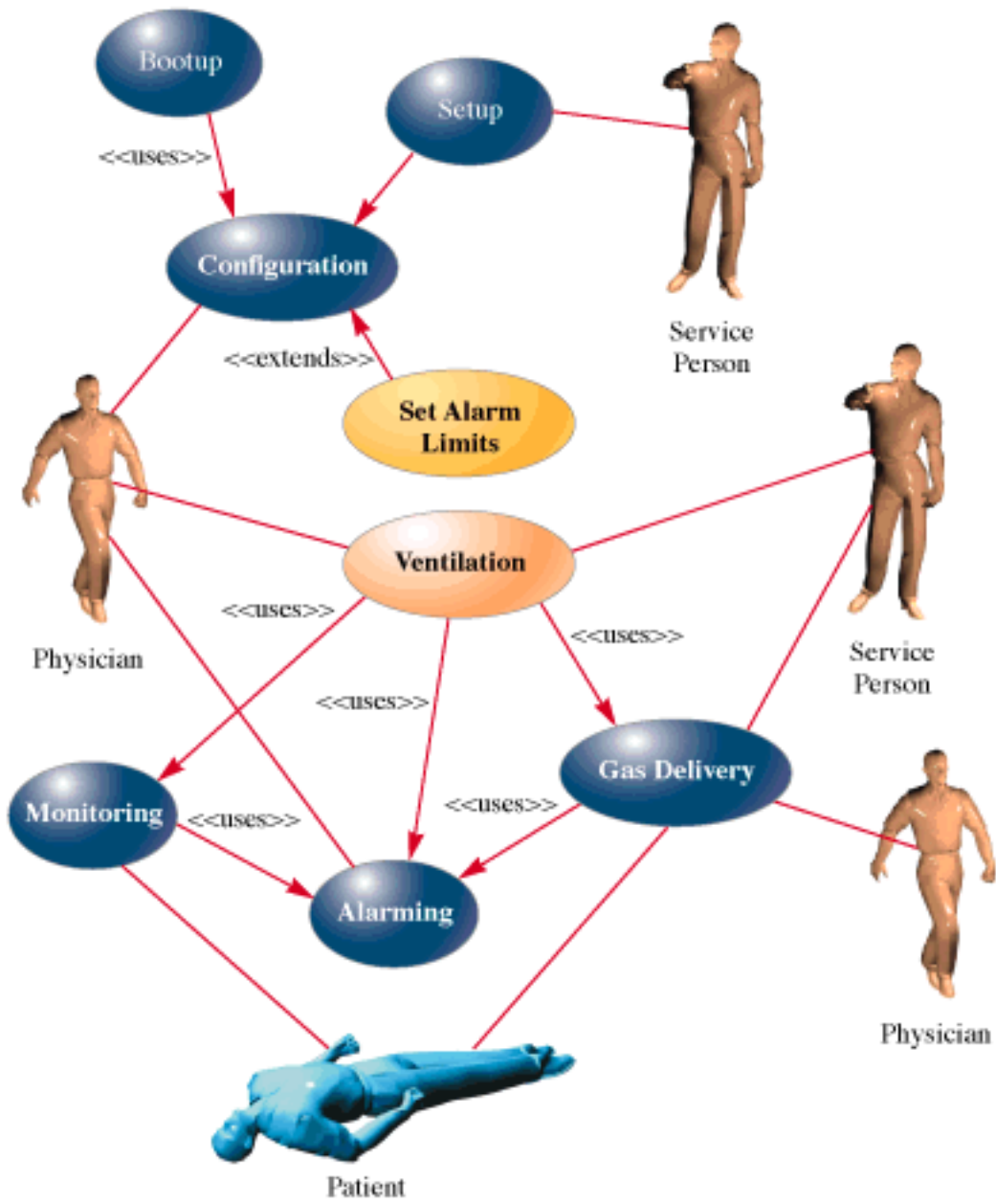
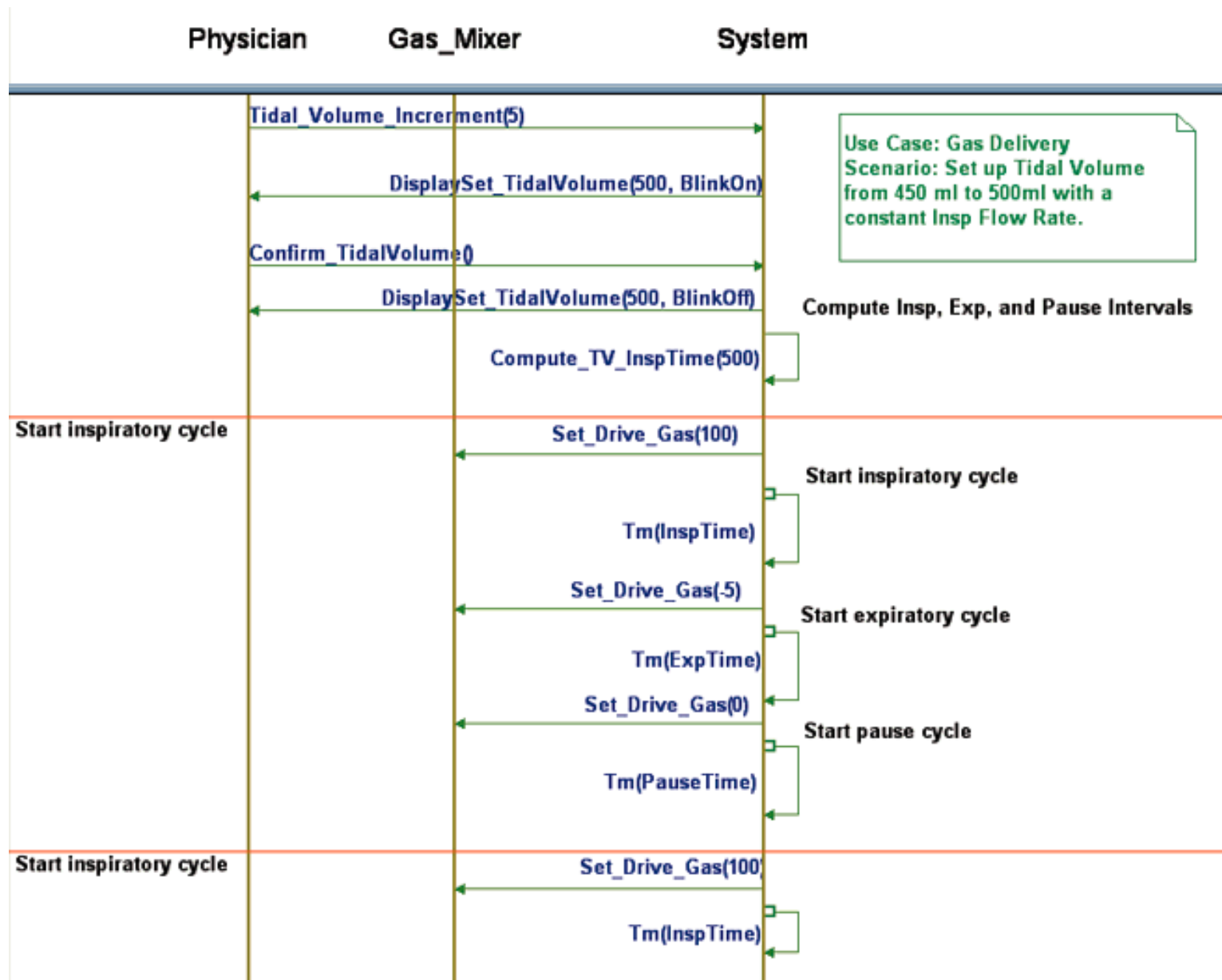
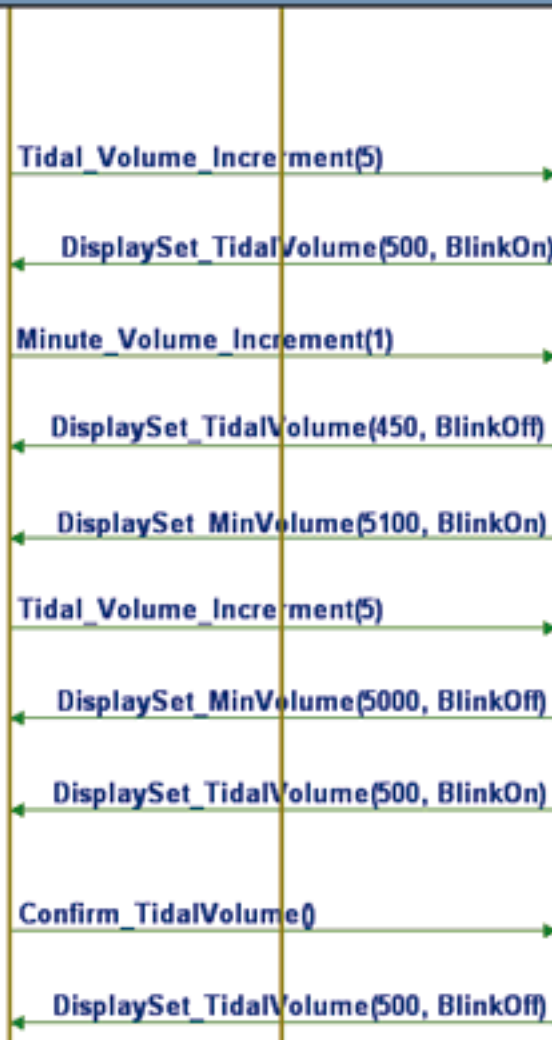


TABLE 3
Gasp-o-matic use cases.

Use Case	Related Use Cases	Participant Actors	Description
Setup		Service person	This use case deals with how to set up and service the machine, including software updating, calibration and replacement of sensors, entry of configuration data (such as altitude) and other service-related functions.
Bootup	Configuration		This use case deals with the power-on initialization of the device including Power On Self Test, initializing devices, and restoring known default alarm limits and settings.
Configuration	Set alarm limits	Physician	This use case deals with how the user configures and sets up the machine.
Set alarm limits	Extends: configuration	Physician	This use case is a particular type of configuration—specifically, the setting of the values of particular alarm limits.
Ventilation	Uses: gas delivery, monitoring, alarming	Physician, gas mixer, patient	This use case deals with the normal operation of the machine. It uses the facilities provided by several other use cases.
Gas delivery	Used by: ventilation	Physician, gas mixer, patient	The Gas Delivery use case deals with how the machine actually controls the delivery of gases to the patient, including control of the drive gas and responding to user parameter settings.
Monitoring	Used by: ventilation	Physician, gas mixer, patient	The Monitoring use case deals with how the machine monitors itself, the delivery process, and the indirect measures of patient status, including $etCO_2$, inO_2 , tidal volume, minute volume, respiration rate, I:E ratio, and inspiratory flow rate.
Alarming	Used by: ventilation	Physician, gas mixer, patient	This use case deals with how the machine notifies the user of unexpected and potentially dangerous conditions and the management of those alarms.



Physician Gas_Mixer System



Use Case: Gas Delivery
Scenario: Use moves another knob before confirmation and must reenter and confirm their original change.

User adjusts knob +5 ticks to move to 500 ml. System responds by displaying set value with blinking turned on.

Oops! User knocks the Minute Volume control knob accidentally. System restores original Tidal Volume setting, and sets Minute Volume setting to 5100 with blinking on.

That's not what the user wanted to do, so now ge resets the +5 ticks on the Tidal Volume. The system restores the original Minute Volume setting to 5000 (no blinking) and displays the (unconfirmed) Tidal Volume setting of 500.

The user confirms and the system accepts the new value and indicates this by turning off

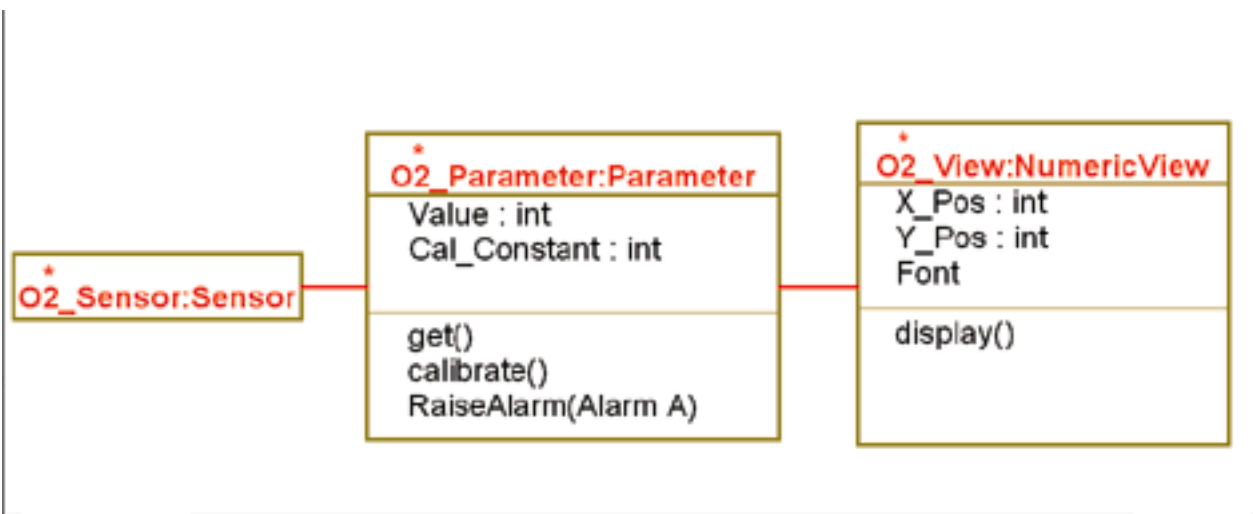
TABLE 4

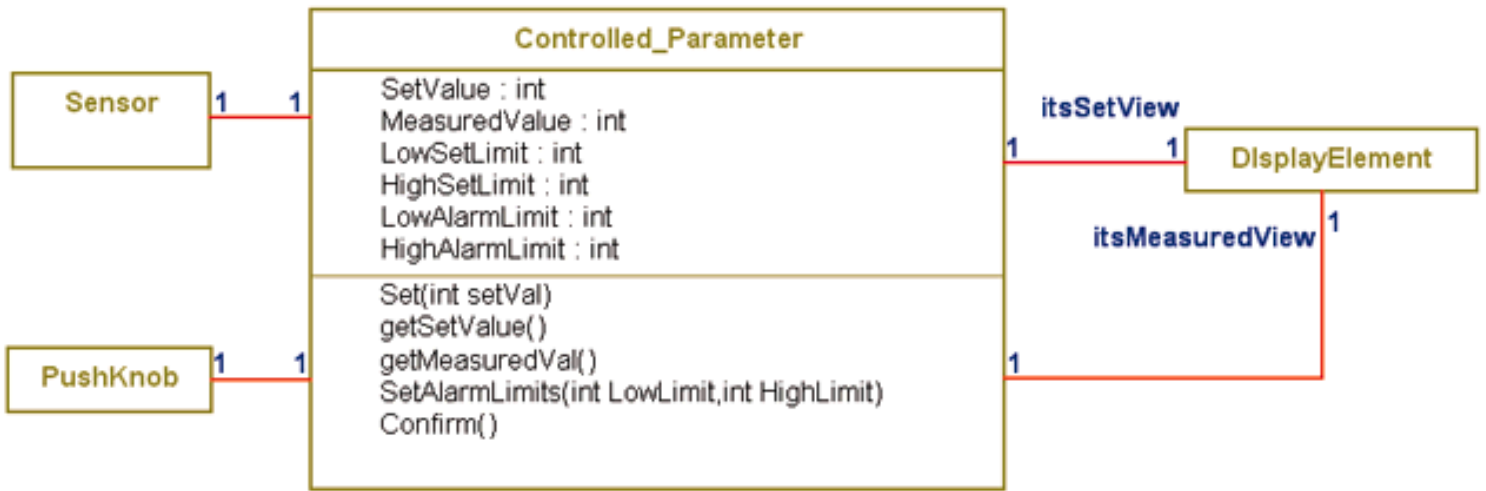
Responsibilities, attributes, and behaviors of an O₂ sensor object.

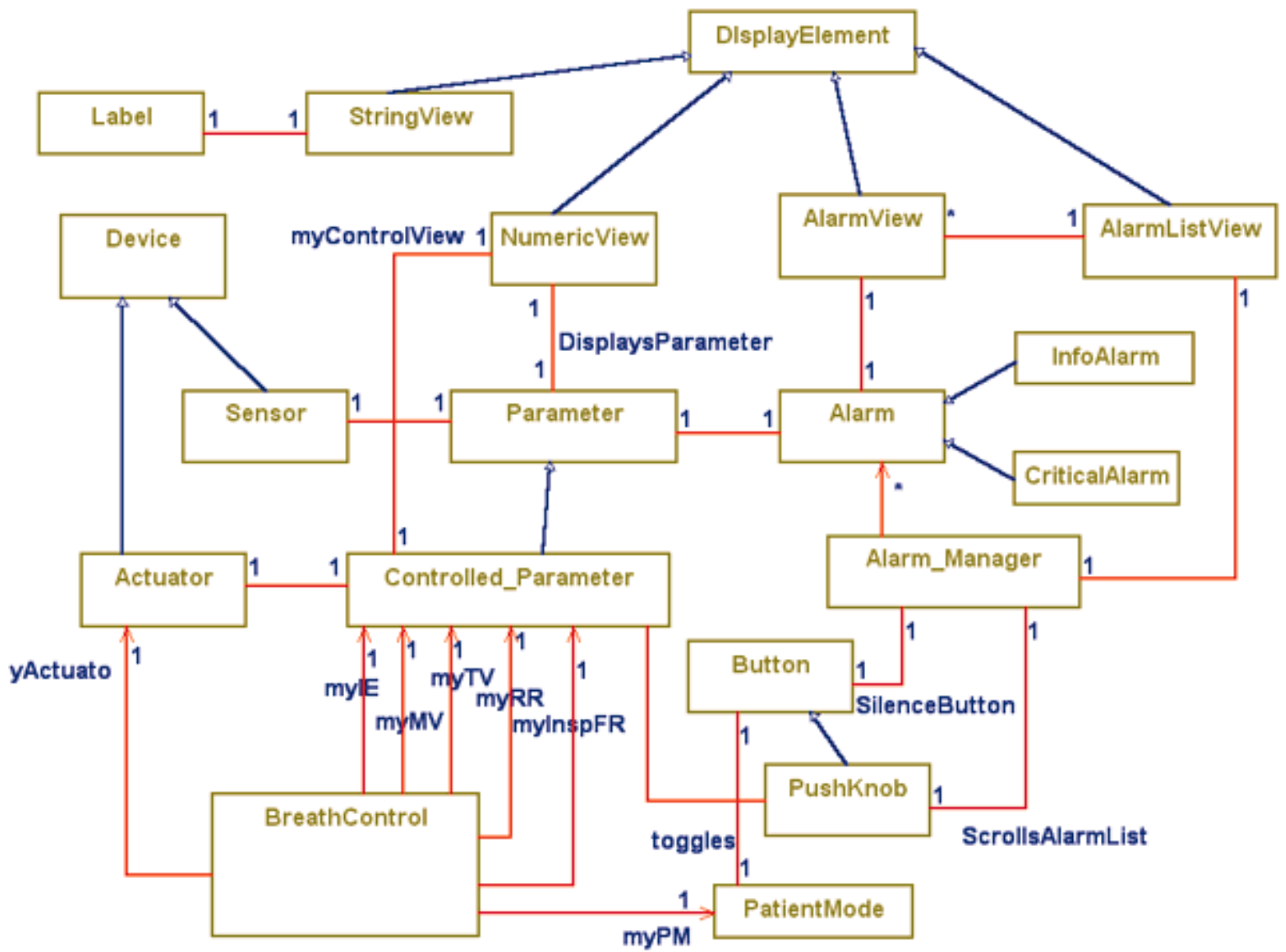
<u>O₂ Sensor Object</u>		
<u>Responsibilities</u>	<u>Attributes</u>	<u>Behaviors</u>
Monitor the O ₂ concentration of the inspired gas to ensure a non-hypoxic mixture.	O ₂ measured value calibration constant	Calibrate() Get()

TABLE 5
A repository of information about potential classes.

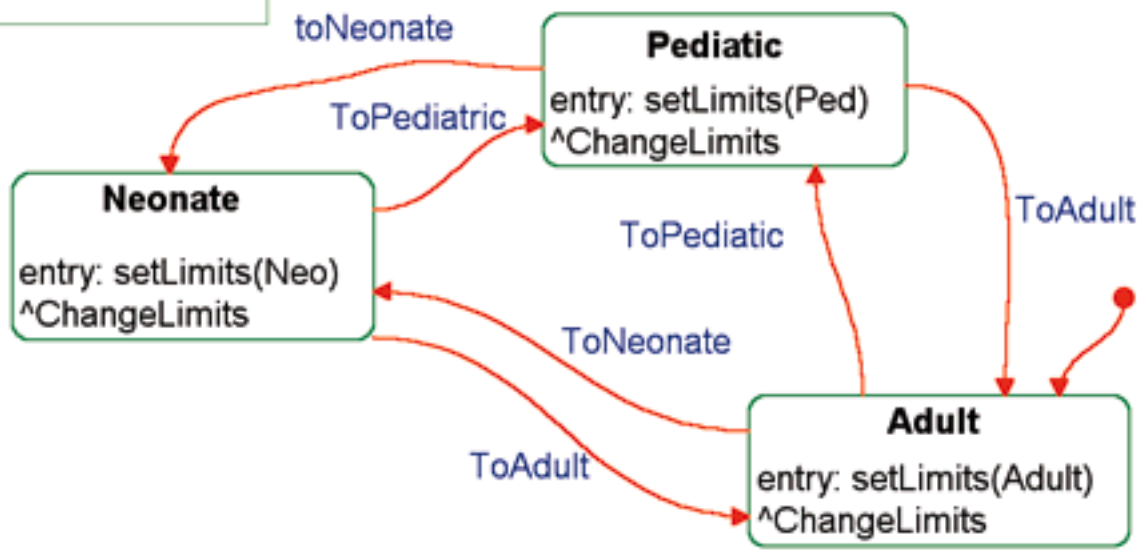
Proposed Class	Responsibility	Attributes	Behaviors
Alarm	Identifies a potentially hazardous condition	Time of occurrence, criticality, has been viewed	Annunciate
Alarm View	Displays information about a single alarm to the user	Position, font, color	Display, hide
Alarm Manager	Manages the set of alarms, deciding when to display alarms and handle the acknowledgement of alarms	Number of alarms	Display, remove, silence, scroll
Numeric View	Displays a numeric value on the display	Position, font, color	Display
Tidal Volume Parameter	Control of commanded tidal volume and monitoring of measured tidal volume	Measured value, controlled value, low set limit, high set limit, low alarm limit, high alarm limit	SetControlled, RaiseAlarm, SetLimits, SetAlarmLimits
Pushknob	Allows the user to set a controlled parameter	Position	Turn, push

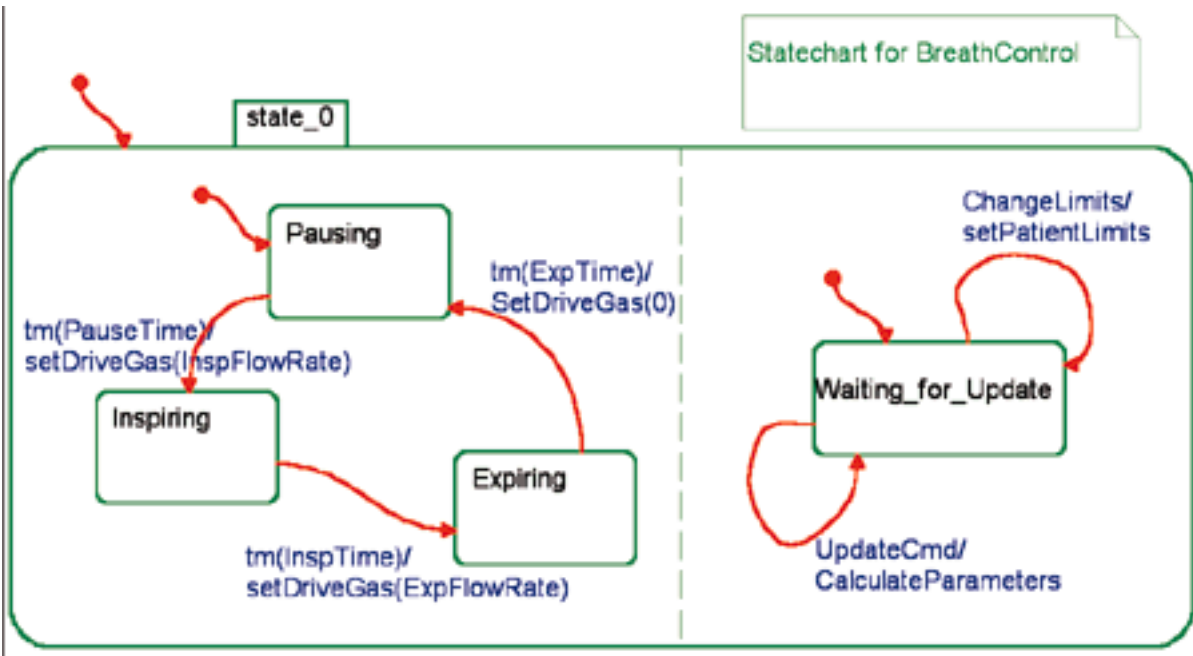




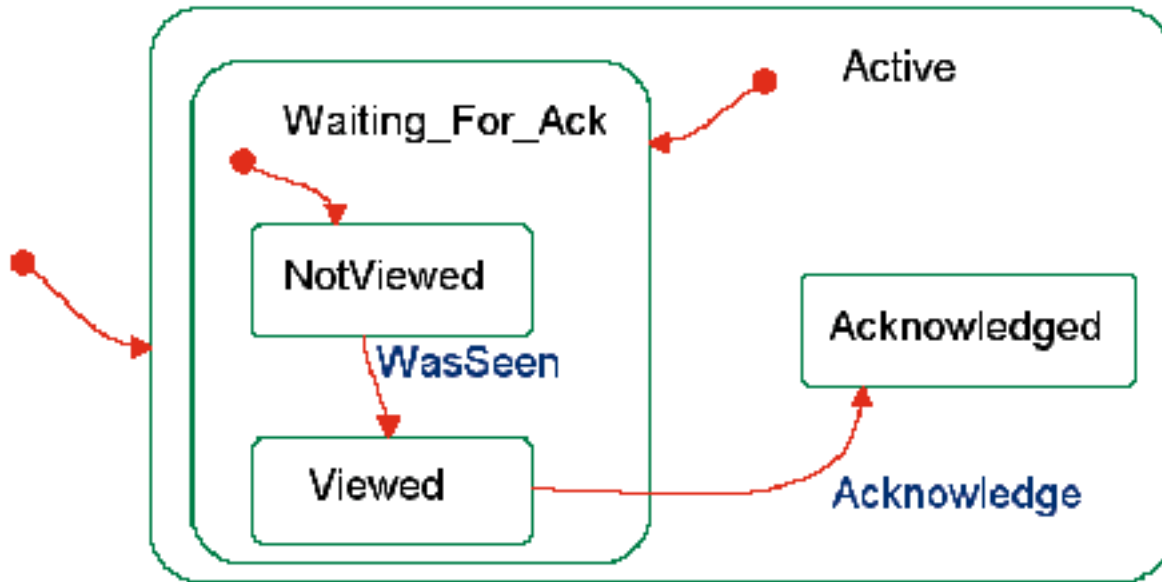


Statechart of PatientMode





Statechart of Alarm
This is used for Caution
alarms.



Statechart for InfoAlarm.
Same behavior but will also self
terminate based on time in the
absence of an explicit user
acknowledgement.

