

Capturing Requirements for Real-Time and Embedded Systems

Bruce Powel Douglass
Chief Evangelist
I-Logix

Introduction: Fitting Requirements into the Development Process

Many developers regard requirements capture with a disdain normally reserved for Windows crashes and Richard Simmons exercise videos. They see it as a waste of time that diverts them from what they *ought* to be doing ... cranking out code. However, in a requirements-driven process, the developers always know that what they're doing actually relates to the goals and purposes of the system. To properly understand what features ought to be designed and implemented, as well as how they ought to work, a deep understanding of the purposes of the system, the workflow of the user (if applicable) with respect to the system, the set of features the system provides, the devices with which it must interact and how those interactions take place, what should happen when something expected or "bad" occurs, and exactly how the features must be visible to the user and the external devices. These are all part of the *requirements* or *specification* of the system. If you do a good job at understanding the requirements, then your development work will be most productive, there will be less rework, and your customers will be the happiest.

In a requirements-centric development, all the work done in the projects relates in some way to the requirements specification of the system. Early in analysis, we try to understand how the system fits into its environment (including the user). Soon, we're detailing exactly which features we want the system to provide to optimize it's work in that environment and exactly how we want those features to appear to elements in the system's environment. Later, we design the internals of the system to meet those specifications, and finally we construct test vectors from the system to ensure that we did it with the appropriate level of completeness, fidelity, and accuracy.

One of the difficulties I have found that real-time and embedded developers have is that they have great difficult separating out requirements to which the system *must* adhere to be correct from its *design*, which usually just one of several ways of meeting the requirements. And this is not just true of newbies just out of school. Many very bright and experienced developers are simply so engrained in design that they find this distinction very elusive. I have developed an approach for understanding, capturing, and manipulating requirements based on my work with complex projects at NASA and

elsewhere, which is the focus of this article. This approach is part of the ROPES process, discussed previously in this magazine and elsewhere¹.

Types of Requirements

Just as there are two kinds of people², there are two kinds of requirements: functional and quality of service. *Functional* requirements are requirements about what the system should do and how it should behavior in a variety of circumstances. For example,

- The system shall adjust the angle of the telescope under user control
- The system shall deliver anesthetic agent in gaseous form to a desired concentration.
- Locking clamps shall engage when the elevator cable breaks.
- The device shall alarm if the heart rate falls to less than 30 beats per minute.
- The pacemaker shall pace in AAI mode (pace the atrium, sense in the atrium, inhibit on an intrinsic heart beat detection).
- The spacecraft shall downlink captured information periodically to the deep space network.

Quality of Service (QoS) requirements specify *how well* a functional requirement shall be accomplished. In real-time and embedded systems, QoS requirements may specify properties of the system, e.g. range, speed, throughput, capacity, reliability, maintainability, evolvability, time to market, safety, predictability, schedulability; or properties of the process. As a rule of thumb, if it's something that can be quantified, or optimized, then it is a QoS requirement. For example (QoS requirements italicized),

- The angle of the telescope *shall be set in units of 0.1 degrees with a maximum error of 0.01 degrees.*
- The anesthetic agent *shall be controllable from 0.00% to 5.00% by volume in units of 0.01% with an accuracy of 0.005%*
- Locking clamps shall engage in the event of an elevator support cable breakage *within less than 0.5 seconds.*
- The device shall alarm *within 10 seconds* if the heart rate falls to less than 30 beats per minute.
- The pacemaker pacing rate shall be *settable from 30 to 120 paces per minute (inclusive) and during operation accuracy of pacing shall be ± 100 ms.*
- The *reliability* of communications from the spacecraft to the deep space network *shall be better than 0.9994%, with a single bit error detection/correction rate of 0.9999% and a multiple bit detection/correction rate of 0.99%*

The *defining characteristic* of real-time systems is the level to which QoS requirements around time figure into the correctness of the system. In non-real time systems, late is

¹ See Embedded Systems Programming Magazine, December 2000 issue *On the ROPES*. Or Chapter 4 in *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns* (Addison-Wesley, 1999).

² Those that divide people into two kinds and those that don't! ☺

acceptable, whereas in real-time systems, late is definitely *bad*. Put another way, a real-time system is not necessarily fast, but it is *predictably timely*. Of course, real-time systems may be hard -- that is,

- responses to events for aperiodic systems or
- actions taken when a periodic task begins (for periodic systems)

must complete by a specified deadline.

Or they may be soft – for example,

- event responses shall be handled on average within a certain time
- a certain number of event response shall be handled within a specified timeframe
- a specified failure rate is permitted

Because the mathematics required to analyze soft real-time systems is more difficult than for the simpler, hard, case³, it is very common to treat soft real-time systems as hard to simplify the analysis. The result of this approach is an over-design of the system, with typically, an increase in the recurring cost of the system due to the over-designed hardware platform.

In general, as we shall see, functional requirements will be modeled as use cases, descriptive specifications, actions, and message sequences. QoS requirements will be modeled as *constraints* of some kind, applied against one or more functional requirements.

Capturing Requirements with Use Cases

A *use case* is a named coherent collection of related requirements organized around system capability. A use case is a large-scale thing, typically corresponding to 3 to 10 pages of textual requirements. Use cases define little in the way of specific requirements per se, but they serve as a way to organize and manage the detailed requirements. The characteristics of good use cases include:

- focuses on the user or actors perspective of the system (not the implementation of its interfaces or its internals)
- captures a closely related set of requirements
- returns a result visible to one or more actors
- does not reveal or imply system internal structure or implementation
- is independent from other use cases and may be concurrent with them
- consists of a set of messages exchanged between the system and one or more actors (more than just one!)

³ There is the old saying that hard real-time systems are *hard*, but soft real-time systems are *harder*, meaning that the accurate characterization of soft real-time systems is technically much more difficult.

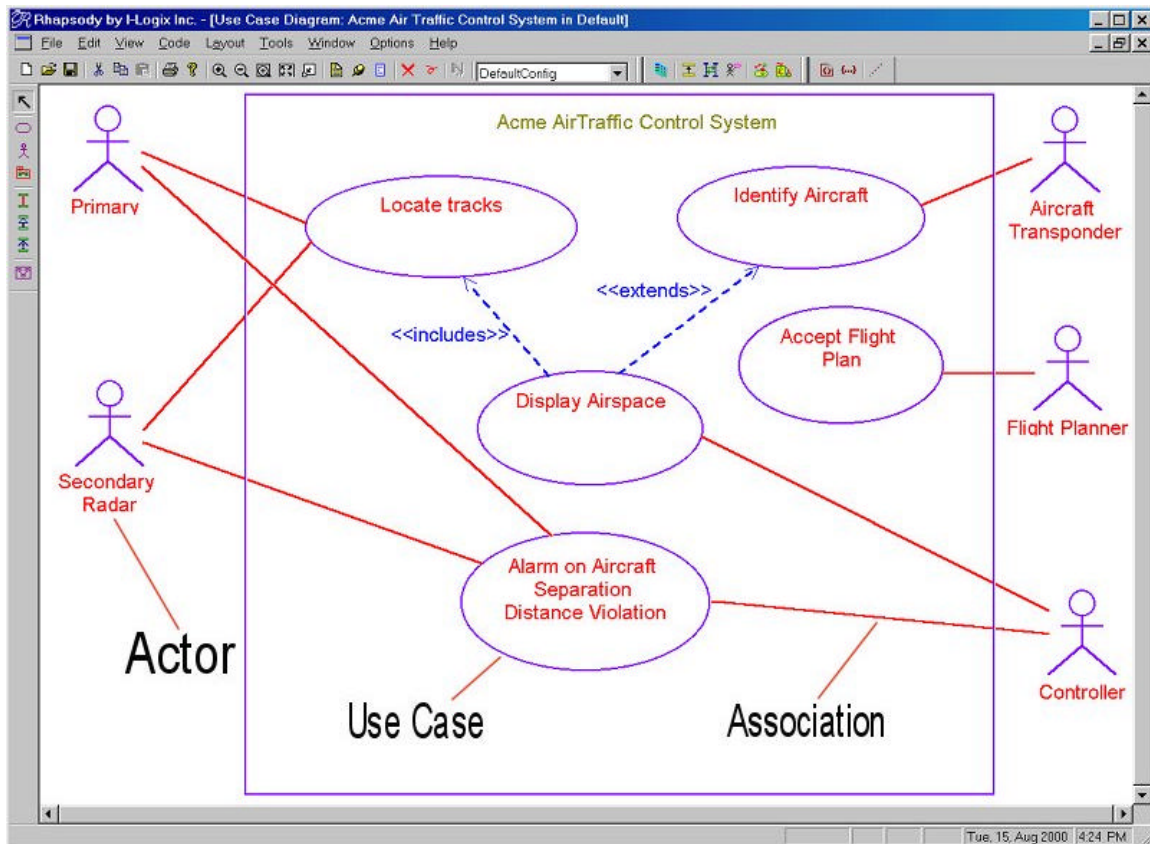


Figure 1: Use Case Diagram

Figure 1 shows a typical use case diagram. The ovals are the use cases. The stick figures are *actors*. An actor is basically an object of some kind, outside the scope of the system, with which the system interacts in ways that we care about. Although the icon for an actor is a stick figure, not all actors are human users of the system. In general, an actor is the thing on the other side of system interface. If the system includes the user interface, then the actor is the human user. If it is a separate software system or legacy hardware device, then that is the actor.

The lines that connect the actors and use cases are *associations*; they permit messages to be exchanged between actors and the system as it fulfills a use case. An open arrowhead on the association can be used when messages only flow in a single direction, but in general, actor-use case associations are bi-directional.

Relationships among use cases can be used – with a caveat. Many newcomers to use case modeling with use these relationships to do a functional decomposition of the system internal structure *which is not what they are for!* The purpose of use case relations is to depict relations among these clumps of requirements. The most common relations are specializations (stereotypes to be specific) of the dependency relation (shown using a

dashed line with an open arrowhead). The «includes» relation means that a larger use case includes a smaller one. For example, a use case for a spacecraft might be “Take a picture of a planet” and another might be “Send information to Earth-side Station”. Executing each of these use case involves rolling the spacecraft to a specific orientation – either to point the camera at the planet or to aim the antenna at the earth. Thus, they could both «includes» a smaller use case “Adjust Attitude”.

«extends» is similar to «includes» includes except that the smaller use case is optional and only used in certain situations. For example, suppose that certain commands sent to a spacecraft could potentially lead to a loss of the spacecraft. You might want user validation and authorization guaranteed before accepting such commands. In this case, the larger “Process Ground Command” use case might be extended by a validate “Validate User”.

Additionally, one use case may be more general or specific than another. For example, there may be multiple ways to do Validate User use case: Validate by Authorization Code, Validate by Fingerprint Scan, Validate by Voice Recognition. Each of these is a more specialized for of the more general Validate User use case. This is shown in Figure 2. Note also that we may shown generalization of actors but not associations (because we don’t model message exchange between actors).

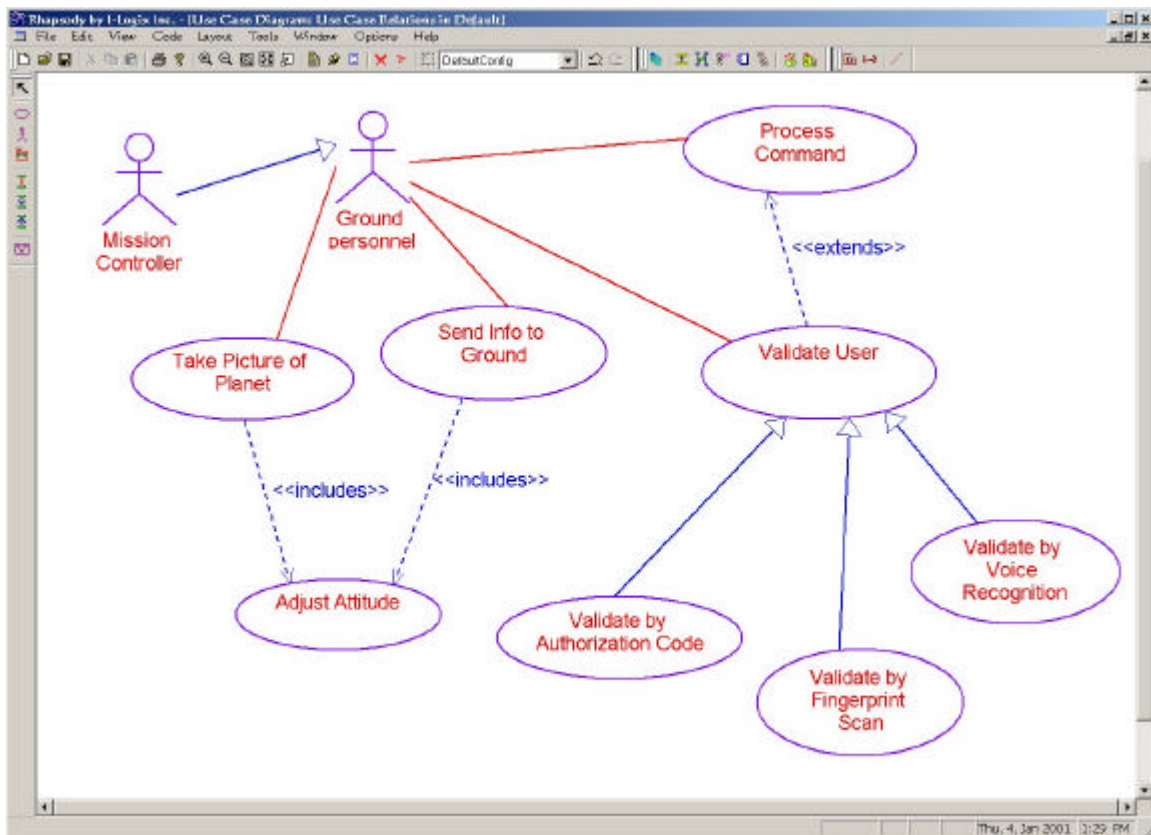


Figure 2: Use Case Relations

We will use these relations in a very specific way when we capture requirements for large complex system. This will be discussed later once we get beyond the basics.

Detailed Requirements

Since a use case is a container of detailed requirements, just providing the name of the use case isn't enough. We need to provide those details. In the ROPES process we call this "detailing the use case."

There are two primary means to detail a use case – by example or by specification. By far the most common is by example. This is done by constructs sets of scenarios of message exchange between the system and the actors that are associated with that use case. There are both advantages and disadvantages of this approach. The advantages include the simplicity of the representation and the ease with which non-technical stakeholders (e.g. all those marketing folks who flunked out of engineering school) can understand how the system operates with respect to the use case. The disadvantages include that a use case can be represented by an *infinite* set of scenarios, so somehow the number that is actually captured must be trimmed down, and that there is typically no way to say "and not" when you give an example. That is, there is no way to specify prohibited behaviors.

Detailing a use case by specification gets around these disadvantages. It provides a single location for the details that applies to each of the infinite set of scenarios. It can also state prohibitions as requirements. On the downside, particularly when formal languages (such as statecharts) are used to specify requirements, a three-digit IQ *is* required, which may disallow certain managers and marketers from understanding the requirements. My recommendation is to generally use *both*, as we will see later.

Scenarios and Message Sequence Charts for Requirements Capture

A *scenario* is a specific path through a use case. The most common representation of a scenario is to use a message sequence chart, as shown in Figure 3. The vertical lines are called *instance lines* and at the system specification level, they represent the actors and either the use case or the system fulfilling the role of the use case. I personally prefer to use the use case because it helps me identify the context of the particular scenario. Note that at this level, we DO NOT include objects inside the system! Looking ahead, later we will *add* internal objects to our scenarios to show how our designs actually meet our requirements, but they should NOT be on the system level use case scenarios. The idea at this point is to capture requirements, not design.

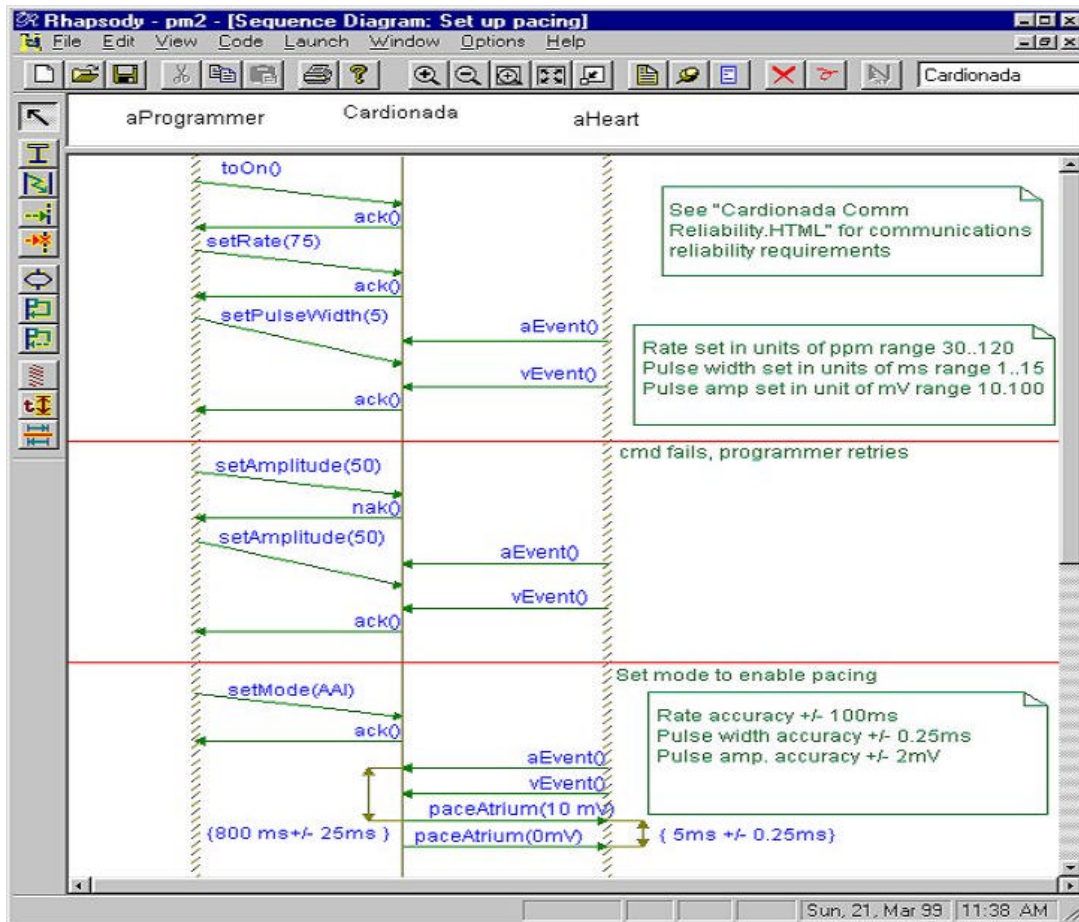


Figure 3: Scenario Example

A typical system might have anywhere from half a dozen to a dozen use cases, and each use case might have half a dozen to several dozen scenarios of interest. Since there is an infinite set from which the scenarios can be drawn, how do you decide which ones to explicitly represent? The ROPES process guideline is simple: Add scenarios to a use case only when they demonstrate or depict at least one or more new requirements. When you can't come up with any scenarios that add a new requirement, then you're done.

Functional requirements are shown on sequence diagrams as ordered message sequences. That is, you're showing that a particular sequence of messages must be allowed. If the order within a message set is unimportant, you can attach a constraint {unordered} to the set of messages. QoS requirements are shown as constraints that attach to the instance lines, individual messages, or to sets of messages. The most common constraints are timeliness constraints, typically applied to an ordered pair of messages. In Figure 3, a timing constraint is shown down at the bottom using a common notation ... a vertical line between two horizontal bars marking points on time on the scenario. Other QoS constraints are shown in note boxes on the right of the diagram.

Specifications for Requirements Capture

The other primary approach to detailing requirements is to do it by *specification*. Either informal or formal languages can be used, or a combination of the two. By informal languages, we usually mean “natural language”, or written specifications. Some authors have elaborate fields used to specify the use case. For example, Schneider and Winters⁴ suggest:

- Use Case Name
- Actors
- (project) Priority
- (project) Status
- Preconditions
- Postconditions
- Extension Points
- Included use cases
- Flow of events
- List of related diagrams (sequence, statechart, activity, etc)
- And more

Of these, I feel only the preconditions and postconditions are really required. The other things are shown using other views (such as the diagrams themselves).

For formal languages, the Unified Modeling Language™ (UML™) provides statecharts and its cousin, the activity chart. Statecharts are most applicable when the use case has states – that is, distinguishable conditions of existence as defined by a set of events or messages accepted, behaviors performed, and reachability of subsequent states. When the use case is in State A, then it accepts a certain set of messages and events, does or can do a certain set of behaviors, and can reach a finite set of other states, and it is distinguishable from other such states in that one or more of these things is different. When an autopilot is executing “Controlling Flight Path”, there are certain things it can do and certain things it cannot when taking off versus when in cruise or landing states.

Activity charts are very similar, since they are just a specialized form of a statechart. Activity charts are used when the primary means to transition from one state to the next is upon *completion* of the actions executed with a state rather than upon receipt of an explicit message or event from somewhere else.

⁴ Schneider and Winters, *Applying Use Cases: A practical guide* Addison-Wesley, 1998

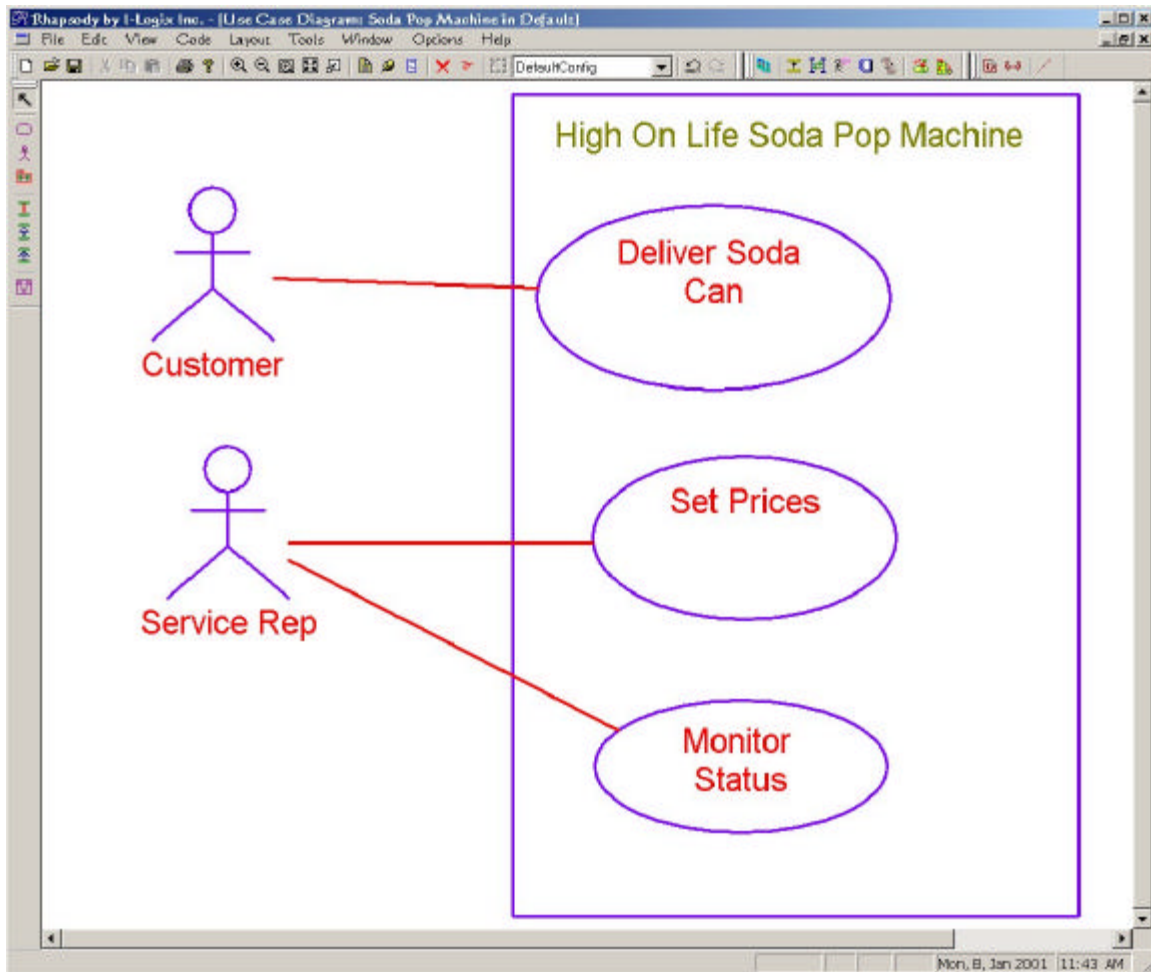


Figure 4: Soda Machine Use Cases

Figure 4 shows a use case diagram for a soda pop machine. It shows two actors (the Customer and the Service Rep) and three use cases. Let's focus on the *Deliver Soda Can* use case. It is difficult to imagine drawing individually all the possible ways in which users might deliver coins and press buttons to get a can of soda from the machine, even without the ability to adjust the price! However, it is relatively straightforward to do so using a statechart, as shown in Figure 5.

The statechart in the figure has only 4 states to manage the transaction of the user inserting coins and selecting the desired flavor of soda⁵. All the actions directly relevant to the specification of the use case are shown on the statechart (although NOT their implementation). Notice also that no internal objects are identified, but some data *are*: specifically, *amt* tracks how much the user has entered, and *AMOUNT_REQUIRED* is the cost of a single can of Soda. There are various operations used within the actions, but it isn't at all implied what objects there are or how they relate to each other.

⁵ For a more complete description of statecharts see either *UML Statecharts*, Embedded Systems Programming, January 1999; or *State Machines and Statecharts Parts 1-3* available for free download at www.ilogix.com.

In fact, a set of objects will *realize* this use case (that's UML-speak for "implement"). All that we can be sure of, is that in any correct design, the objects specified will collectively be able to provide the services as specified by the statechart in Figure 5.

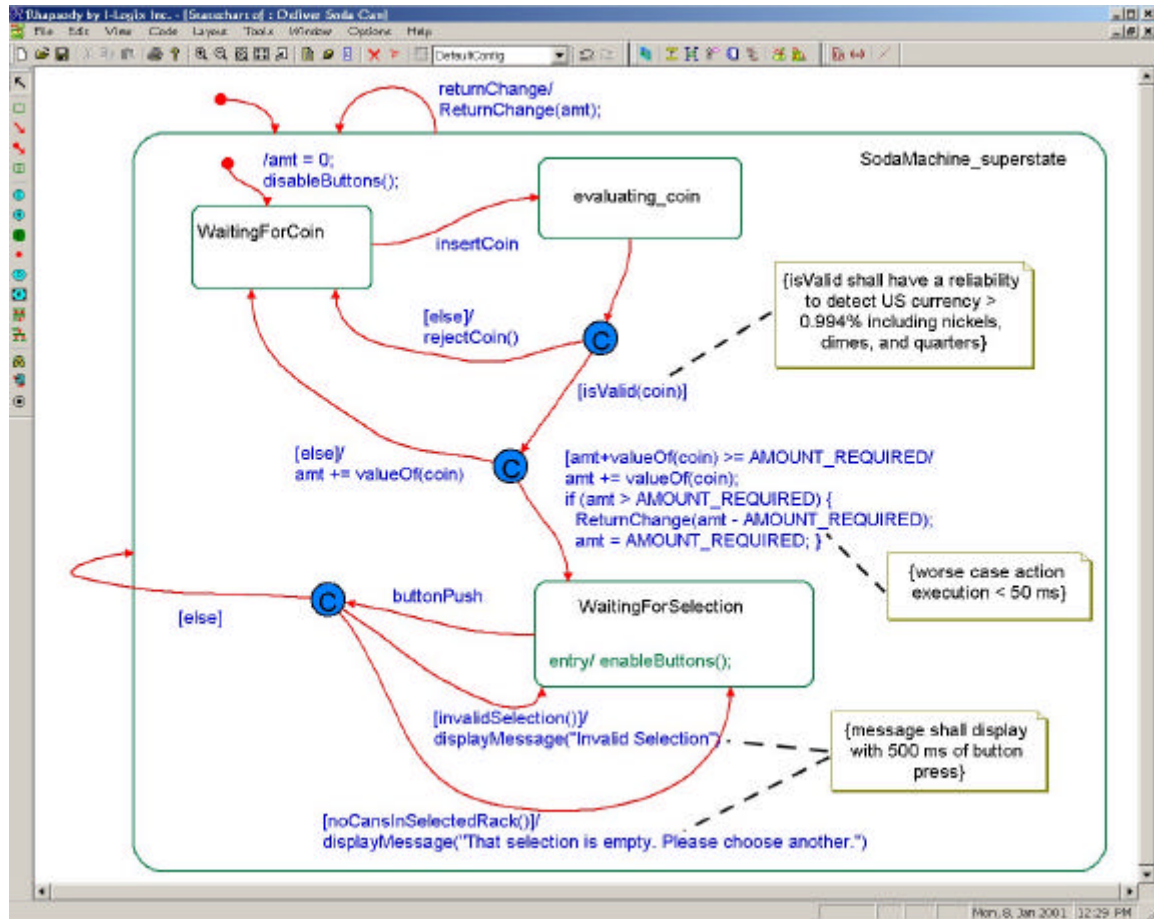


Figure 5: Soda Machine Use Case Statechart

Activity diagrams are a specialized form of statemachine that has been called a “concurrent flowchart”. It is used primarily when the transition from state to state occurs as a result of *completion* of the previous actions rather than the receipt of an explicit signal or event. The activity chart in Figure 6 show how photo printer might work. Once initiated, the machine marches through its states, executing appropriate actions, based on the completion of the previous activity. Once started, two independent activities occur. On one hand, the correct paper tray is loaded and then a sheet of paper is removed. On the other hand, the image is loaded, formatted and scaled. Once all of this is done, then the photo is printed.

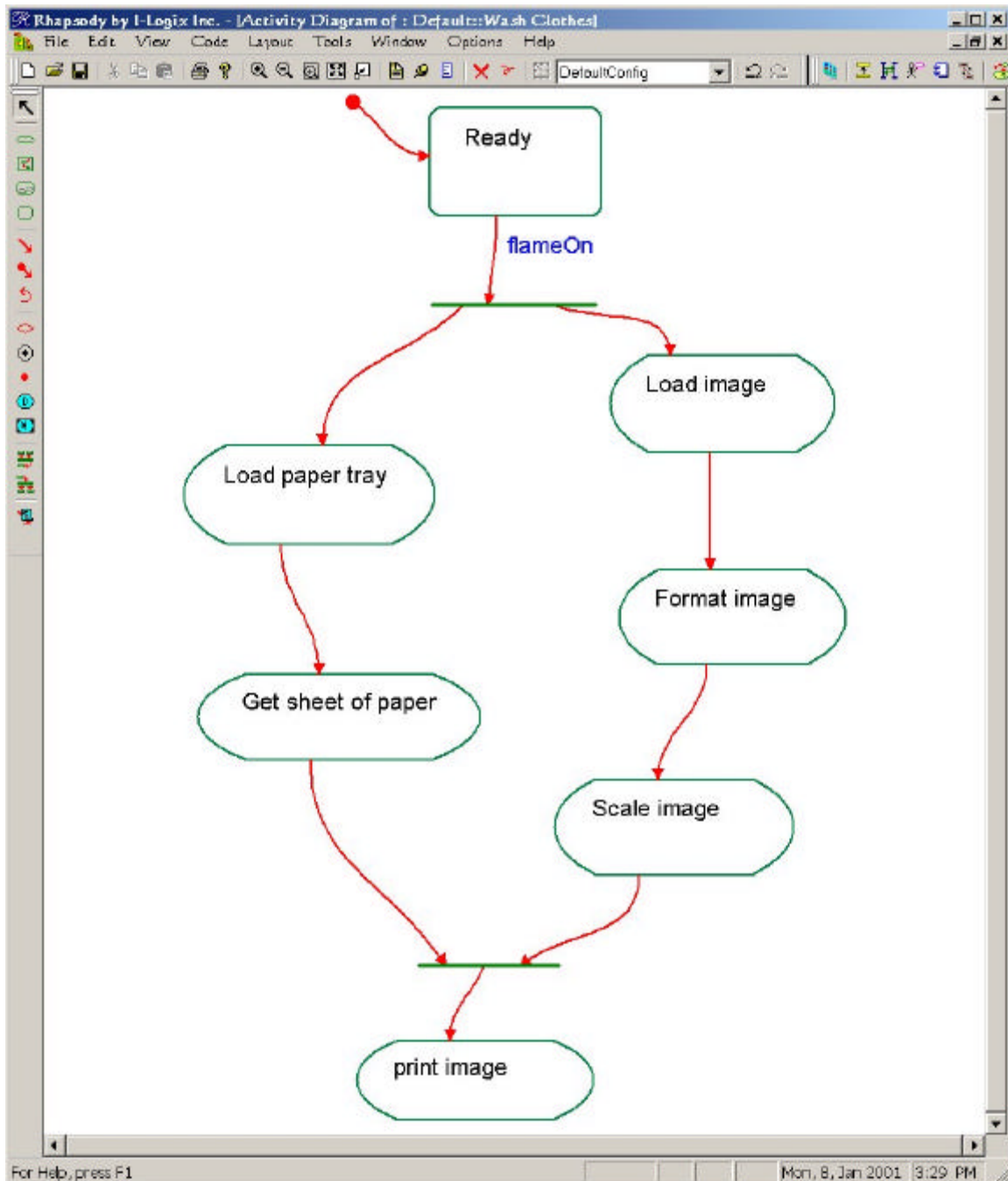


Figure 6: Activity Chart for "Print Image"

In the final analysis, either statecharts or activity diagram can be used for specification of requirements.

Relating Specifications and Scenarios

When you use a formal language, such as statecharts, to specify a use case, you are capturing the *entire infinite set* of scenarios all in one place. A scenario is nothing more

than a particular path through the statechart. For example, Figure 7 shows a one particular scenario represented by this statechart. In this scenario, the cost of the can is 55¢. The customer puts in two quarters and a dime and receives a nickel in change. Then he selects Coke, but there is no coke, and the machine displays a message to that effect. The customer then selects Diet Coke and the system delivers it. Notice that some of the relevant states in the state machine are shown on the use case instance line – this aids the reader in relating the scenario back to the statechart specification.

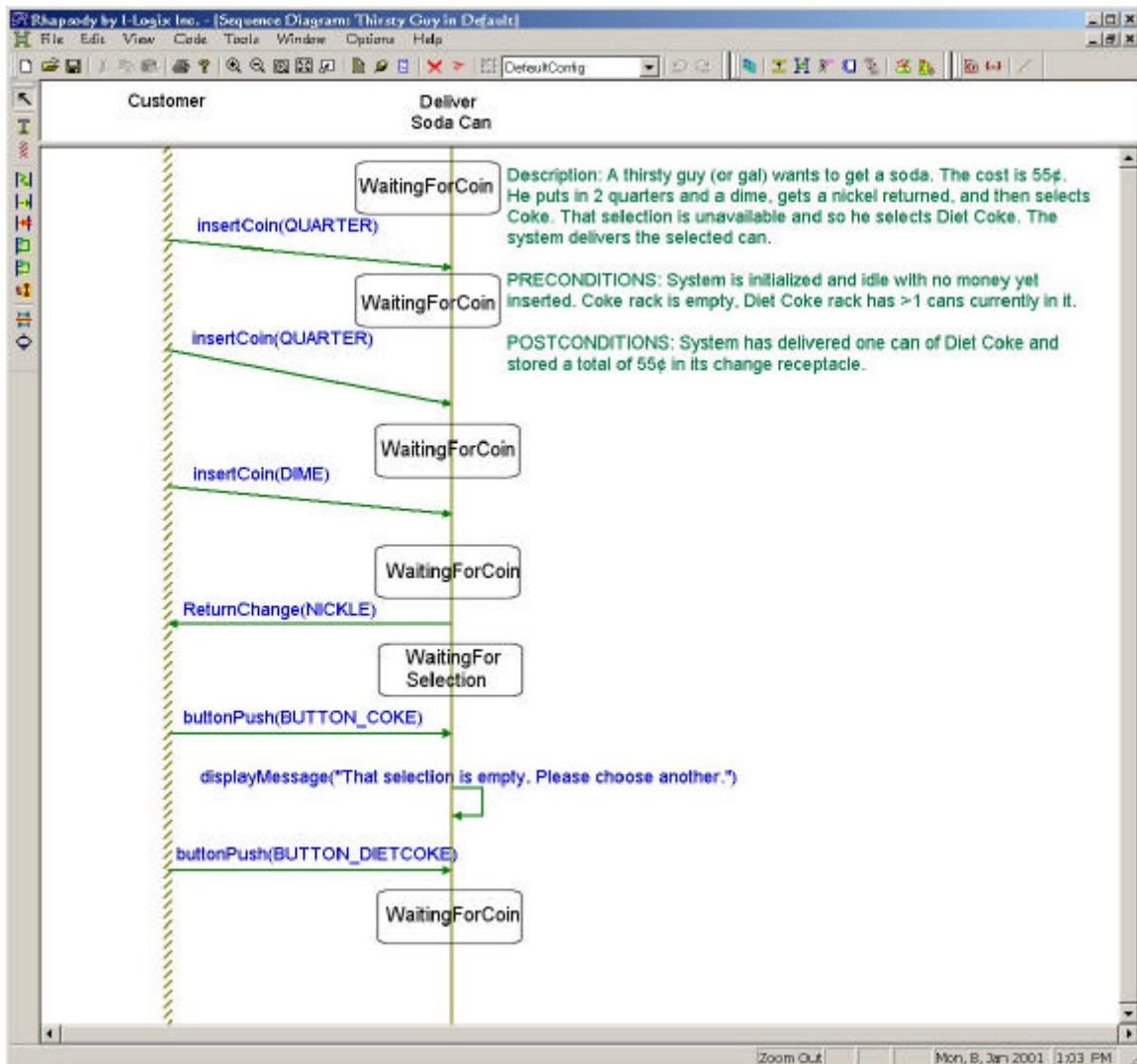


Figure 7: Thirsty Guy Scenario

Of course there are other paths through the statechart – these are simply other scenarios. In general, you will want to construct a set of scenarios from the statechart. You do this by making a different scenario for every different path through the statechart, although you'll only want to do the looping paths once, and representative examples of the concurrent regions (and-states) of the statechart.

Moving from Requirements into Design

As I mentioned earlier, a use case is ultimately realized by a set of objects working together to provide the necessary behavior. This set of objects is called a *collaboration* in UML. It has a specific notation (a dash oval) but it is not commonly used. Most often, the use case collaboration is shown as a class diagram, showing the relevant classes of the objects that participate, at run-time, in the collaboration.

Getting a good set of objects can be tricky and it *is not at all obvious* from the use case model. Some newbies will attempt to map each use case state to separate object and this results in *horrendously bad* object designs. Don't go there!

In the ROPES process, you use object identification strategies to identify the object participating in the collaboration. The ROPES process identifies about a dozen different strategies which, while different and distinct, overlap in the objects they find to a significant degree. Commonly, you will apply 3 or 4 different strategies simultaneously to identify all objects in the collaboration. Using such an approach, one could come up with an object model such as that shown in Figure 8.

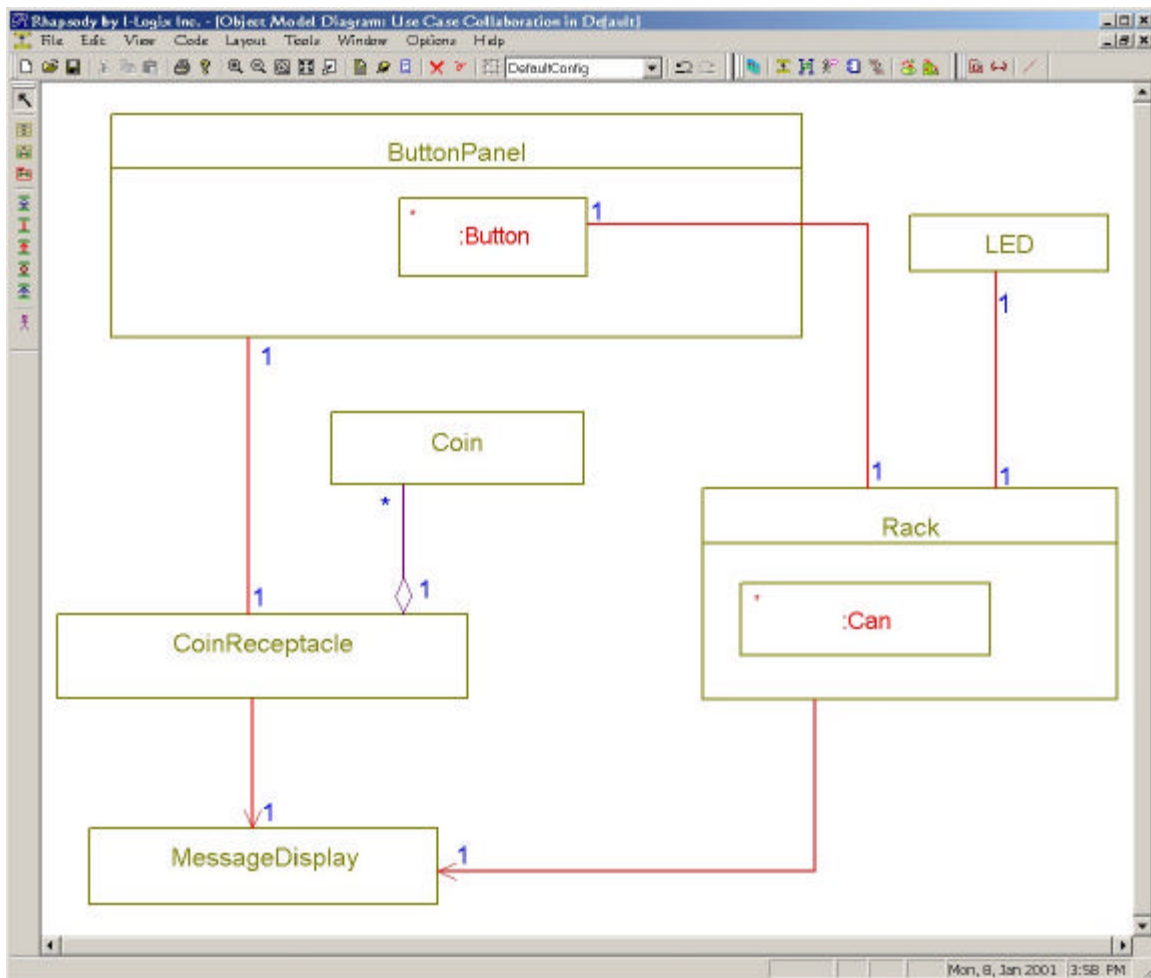


Figure 8: Soda machine Collaboration Class Diagram

The really important question is “How you to ensure and demonstrate that the design model *really does* realize the use case model?” The answer is *through execution*. We evaluate and demonstrate the adequacy of a design model by executing that design model. Specifically, we want to execute the very same scenarios we used to state requirements using the newly added design elements. If the design can execute all of the requirements then we’ve done a good job. If it can’t, then we need to fix our design model.

How we do this in the ROPES process is first we come up with a set of scenarios at the system use case level where the players are the actors and The System (or the system playing the role of a specific use case). To test the adequacy of a design, we do *the same scenarios*, but now we add the design elements to the scenario and show the exact exchange of messages among them necessary to fully realize the behavior.

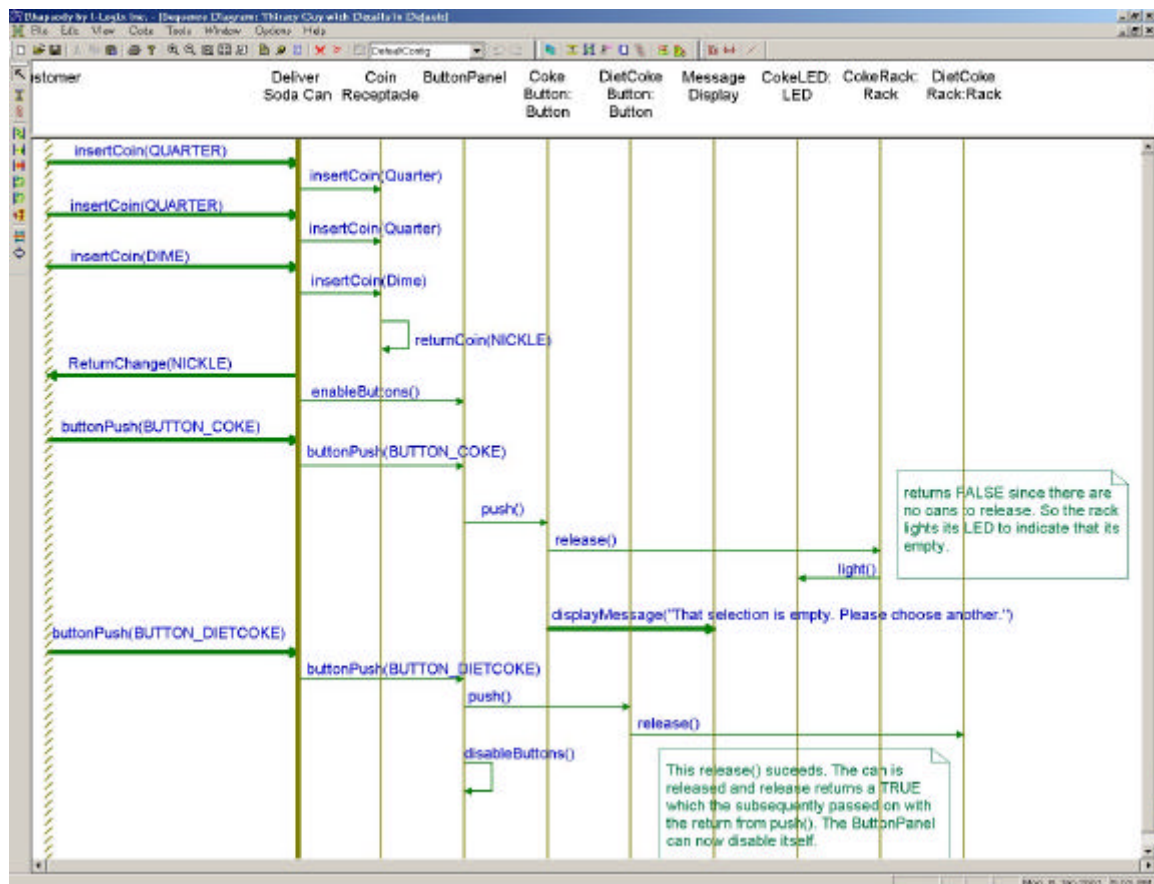


Figure 9: Thirsty Guy Scenario with design detail

Figure 9 is the very same scenario as shown in Figure 7, but we’ve added the collaborative elements from the class diagram (Figure 8)⁶. We can walk through the scenario and see how the objects inside the system collaborate to realize that particular

⁶ Remember that scenarios show *instance* not *types* (i.e. classes). So the scenario has two different Rack instances, for example, but the class diagram shows only a single Rack class.

scenario. Notice that we kept the Deliver Soda Case use case instance line. We do this because it eases the traceability among diagrams that depict the *same* scenario, but at different levels of abstraction, such as the scenarios shown here. For the same reason, I like to make the lines from the previous level of abstraction thicker, as it shows more easily what's been added to at this level of abstraction.

This approach of validating via execution can be applied at any level of abstraction. If we have a highly complex system with subsystem, components, and composite classes, as we add more and more detail we can always be sure that we're doing a good job when at the specified level of abstraction it executes all of the scenarios defined for the use case.

This approach means that even if you use statecharts to specify requirements, you will *also* draw scenarios. Since statecharts can have loops and concurrent regions, a statechart can represent an infinite set of scenarios. Which scenarios should you draw and trace through the levels of design abstraction? Again, the ROPES process has an answer: draw a scenario for every non-looping path through the or-states and each looping path exactly once; for concurrent regions do some representative interleaving of the concurrent regions.

Scaling to Large Complex Systems

One of the challenges methodologies face is coming up with approaches that are not too onerous to be used on small projects, but scale up to large problems. Large-scale systems are, after all, where you REALLY need a good methodology to help you out. For large systems, it is common to break them down into large subsystems, each of which may have a separate development team. When this is the case, the ROPES process provides a Systems Engineering Phase, occurring immediately following the Requirements Analysis phase. The system engineers take the requirements analysis (represented by the use cases and their supporting details), construct a subsystem model, map the use cases into the subsystems, define the subsystem-subsystem interfaces, and do hardware-software breakdown and responsibility assignment.

A subsystem is a large-scale organization of run-time elements (e.g. objects). Items are placed in a subsystem using a simple criterion: *common run-time purpose*. For example, a spacecraft might be broken down into Avionics, Power, Guidance and Navigation, Thermal, Launch, and Communications subsystems. Each of these subsystems itself has lots of sub-elements inside of it. The objects that go into the Power subsystem, for example, all help it deal with managing power; this might include objects like switches, relays, batteries, waveforms (to monitor power fluctuations), messages (to allow other subsystems to request power), and so on.

The hard part for the Power subsystem team is to know what their requirements are. It is the job of the system engineer to construct a good subsystem model and help the team understand exactly what their role and requirements are. So, how do we know if a

subsystem organization is good? Class?

Don't all raise your hands at once.

That's right, we *execute* the subsystems as they collaborate to realize the system-level use cases. Since subsystems are just *really big* objects, at the level-1 abstraction layer (level-0 is the system level), the subsystems collaborate together to realize the use cases. Want to do a use case like "Achieve Stable Orbit"? Then the Avionics, Power, Guidance and Navigations, Thermal, Launch, and Communications subsystem work together to achieve this. So to see if we have a good subsystem model, we show *how they work together* to realize the system level use cases. Pretty slick huh?

As we "drill down" and design the subsystems, we can then refine the system level scenarios by adding increasing levels of detail to our scenarios and show that they *continue* to meet the requirements.

In terms of identifying the requirements for the subsystems (so the teams can work relatively independently), the system engineer decomposes the use cases through the «includes» and «extends» relationships to take a system level use case and construct a set of subsystem-level use cases from it. The subsystem level use cases (also known as level-1 use cases) need to be a complete set for a given system use case (also known as a level-0 use case); that is, they should span ALL of the requirements specified in the level-0 use case. How do you know when you're done decomposing the use cases? Each level-1 use case maps to a *single* subsystem. When that's true, from some number of level-0 use cases you get a set of level-1 use cases for each subsystem. That means that each subsystem team now *knows* what their requirements are – they have a comprehensive set of level-1 use cases that describe them.

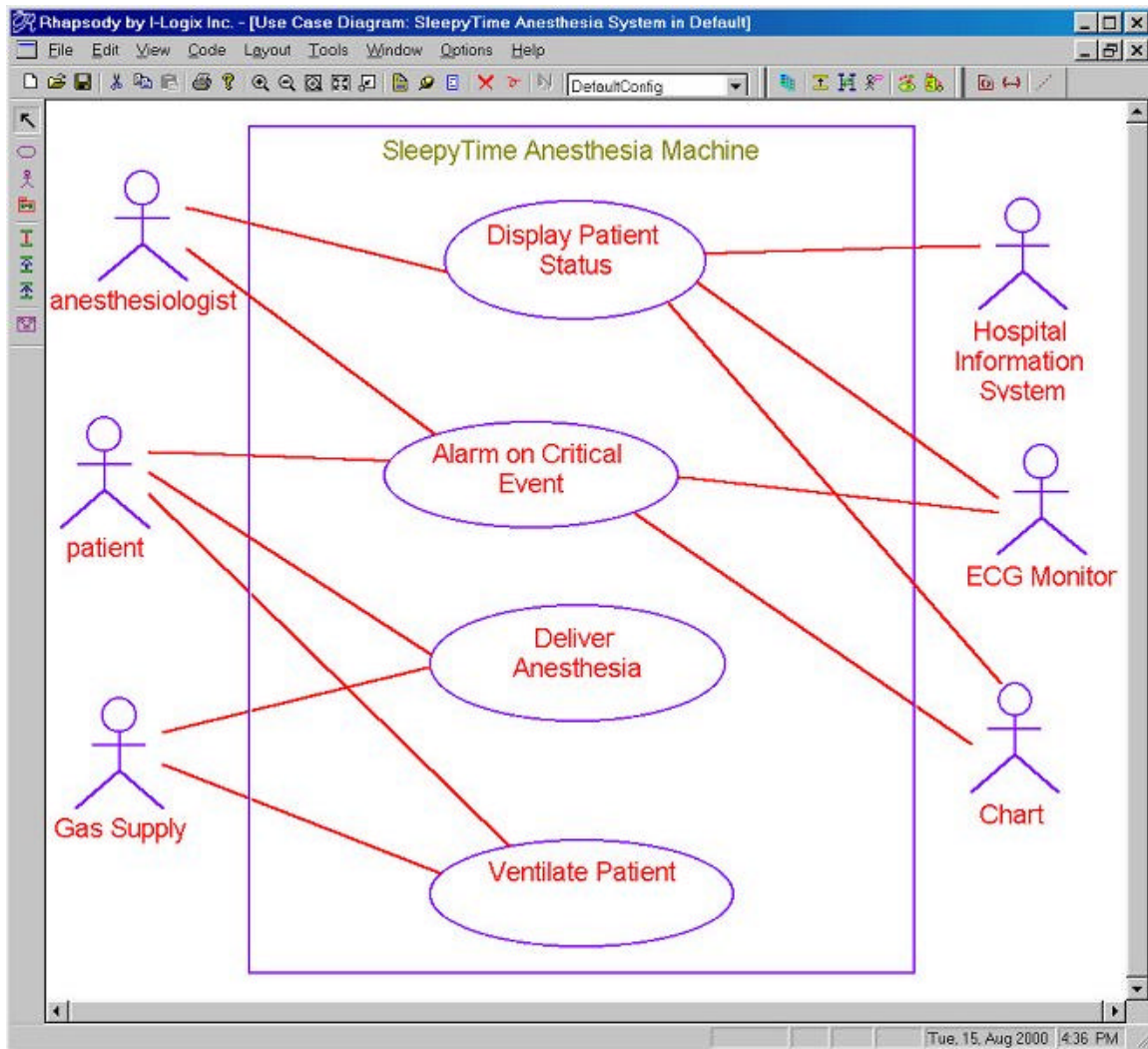


Figure 10: Anesthesia Machine Use Cases

Figure 10 shows some use cases for an anesthesia machine while Figure 11 shows the subsystems of the anesthesia machine. You can see that I've used classes with the stereotype «subsystem». You can see that the subsystems have associations with actors and with peer subsystems. These associations can be thought of as conduits that allow message transfer and collaboration among the players.

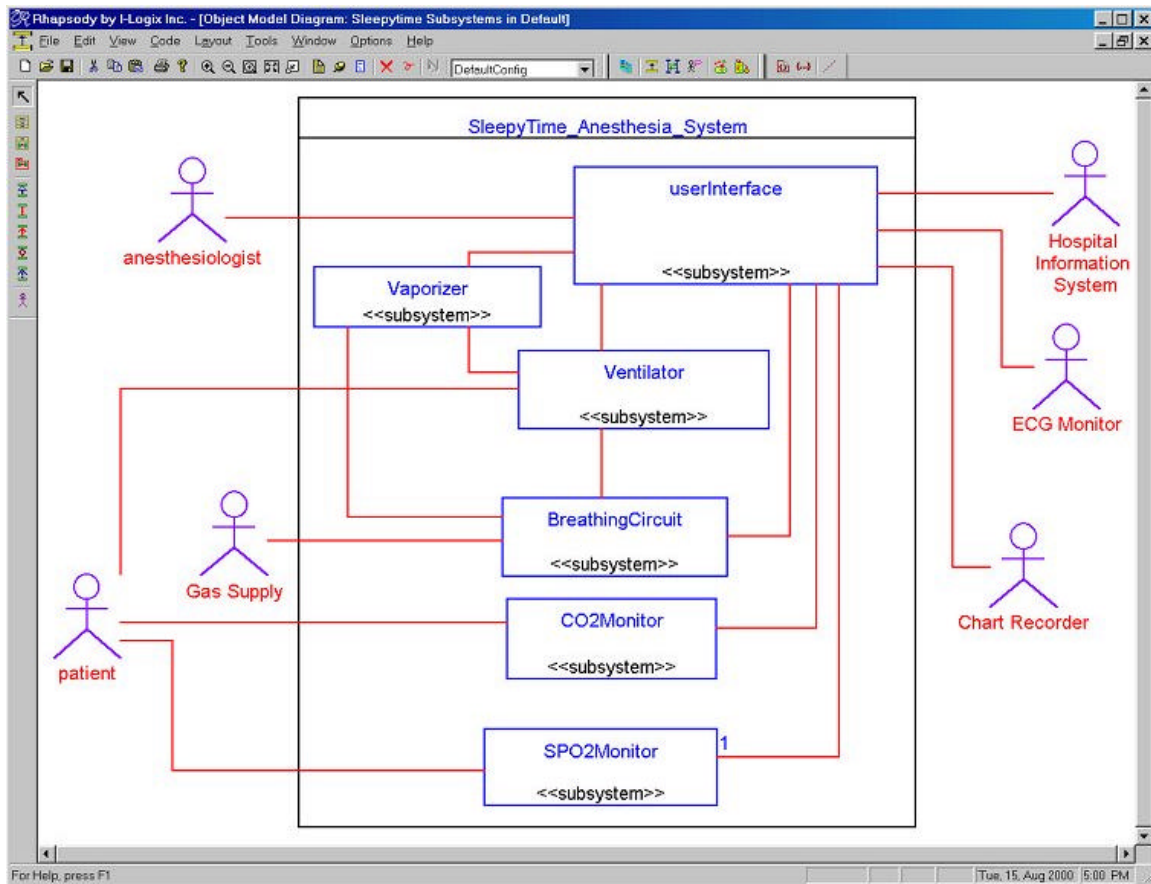


Figure 11: Anesthesia Machine Subsystems

Figure 12 is a use case diagram for one of the subsystems – in this case, the Use Interface subsystem. These are the level-1 use cases, derived from the level-0 use cases shown in Figure 10. Note that there are more actors on this diagram than in the original. This is because from the perspective of this subsystem, all the peer subsystems are actors. They are internal to the system, but external to the scope of concern of the User Interface subsystem. We call these *internal actors*. I like to use a naming convention which is the name of the subsystem prefaced with a letter 'a'.

From here, the individual subsystem teams can work more or less independently, coming together for the integration of the prototypes, according to the project plan.

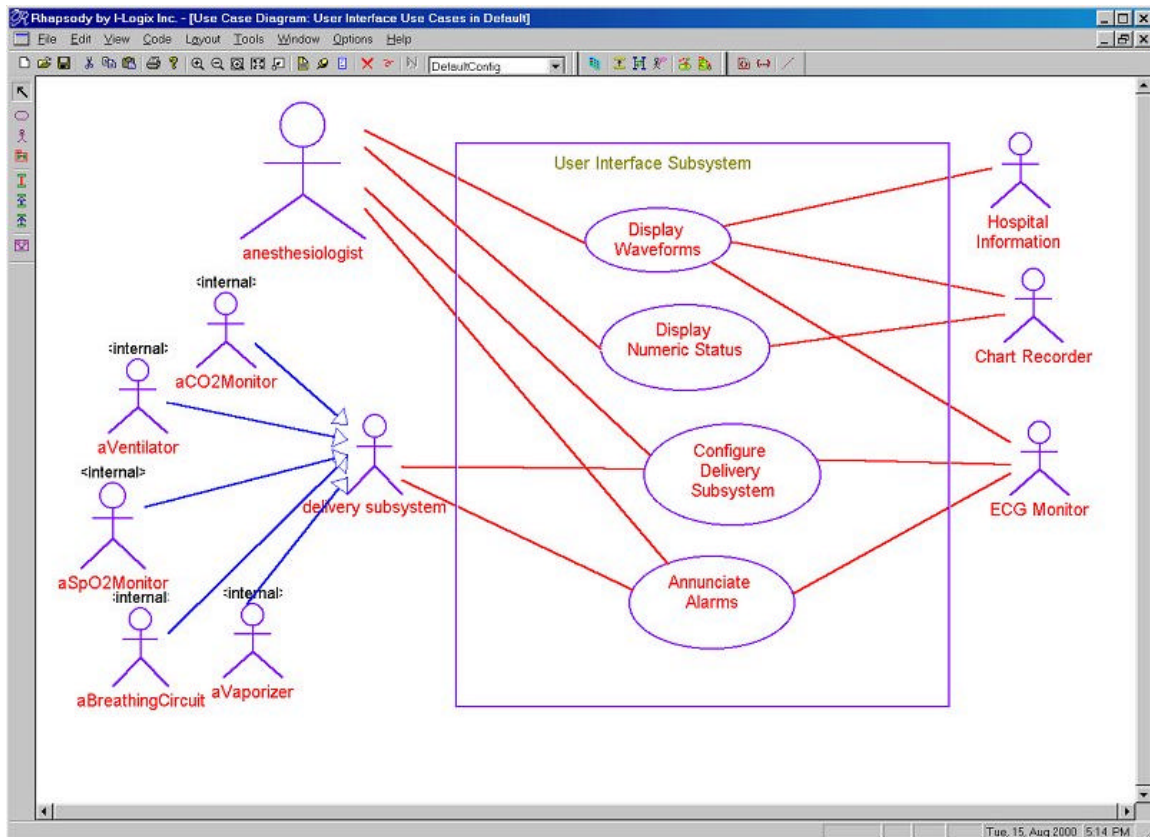


Figure 12: Subsystem Use cases

As the design of the subsystems progress, the scenarios of the level-1 use cases are further elaborated to ensure that the pieces continue to meet the original and derived requirements. If it happens (as it often does) that the interfaces between the subsystems are found to be inadequate later in design, the interface is thawed from configuration management, renegotiated among the subsystem teams, and then refrozen.

Summary

We've discussed an approach to the capturing of requirements for real-time and embedded systems that has been shown to be effective in projects large and small. Use cases, as defined in the UML are used as an organizing principle for managing requirements, and a combination of scenarios (represented using sequence charts) and specifications (represented using statecharts and activity diagrams) to capture the detailed requirements. For large systems, a system engineering phase captures the subsystem architecture and maps the level-0 use cases onto it. This allows the teams to go off and work independently with some level of assurance that their subsystems will properly fit into the system when they're all done.

About the Author

Bruce Powel Douglass has over 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 and the UML World Conference, OOPSLA, and Software Development Magazine. Bruce has written more many computer publications including Embedded Systems programming, Software Development, Dr. Dobbs, and others. He has authored several books, including "Real-Time UML Second Edition: Efficient Objects for Embedded Systems" (Addison-Wesley, 1999) and "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns" (Addison-Wesley, 1999). He worked with methodologists at Rational and other companies on the UML specification. He is a cochair for the Real-Time Analysis and Design Working Group in the OMG™ standards organization, focusing on the application of UML in real-time and embedded systems. He is the Chief Evangelist at I-Logix, a leading real-time object-oriented and structured systems design automation tool vendor. He can be reached at bpd@ilogix.com.

I-Logix and the I-Logix logos are trademarks of I-Logix Inc. OMG marks and logos are trademarks or registered trademarks, service marks and/or certification marks of Object Management Group, Inc. registered in the United States.