Regular Paper

Using Objected-Oriented and UML Tools for Hardware Design: A Case Study

Terry Bahill[1,] * and Jesse Daniels[2]

[1]*Systems and Industrial Engineering, University of Arizona, Tucson,* AZ 85721-0020

[2]BAE SYSTEMS, *San Diego,* CA 92127

Received March 21, 2002; Accepted October 8, 2002

ABSTRACT

This paper argues for the adoption of object-oriented design and UML tools for nonsoftware designs, i.e., systems, hardware and algorithms: This is a controversial position. It presents a case study, the design of a heating, ventilation, and air conditioning system, using UML tools. This case study also shows the incremental elaboration used to progress from the requirements model, to the analysis model, to the design model, etc. The paper finally discusses some difficulties that must be overcome in order to apply UML tools to system designs. © 2002 Wiley Periodicals, Inc. Syst Eng 6: 28–48, 2003.

Key words: unified systems engineering process; unified modeling language; UML; design tools; object-oriented design; object-oriented systems engineering; engineering communication

1. INTRODUCTION

Most 20th century systems were primarily mechanical hardware systems. Most 21st century systems will be primarily electronic software systems. In moving from the 20th to the 21st century, systems engineering tools should evolve. It would be nice if these new tools were used by both systems and software engineers. It would also be nice if systems and software engineers spoke the same language.

Software engineers traditionally have been about 5 years ahead of systems engineers in creating computer tools to help them do their jobs. Fifteen years ago software engineers developed CASE tools; 10 years ago

*Author to whom all correspondence should be addressed (e-mail: terry@sie.arizona.edu).

Systems Engineering, Vol. 6, No. 1, 2003

systems engineers developed requirements tools like RDD-100, DOORS, Slate, RTM, Excel etc. Within the last 5 years, they created the Unified Modeling Language (UML), a Unified Software Development Process [Jacobson, Booch, and Rumbaugh, 1999] and the Rational Unified Process [RUP, 2002]. We think it is time for systems engineers to adopt some of these tools.

To design systems, systems engineers use requirements, specifications, block diagrams, scenarios, timelines, functional decomposition, functional flow block diagrams, finite state machine diagrams, verification matrices, etc. [Bahill et al., 1998]. There is no standard usage, however, and these tools do not link together. The software folks have taken these tools, improved them, and made them interact. Then, they created a standard for using these tools, a standard that most universities are teaching and hundreds of companies are using.

The Unified Modeling Language™ (UML) is a standard for specifying, visualizing, constructing and documenting system designs. It encourages hierarchical designs. The UML is a collection of tools for communicating system designs [Fowler and Scott, 2000].

The UML supports about nine views of a system. There are two important reasons why so many views are used. First, a system cannot be described with only one view: multiple views are necessary. Second, much of the information in a UML model appears in several views. This produces double-checking and incremental growth of the models. However, using UML support tools, you only have to change an item in one place, and the change automatically propagates throughout all the views.

1.1. Problem Statement: The Deficiency

Systems engineering design tools are old-fashioned, they do not link together, and they are used differently by different people. Systems and software engineers do not communicate well. Some systems engineers are still using waterfall processes. Flow-down of requirements often causes excess design margins and expensive redesigns. Late changes in requirements cause costly redesign.

1.2. Unified Systems Engineering Process

The Unified Systems Engineering Process [based on Bahill and Gissing, 1998; Jacobson, Booch, and Rumbaugh, 1999] is iterative and incremental. It does not use a waterfall process. In a waterfall process, the requirements are flowed down from top-level requirements. This produces multiple analyses of requirements at each new level of detail. It causes design margins to increase at each new level, creating requirements that are more stringent than necessary. Finally, if the requirements are not achievable at the lowest level, specifications and designs must be radically changed.

Instead, the Unified Systems Engineering Process uses vertical requirements development in which each requirement (use case) is developed through iterative increments: the requirements model, the analysis model, the design model, etc. The behavioral requirements (use cases) and supplemental requirements (performance, reliability, usability, maintainability requirements) are analyzed and expanded during each iteration. Each iteration involves push back on unachievable requirements and a series of negotiations and tradeoffs between systems engineers, designers and stakeholders.

The Unified Systems Engineering Process, shown in Figure 1, has time running from left to right through the system lifecycle: Inception, Elaboration, Construction, Transition, and Operation, Retirement and Replacement. The orthogonal direction shows the development of the design though the various models: Requirements, Analysis, Design, Implementation, Verification and Operations.

The Unified Systems Engineering Process has five major themes: (1) requirements (get them early and get them right, but plan for change), (2) architecture (design the interfaces early), (3) use and reuse components, (4) plan frequent small iterations, and (5) manage risk (start risk analysis early and develop high-risk subsystems first).

1.3. Advantages

The Unified Systems Engineering Process is based on what the software community calls object-oriented development. Douglas [2000] described the following advantages of object-oriented development: (1) The same modeling tools and views are used in all phases of development. Therefore, there is a consistency between the models mentioned in Figure 1. This is doubly an advantage when systems engineers hand over a design to software engineers. (2) Object-oriented modeling has a strong cohesion among data items and the functions that manipulate them. This improves problem domain abstraction. (3) Because many object-oriented abstractions are based on the real world, they are more stable. Accommodating changing requirements may necessitate the addition or removal of objects, and the reassignment of responsibilities among objects, but it is less likely to require a total reconstruction of the system. (4) The techniques of generalization and refinement improve prospects for reuse. (5) The abstraction and encapsulation properties of object-oriented models allow a looser coupling between components and provides further resilience to change. This, along with the

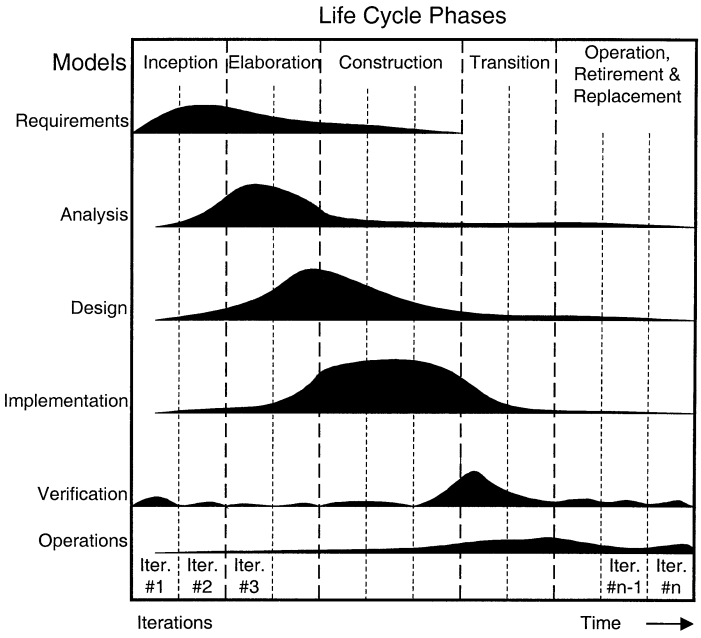


Figure 1. The Unified Systems Engineering Process.

previously mentioned consistency of views from analysis to design to implementation, creates better scalability to large systems. (6) Better abstraction, encapsulation, and definition of interfaces improves reliability, because you can control how the components interact. (7) Object-oriented systems are inherently concurrent, which enhances the ability to model complex, real-world systems where concurrency is also inherent.

In addition, Marasco [2002] emphasizes that the UML tools are universal and graphical. A picture is worth a thousand words, because humans are visual creatures. Figure 2 [based on Marasco, 2002] shows some graphical symbols from ancient disciplines. They

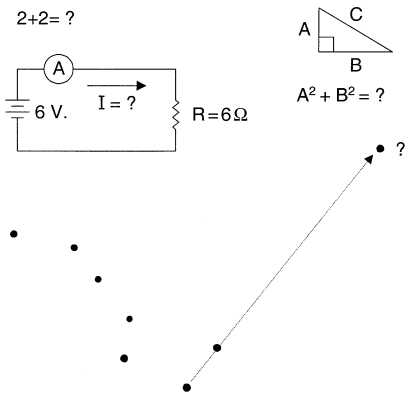


Figure 2. These four questions are posed nonverbally.

are understandable without words of explanation. They communicate among diverse peoples. Someday the UML tools might be just as universal.

1.4. The Controversy

In this paper, we argue for the adoption of UML tools for hardware and algorithm design. Others share our sentiments; see, for example, Ogren [2000], Bienvenu, Shin, and Levis [2000], and the OMG Systems Engineering Domain Special Interest Group (OMG SE DSIG) at http://sysengr.omg.org. But this proposition is controversial. Many systems engineers argue vociferously that systems engineers should not use UML tools. Time will tell.

When Raytheon submitted their preliminary proposal for the DD(X) program, the customer said their use of object-oriented design and UML tools for system design was a "weakness." Eighteen months later in May 2002 when they won the proposal, the customer said that their use of object-oriented design and UML tools for system design was a "major strength." It had systems and software engineers talking the same language [Brian Wells, personal communication, 2002].

In 2001, Lockheed Martin won the Joint Strike Fighter (JSF) program. Early on they noted a government suggestion in the RFP that object-oriented design and UML tools be used for the system design. Lockheed Martin's proposal was substantially superior in this aspect.

1.5. What the UML Is Not

The UML design tools do not replace *all* other tools. You still need Mathematica, MatLab, Maple, Microsoft Project, Kalman filters, Minitab, etc. Also, you still have to create measures of effectiveness and investigate alternative designs so that you can do tradeoff studies, define system boundaries, run simulations, decide on "make or buy," do configuration management and perform sensitivity analyses. The UML does not supplant Linear Systems Theory or Statistics. In selecting actors, classes, and functions, the systems engineer must still avoid using preconceived solutions as statements of the problem. The UML does not fill the shoes of intelligence or creativity. A fool with a tool is still a fool. You can bungle a design using UML just as easily as any other tool.

The rest of this paper is a case study illustrating the Unified Systems Engineering Process and some of the UML tools. A reader unfamiliar with the UML might want to read the Appendix first.

2. CASE STUDY: AN HVAC SYSTEM

2.1. Case Study Problem Statement

Design a heating, ventilation and air conditioning (HVAC) system for a typical Tucson family house. Assume that the outside air temperature varies between 25ºF and 115ºF (–4ºC and 46°C) and that the room temperature is supposed to stay between 70ºF and 73ºF (21ºC and 23°C). The primary components, of this HVAC system are the heater, air conditioner, fan, thermostat, and controller. Please note that Section 3 of this paper is a glossary of HVAC specific terms.

2.2. The Requirements Model

The requirements model establishes what the system should do and defines the boundaries of the system. It should start with an investigation of the highest risk aspects of the proposed system. High-risk aspects are identified by analyzing behavioral threads through the system, possibly constrained by performance measures, which are flagged as high priority by the customer. New or advanced technology and complex algorithms may also contribute to the risk. The requirements associated with such characteristics of the system should be thoroughly validated. The requirements model starts with a skeleton of these high-risk functions. The essence of this model, and the fundamental basis of the Unified Systems Engineering Process, is the use case.

2.2.1. Use Cases

A use case is an abstraction of a required function of a system. A use case is described by a sequence of interactions between one or more actors and the system. [Cockburn, 2001; Kulak and Guiney, 2000].

2.2.1.1. Use Case Example

Name: Regulate Temperature.
Brief description: Maintain the room temperature (roomTemp) between lowerThreshold (see Rule1) and upperThreshold (see Rule2) degrees Fahrenheit using a Heater and an Air Conditioner.
Scope: An HVAC controller for a typical Tucson family house.
Primary actor: Home Owner.
Supporting actors: Heater, Air Conditioner, and Environment.
Precondition: All equipment is in good working condition.
Main Success Scenario:
1. The Home Owner turns on the Controller and sets the temperature thresholds.
2a. The roomTemp exceeds upperThreshold.
3. The system turns on the Air Conditioner.
4. The roomTemp drops below upperThreshold.
5. The system turns off the Air Conditioner [repeat at step 2].
6. Home Owner turns off the Controller

Anchored Alternate Flow:
2b. The roomTemp drops below lowerThreshold.
2b1. The system turns on the Heater.
2b2. The roomTemp exceeds lowerThreshold.
2b3. The system turns off the Heater [repeat at step 2].

Rules:
Rule1: lowerThreshold default value is 70ºF.
Rule2: upperThreshold default value is 73ºF.
Author: Terry Bahill
Date: January 1, 2002

This is a simplistic example of a use-case description. This use case describes how the primary actor, Home Owner, expects to use the Controller and also how the Home Owner expects the system to operate as a result of the Home Owner's interactions with the system. The engineer designing the Controller for this system would accept this use case as input to understand how the controller is to be used by the Home Owner, and how the Home Owner expects the system to operate. Then, in accordance with the Unified Systems Engineering Process, the controller designer would write lower level use cases based on this one that describe the exact algorithm for controlling the interfaces between the Air Conditioner and Heater, as well

as specify the protocol for communicating with these external systems (to the Controller designer, anyway).

This use case would also be very important to potential users and user interface designers. Potential users of the system can provide input and suggestions to the operation of the system through a use case such as this one. The use case serves as a communication tool between the systems engineers and the user community. User interface designers can extract information that helps them design an efficient interface based on the usage of the system as described in the use case.

There is no standard that specifies which fields should be in a use-case description. Your minimal set should be based on your company requirements' template. The number of fields and the detail in each field increases as the design progress from the requirements model to the analysis model to the design model to the implementation model. Bahill's use case template is available at http://www.sie.arizona.edu/sysengr/slides/templates.doc.

2.2.1.2. Definitions

A *use case* is an abstraction of a required function of a system. A use case produces an observable result of value to the user. A typical system will have many use cases each of which satisfies a goal of a particular user. Each use case is described by a sequence of interactions between one or more actors and the system. This sequence of interactions is described in a *use case description* and the relationships between the system and its actors are portrayed in a *use-case diagram*.

A *use case description* is written in a natural language. It is written as a sequence of numbered steps with a main success scenario and alternate flows. It contains the name of the use case, a brief description and the sequence of steps: it may also contain its level, the scope, the primary and supporting actors, preconditions, postconditions, the trigger, the priority, the frequency of use, and the owner/author. The steps should be written in clear, concise, present tense, active voice.

Actors reflect roles of things outside the system that interact with the system. Primary actors initiate the functions described by use cases. Supporting (or secondary) actors are used by the system. They are not a part of the system, and thus cannot be designed or easily altered. They often represent external systems or commercial off the shelf (CotS) components.

2.2.1.3. Another HVAC Use Case

Name: Display System Status.

Brief description: The system shall monitor the health of each component in the system and display the status of the complete system.

Scope: An HVAC controller for a typical Tucson family house.

Primary actor: Home Owner.

Frequency: System Status shall be displayed continuously.

Main Success Scenario:

1. The Home Owner asks the system for its status.
2. Each component in the system reports its status to the Controller.
3. The system accepts this information and updates the system status display.
4. The Home Owner observes the System Status [repeat at step 1].

Author: Terry Bahill

Date: January 1, 2002

Most fields in a use-case description are optional: Use them when they are useful; do not use them when they are not. An exception is the scope: The scope is almost always useful. In this example, the design would be quite different if the scope were an HVAC system for a university or an automobile. The scope is the same for each use case in this household HVAC system. This is not true in general, and defining the scope, or the system boundaries, is a difficult task.

2.2.2. A Use-Case Diagram

A use-case diagram, as shown in Figure 3, shows the relationships between the use cases and the actors. The Unified Systems Engineering Process is architecture based. It also requires investigation of alternatives, risk analyses and business analysis.

2.2.3. Other Important Parts of the Requirements Model

Candidate Architecture. We plan to use a natural gas heater and an electric air conditioner. There will be one central thermostat.

Alternatives. An important task is investigating alternative designs. For our HVAC system, we will also consider electric heat, wood, oil, coal, heat pumps, solar panels, three-phase electricity, steam, hot or chilled water systems, fans, ice farms and cooling towers. In our tradeoff study, we will use criteria like total life cycle cost and performance. A possible performance measure of effectiveness would be how quickly the system can heat or cool the house by 10ºF.

Preliminary Risk Analysis. At this point, we have identified three major risks: (1) The capacity might be inadequate to heat or cool the house. We will investigate typical equipment in a dozen Tucson homes and also use professional guidelines in selecting equipment. (2) The air condi-

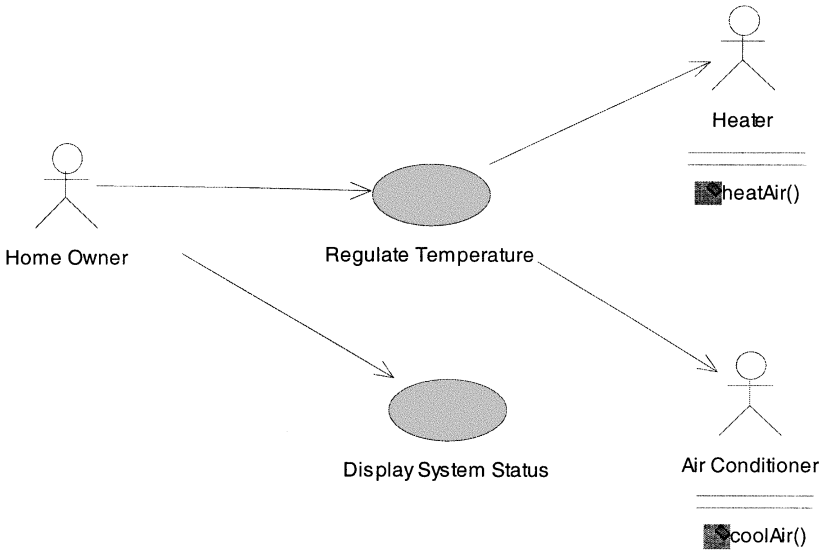


Figure 3. Requirements-model use-case diagram.

tioner could be too expensive for the homeowner. We will be cognizant of his and her budgets. (3) If the air conditioner were to turn on and off every minute, it would be very annoying.

Initial Business Case. On a hot day, an air conditioner can use $7 of electricity. On a cold day, a heater can burn up $3 of fuel. The annual cost of operating this system is a measure of performance that should be tracked.

2.3. The Analysis Model

The analysis model serves as a point of inflection in our modeling effort. While the requirements activity focuses on the "voice of the customer," the analysis activity shifts that focus to the "voice of the designer." In analysis, we begin to reason about architectural components, or classes, and the relationships among them. These decisions are diagrammed with additional UML models.

Analysis starts with the skeleton use cases derived in the requirements model. It adds muscle to those skeletons and then creates more skeleton use cases. In our HVAC case study, our customer has just given us a new nonfunctional performance requirement. If the air conditioner were to turn on and off every minute, it would be very annoying. Therefore, on–off cycles should last at least 15 minutes. Creating a temperature dead zone is one obvious solution for this problem.

2.3.1. Analysis-model Use Cases

In moving from the requirements model to the analysis model, use cases should be added and details should be added to the existing use cases as decisions are made during the analysis-modeling activity. Here we have decomposed the Regulate Temperature use case into Cool House and Heat House use cases, and we have added detail to all use cases.

Name: Cool House.
Brief description: When it is hot outside, cool the house with an Air Conditioner (AC) and maintain the room temperature (roomTemp) between ACLowerLimit (see Rule2) and ACUpperLimit (see Rule3) degrees Fahrenheit.
Scope: An HVAC controller for a typical Tucson family house.
Primary actor: Home Owner.
Supporting actor: Air Conditioner.
Frequency: The system operates continuously.
Precondition: The Home Owner has turned on the cooling system and the roomTemp is between ACLowerLimit and ACUpperLimit.
Main Success Scenario:
1. The roomTemp exceeds ACUpperLimit.
2. The system turns on the Air Conditioner [in accordance with Rule1].
3. The roomTemp drops to the ACLowerLimit.
4. The system turns off the Air Conditioner [Rule1] [exit use case].

Rules:
Rule1: When the Air Conditioner is on, turn the Fan on.
When the Air Conditioner is off, turn the Fan off.
Rule2: ACLowerLimit default value is 72ºF.

Rule3: ACUpperLimit default value is 73ºF.
Nonfunctional performance requirement: On-off cycles should last at least 15 minutes.
Author/owner: Terry Bahill
Date: January 8, 2002

Name: Heat House.
Brief description: When it is cold outside, heat the house with a Heater and maintain the roomTemp between heaterLowerLimit (see Rule2) and heaterUpperLimit (see Rule3) degrees Fahrenheit.
Scope: An HVAC controller for a typical Tucson family house.
Primary actor: Home Owner.
Supporting actor: Heater.
Frequency: The system operates continuously.
Precondition: The Home Owner has turned on the heating system and the roomTemp is between heaterLowerLimit and heaterUpperLimit.
Main Success Scenario:
1. The roomTemp falls below heaterLowerLimit.
2. The system turns on the Heater [in accordance with Rule1].

3a. The roomTemp rises to the heaterUpperLimit.

4. The system turns off the Heater [Rule1] [exit use case].

Alternate Flows:

3b. Home Owner smells "gas."

3b1. Home Owner turns off the Heater [exit use case].

Rules:

Rule1: When the Heater is on, turn the Fan on.
When the Heater is off, turn the Fan off.
Rule2: heaterLowerLimit default value is 70ºF.
Rule3: heaterUpperLimit default value is 71ºF.
Nonfunctional performance requirement: On–off cycles should last at least 15 minutes.
Author/owner: Terry Bahill
Date: January 8, 2002

Name: Display System Status.
Brief description: The system shall monitor the health of each object in the system and display the status of the complete system. The display could be a simple go/no-go or it might be more sophisticated.
Scope: An HVAC controller for a typical Tucson family house.
Primary actor: Home Owner.
Frequency: The system shall display System Status continuously.
Main Success Scenario:
1. The Fan reports status to the Controller.
2. The Air Conditioner reports status to the Controller.
3. The Heater reports status to the Controller.
4. The Thermostat reports status to the Controller.
5. The Controller computes the System Status and sends results to the Thermostat.
6. The Thermostat displays the System Status.
7. The Home Owner observes the System Status periodically [repeat at step 1].

Author/owner: Terry Bahill
Date: January 10, 2002

Figure 4 shows the use-case diagram for the analysis-model use cases.

2.3.2. Potential Classes

A *class* is an abstraction of common properties from a set of similar objects. For example, the class mammal would contain dogs, tigers, and platypuses. Their common properties are that they have fur, are homoeothermic and feed their offspring milk.

Now, we need to identify the classes that may be needed in our HVAC system. Thinking about key abstractions, we get Heater and Air Conditioner (AC). Thinking about interfaces, gives us the Thermostat, the Heater Interface, and the Air Conditioner Interface. Most systems have a controller, such as our Regulate Temperature Controller. Underlining nouns in the problem statement and use cases gives us the Fan and Room Temperature. Douglas [2000] gives many more strategies for identifying classes.

Next, we identify particular instances of these classes, called objects. For this case study we used the objects :Thermostat, :Controller, :AC Interface and :Air Conditioner.

2.3.3. Collaboration Diagram

The analysis model often introduces a new view, the collaboration diagram, which shows sequential messages being passed between these objects [Gomaa, 2000]. Figure 5 shows such a collaboration diagram.

2.3.4. Class Diagram

The collaboration diagram helps us understand the relationships between the classes. These relationships are shown in the class diagram, as shown in Figure 6. Each class box nominally has three compartments: the top contains the name, the middle lists the attributes, and the bottom lists the operations, which are the functions [Oliver, Kelliher, and Keegan, 1997: 44].

2.3.5. Other Important Parts of the Analysis Model

Candidate Architecture. To ameliorate the risk of being too expensive, we have decided to switch to a piggyback cooling system. In the winter, the house will

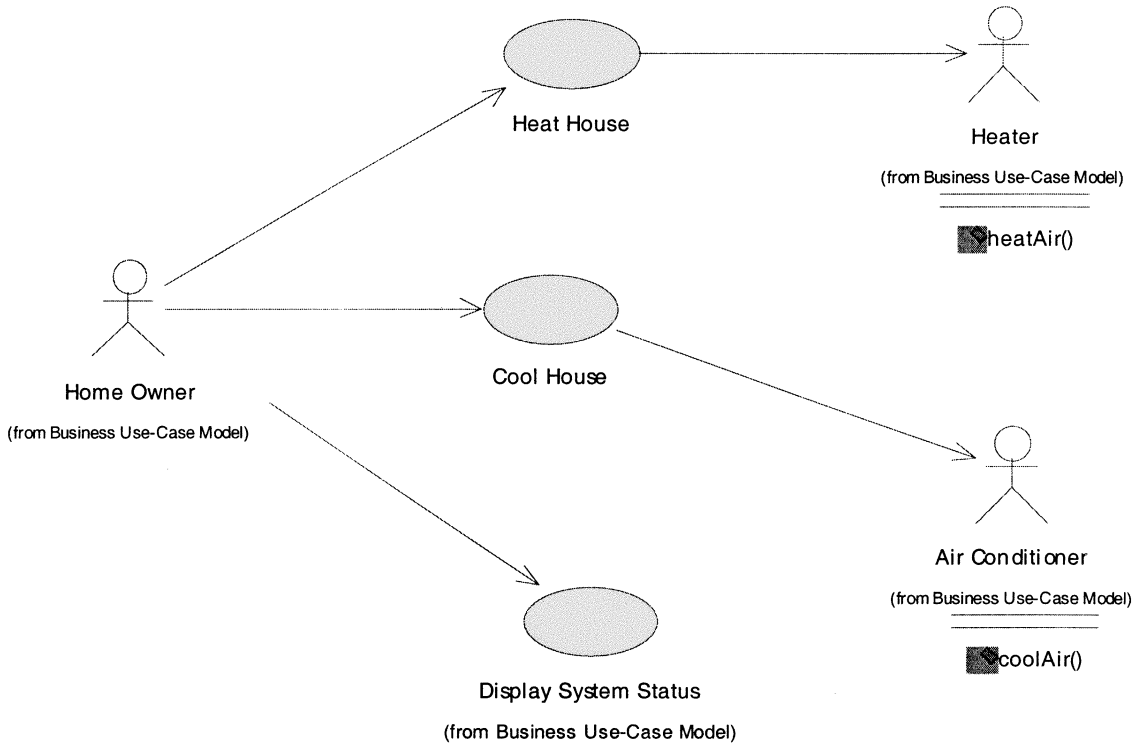


Figure 4. Analysis-model use-case diagram.

use a natural-gas heater. In the spring and fall, it will use an evaporative cooler. And July to September, it will use a 220-V electric air conditioner. A Cooling System is a generalization of the classes Evaporative Cooler and Air Conditioner. There will be one central thermostat for all three systems. Because of our new performance requirement that the on–off cycles should last at least 15 minutes, we have a new measure of effectiveness: average duration of on–off cycles.

Interface Definition. During the discovery of potential classes in the analysis model, thought should be given to the interfaces between these classes. The *port*, *connector*, and *protocol* concepts of the future UML 2.0 release can be used to define interfaces in a UML model [Hofmeister, Nord, and Soni, 2000; U2 Partners, 2001].

A port represents an interaction point between conceptual classes or components in a UML model. A port may be used to model both inputs and outputs of a class. A port obeys a well-defined protocol. The protocol should be described in terms of incoming and outgoing messages that flow into and out of the ports. Sequence diagrams can be used to show the time-sequenced receipt and dispatching of messages that define the protocol. A connector joins two ports and serves as a conceptual communications path between the classes. A connector may be physical or conceptual.

Interfaces between classes can be described by defining and describing the ports associated with each class (both input and output ports should be considered), the protocol expected when communicating with the class through each port, and the connectors between the classes. See Hofmeister, Nord, and Soni [2000] and

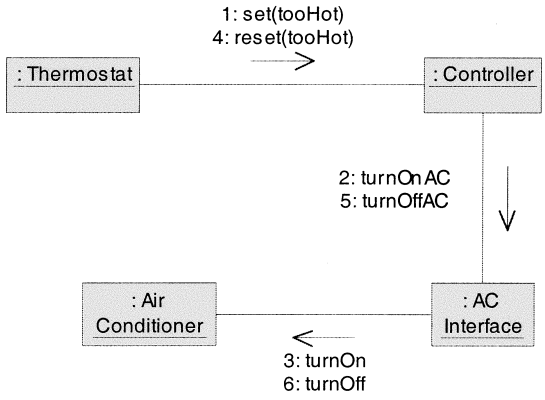


Figure 5. Analysis-model collaboration diagram.

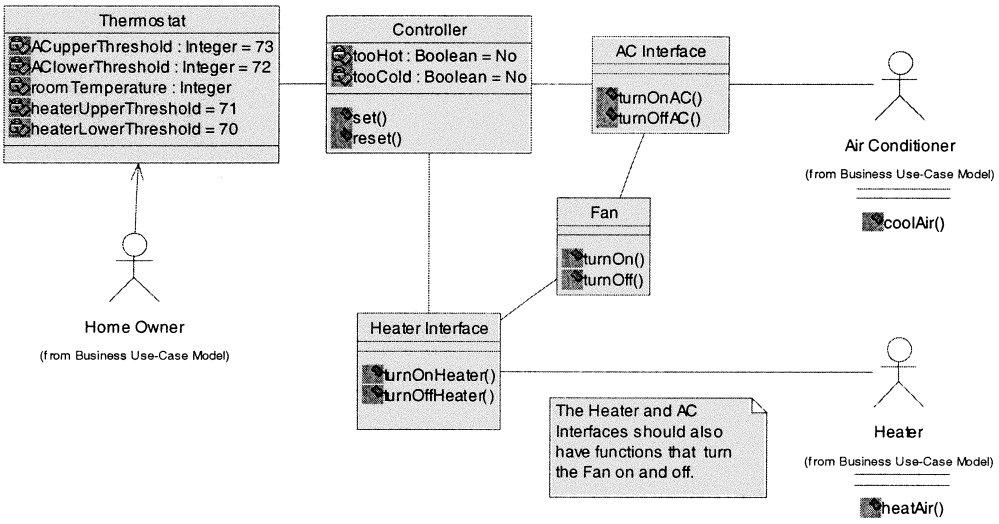


Figure 6. Analysis-model class diagram.

U2 Partners (http://cgi.omg.org/docs/ad/01-11-01.pdf) for a more detailed treatment.

Risk Analysis. At this point, we have identified two major risks: (1) The capacity might be inadequate to heat or cool the house. So, in selecting equipment, we will consult HVAC specialists as well as electrical and mechanical engineers. (2) The air conditioner could be too expensive for the homeowner. We have mitigated this risk by using a piggyback cooling system.

The Business Case. Assume the system is for an existing 2000 square foot house. On the hottest of days, without adequate insulation and careful caulking an air conditioner can cost as much as $7 a day. On a very cold day, heating could cost as much as $3 a day. Several aspects of the business case are given in Table I. The business case should also investigate schedule, cost, and performance.

2.4. The Design Model

The design model takes the skeletons and the muscularized skeletons developed in the analysis model and then adds muscle to the skeletons, develops the muscularized skeletons, and adds new skeleton use cases. It also further develops the system's classes. The Unified Systems Engineering Process uses iterative, incremental elaborations. Our design model must accommodate the new piggyback cooling system.

2.4.1. Design-model Use Cases

As with analysis modeling, design modeling is likely to produce more inputs into the use case model as specific design areas are finalized. The elaborated use cases should contain more concise business rules that exercise many alternative paths through the use case.

Name: Cool House.

Brief description: When it is hot outside maintain the room temperature (roomTemp) between coolerLowerLimit (see Rule2) and coolerUpperLimit (see Rule3) degrees Fahrenheit using a Cooler (either an Evaporative Cooler or an Air Conditioner).

Scope: An HVAC controller for a typical Tucson family house.

Level: Low.

Primary actor: Home Owner.

Supporting actor: Cooler.

Frequency: The system operates continuously.

Precondition: The Home Owner has turned on the system, the system mode is Cooler, system state is Cooler Off, and the roomTemp is between coolerLowerLimit and coolerUpperLimit.

Trigger: The roomTemp exceeds coolerUpperLimit.

TABLE I. Part of the Business Case

System	Capacity	Purchase Price	Annual Maintenance	Maximum Daily Cost
Heater	100,000 BTU	$ 500	$ 40	$3
Evaporative Cooler	7000 cfpm	$ 500	$ 50	$1
Air Conditioner	4 tons	$3000	$250	$7

Main Success Scenario:
1. The system turns on the Cooler [in accordance with Rule1].
2. The roomTemp drops to the coolerLowerLimit.
3. The system turns off the Cooler [Rule1] [exit use case].

Rules:
Rule1: When the Air Conditioner is on, turn the Fan on.
When the Air Conditioner is off, turn the Fan off.
Rule2: coolerLowerLimit default value is 72ºF.
Rule3: coolerUpperLimit default value is 73ºF.
Nonfunctional performance requirement: On–off cycles should last at least 15 minutes.
Author/owner: Terry Bahill
Date: January 31, 2002

Name: Heat House.
Brief description: When it is cold outside, maintain the roomTemp between heaterLowerLimit (see Rule2) and heaterUpperLimit (see Rule3) degrees Fahrenheit using a Heater.
Scope: An HVAC controller for a typical Tucson family house.
Level: Low.
Primary actor: Home Owner.
Supporting actor: Heater.
Frequency: The system operates continuously.
Precondition: The Home Owner has turned on the system, system mode is Heater, system state is Heater Off, and roomTemp is between heaterLowerLimit and heaterUpperLimit.
Trigger: The roomTemp falls below heaterLowerLimit.
Main Success Scenario:
1. The system turns on the Heater [in accordance with Rule1].

2a. The roomTemp rises to the heaterUpperLimit.
3. The system turns off the Heater [Rule1] [exit use case].

Alternate Flows:
2b. Home Owner smells gas.
2b1. Home Owner turns off Heater [exit use case].
Rules:
Rule1: When the Heater is on, turn the Fan on.
When the Heater is off, turn the Fan off.
Rule2: heaterLowerLimit default value is 70ºF.
Rule3: heaterUpperLimit default value is 71ºF.
Nonfunctional performance requirement: On–off cycles should last at least 15 minutes.
Author/owner: Terry Bahill
Date: January 31, 2002

Name: Display System Status.
Brief description: The system shall monitor the health of each object in the system and display the status of the complete system. This display should simply indicate *ready* or *fault.*
Scope: An HVAC controller for a typical Tucson family house.
Level: Medium
Primary actor: Home Owner.
Frequency: The systemStatus shall be computed periodically (e.g. every minute) or it may be event driven (e.g., on each state transition). The system shall display the systemStatus continuously.
Precondition: None.
Main Success Scenario:
1. The Fan Interface executes its Built in Test (BiT) for the Fan and the Fan Interface and reports the results to the Controller.
2. The Air Conditioner Interface executes its BiT and reports the results to the Controller.
3. The Evaporative Cooler Interface executes its BiT and reports the results to the Controller.
4. The Heater Interface executes its BiT and reports the results to the Controller.
5. The Thermostat executes its BiT and reports the results to the Controller.
6. The Controller executes its BiT and computes the systemStatus.
7. The Controller sends the systemStatus to the Thermostat.
8. The Thermostat displays the systemStatus.
9. The Home Owner observes the systemStatus periodically [repeat at step 1].

Author: Terry Bahill
Date: January 31, 2002
Note: Other system modes that could be displayed by the same display unit include on/off, heat/cool, fan/auto.

Name: Set Temperature Limits.
Brief description: Home Owner changes the HVAC limits.
Scope: An HVAC controller for a typical Tucson family house.
Level: Medium.
Primary actor: Home Owner.
Frequency: Daily.
Precondition: systemStatus must be *ready*.
Main Success Scenario:
1. Home Owner sets coolerUpperLimit.
2. Home Owner sets coolerLowerLimit.
3. Home Owner sets heaterUpperLimit.
4. Home Owner sets heaterLowerLimit.
5. Exit Set Temperature Limits use case.

Postcondition: All four limits are set.
Author/owner: Terry Bahill
Date: January 31, 2002
Alternative designs:
1. The Home Owner might set the upperLimits and a deadZone.
2. The deadZone might be fixed (at say 2°F) and the Home Owner only sets the coolerUpperLimit and the heaterLowerLimit.
3. Use a timer instead of a dead zone.

Energy saving hints: Raise the Air Conditioner limits 1ºF for each hour that the house will be unoccupied. Do not switch back and forth between the evaporative cooler and the air conditioner more frequently than weekly.

Name: Choose Equipment.
Brief description: Home Owner selects the equipment to turn on: heater, air conditioner, or evaporative cooler.
Scope: An HVAC controller for a typical Tucson family house.
Level: High.
Assume that if the Home Owner wants heating and cooling on the same day, then he or she will switch back and forth manually.
Primary actor: Home Owner.
Frequency: There is an annual cycle.
Precondition: systemStatus must be *ready*.
Main Success Scenario:
1. March 21 Home Owner turns Heater off and Evaporative Cooler on.
2. June 21 Home Owner turns Evaporative Cooler off and Air Conditioner on.
3. September 21 Home Owner turns Air Conditioner off and Evaporative Cooler on.
4. November 21 Home Owner turns Evaporative Cooler off and Heater on [cycle back to step 1].

Postcondition: One of the three systems is turned on.
Author/owner: Terry Bahill
Date: January 31, 2002
Alternative designs: Instead of switching on fixed dates, we could use rules to determine when to switch from heater, to cooler, to AC. Rule inputs would be average humidity, dew point temperature, lowest daily temperature, cost of AC, cost of evaporative cooling, number of days since the last switch and customer preferences.

Other use cases that might be considered for this HVAC system include "Switch Fan between *Automatic* and *On*" (this would change Rule1), "Control Humidity," and "Ventilate with Fresh Air."

Figure 7 shows the design-model use-case diagram. This diagram serves as a table of contents for the design-model use cases. That is, all of the use cases that have been developed for the model are shown on this diagram, and only those use cases are shown.

2.4.2. Sequence Diagrams

Sequence diagrams show the messages (or commands) being passed between the objects for one particular instance of one flow of one use case. A sequence diagram contains the same information as a collaboration diagram: it is just displayed differently. In fact, the computer usually creates one from the other. Figure 8 shows a sequence diagram for the normal operation of the Cool House use case. Figure 9 shows a sequence diagram for the normal operation of the Heat House use case. Figure 10 shows a sequence diagram for the alternate flow of the Heat House use case when the Home Owner smells "gas."

2.4.3. Design-model Class Diagram

In the design model, we will add detail to the classes identified in the analysis model and we will add more classes. For each class we might add attributes, responsibilities, stereotypes, roles, dependencies, associations or multiciplicies.

Attributes: For each attribute we show

name: Type = default value

Thermostat

coolerUpperLimit: Integer = 73

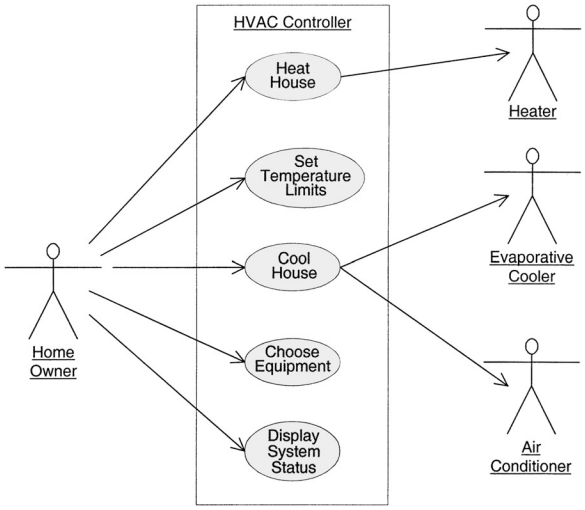


Figure 7. Design-model use-case diagram.

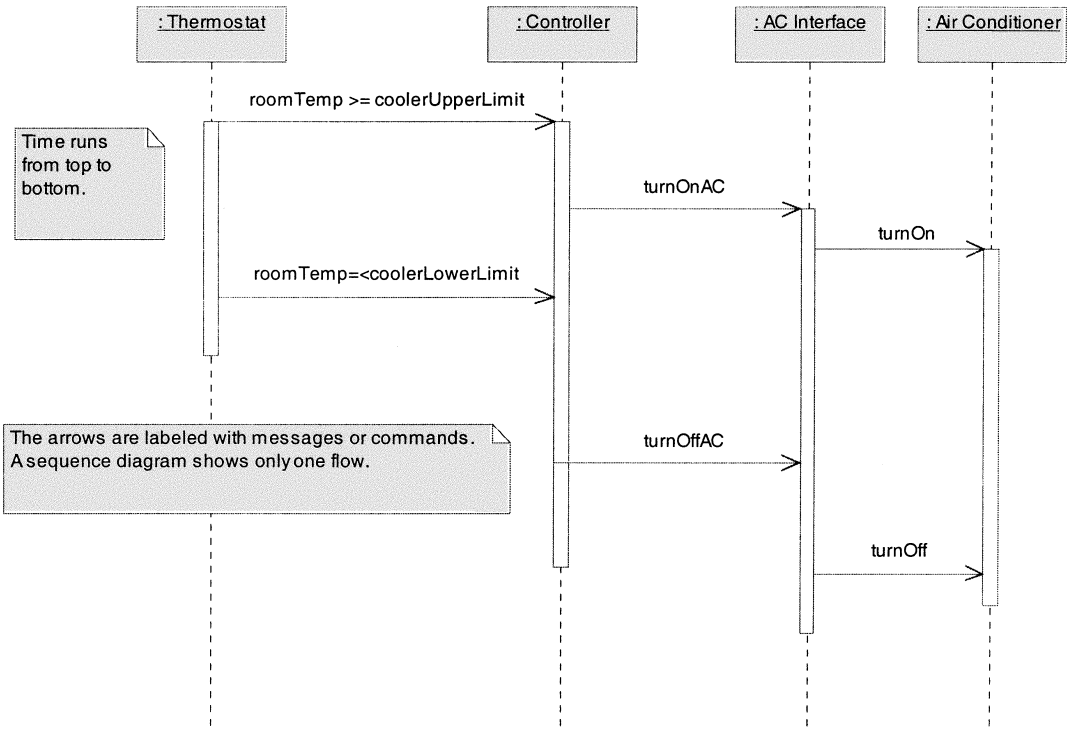


Figure 8. Design-model sequence diagram for Air Conditioner normal operation.

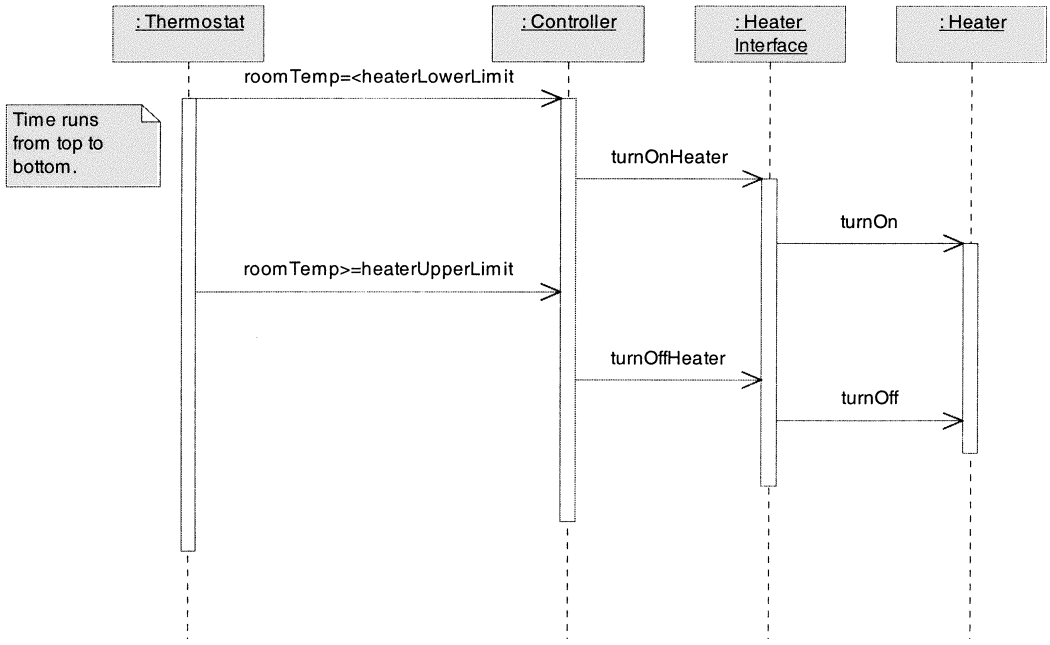


Figure 9. Design-model sequence diagram for Heater normal operation.

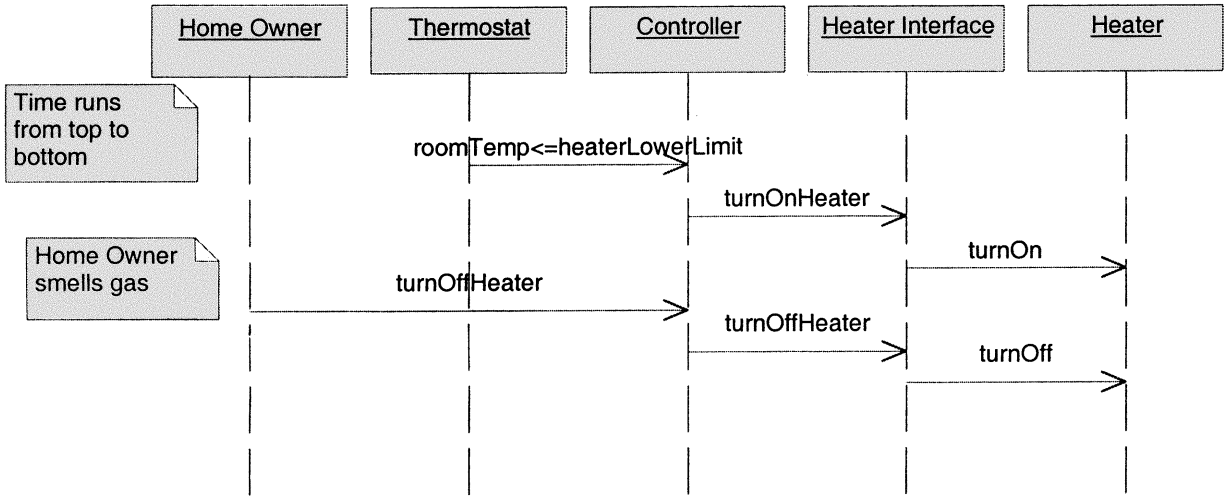


Figure 10. Design-model sequence diagram when the Home Owner smells gas.

coolerLowerLimit: Integer = 72

heaterUpperLimit: Integer = 71

heaterLowerLimit: Integer = 70

roomTemp: Integer

Controller

systemStatus: Boolean = fault

Thermostat operations

roomTemp >= coolerUpperLimit could be renamed contactsClosed.

roomTemp <= coolerLowerLimit could be renamed contactsOpened, etc.

This level of detail would be appropriate for an implementation model, but a design model should stay closer to the problem domain and farther from the solution domain.

Other classes that may be necessary: Calendar, Timer, and Algorithm (for selecting equipment).

Figure 11 shows the class diagram for this model. However, this is an impoverished example of a class diagram. It lacks associations, roles, multiplicities, visibility, etc. This often happens with extremely simple hardware examples. We could make the diagram richer by positing a class called Cooler, of which Air Conditioner and Evaporative Cooler are subtypes; this would show the inheritance relationship. We could add multiplicities, but most are one, except the Home Owner and the evaporative cooler could be one or two. However, too much detail may confuse rather than clarify. Because actors are not a part of the system being designed, they are often excluded from class diagrams.

2.4.5. Statechart

A statechart shows the dynamic behavior of an object [Douglas, 2000; Gomaa, 2000]. Statecharts are typically constructed to understand the behavior of *Active Classes*. An active class represents a thread of control in the system. Objects that are instances of active classes can initiate communication with other classes and serve to control the overall flow for a particular behavioral thread—usually to help satisfy a given use case. Figure 12 shows the statechart for the controller class in our system.

2.4.6. Other Stuff

Because we switched to a piggyback cooling system, we now have an evaporative cooler, which is inexpensive, but it does not have a small dead zone. This therefore necessitates a new measure of effectiveness: excursions out of the dead-zone; number, size, and duration.

Infrastructure. The house will need to have a gas line, a water line, a drainpipe, 110- and 220-V electricity and air ducts.

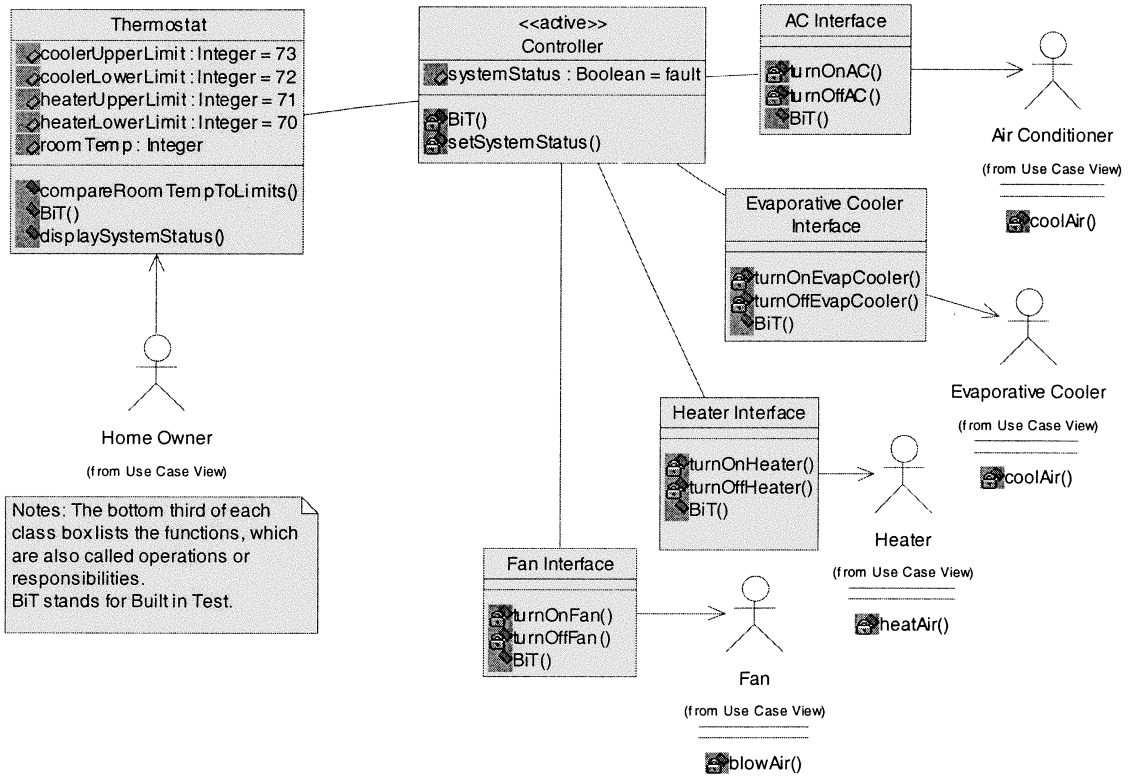


Figure 11. Design-model class diagram.

2.5. Implementation Model

In this case study, we did not do anything for the implementation model because we are designers, not builders. We would hire specialty engineers to help us construct the implementation model. The class descriptions would contain or reference manufactures' names, models, and specifications. This model should also contain schematic diagrams, blueprints, deployment diagrams, etc. Most importantly, it would be based on our design model using the same UML tools. Of course, we already got advice from these specialty engineers in the first phase of the project where we defined the operational scenario.

2.6. Verification Model

System verification means building the system right: ensuring that the system complies with its requirements and conforms to its design. Requirements verification means proving that each requirement has been satisfied: this can be done by inspection, modeling, simulation, analysis, test or demonstration.

Plans for testing the system should start at the beginning of the project. Tests should be performed at the end of each iteration. The following shows some of the tests that could be performed at the end of the construction phase. Obviously, the verification model for a system should be much bigger than this example.

2.6.1. Cooling System Test Case

Tests can be designed using statecharts. Normal inputs should be used as well as those that stress the system. The following inputs would be suitable for testing the Cooler:

Normal:

coolerUpperLimit = 73°F and coolerLowerLimit = 72°F

Extremes:

coolerUpperLimit = 85°F and coolerLowerLimit = 83°F

coolerUpperLimit = 65°F and coolerLowerLimit = 63°F

Mistakes:

coolerUpperLimit = 71°F and coolerLowerLimit = 73°F

These inputs would be applied to the system and changes of state would be observed. Table II shows a matrix of test vectors for the Cooler. Notice that this test case reveals the need for another output: signalError.

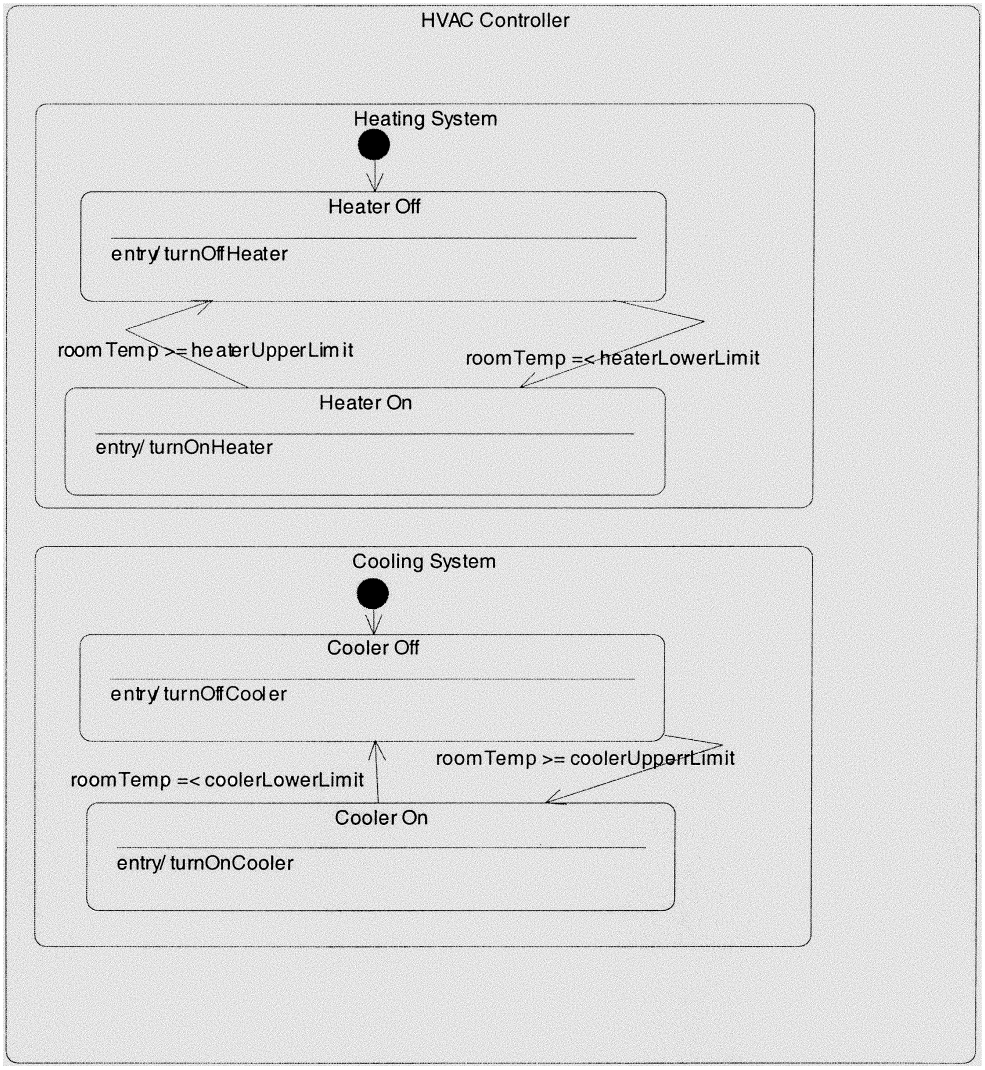


Figure 12. Statechart for the HVAC controller.

The following test procedure specifies how the test vectors will be applied.

1. Choose a hot day, e.g., outside temperature > 90°F.
2. Set Cooler On/Off switch to On.
3. Set coolerUpperLimit and coolerLowerLimit according to the Test Case [ ].
4. Observe room temperature for 1 hour.

TABLE II. Test Matrix for the Cooler

Initial State	Event (Inputs)	Next State	Resulting Output
Any	roomTemp > coolerUpperLimit	Cooler on	Cooler on
Cooler off	roomTemp < coolerUpperLimit	Cooler off	Cooler off
Cooler off	roomTemp = coolerUpperLimit	Cooler on	Cooler on
Cooler on	roomTemp > coolerLowerLimit	Cooler on	Cooler on
Cooler on	roomTemp = coolerLowerLimit	Cooler off	Cooler off
Any	roomTemp < coolerLowerLimit	Cooler off	Cooler off
Any	coolerUpperLimit ≤ coolerLowerLimit	Cooler off	SignalError

TABLE III. Test Matrix for the Heater

Initial state	Event (Inputs)	Next State	Resulting Output
Any	roomTemp > heaterUpperLimit	Heater off	Heater off
Heater off	roomTemp > heaterLowerLimit	Heater off	Heater off
Heater off	roomTemp = heaterLowerLimit	Heater on	Heater On
Heater on	roomTemp < heaterUpperLimit	Heater on	Heater on
Heater on	roomTemp = heaterUpperLimit	Heater off	Heater off
Any	roomTemp < heaterLowerLimit	Heater on	Heater on
Any	heaterUpperLimit ≤ heaterLowerLimit	Heater off	SignalError

5. If it goes outside the coolerLimits, record a defect.

2.6.2. Heating System Test Case

The following inputs would be suitable for testing the Heater.

Normal:

heaterUpperLimit = 71°F and heaterLowerLimit = 70°F

Extremes:

heaterUpperLimit = 80°F and heaterLowerLimit = 78°F

heaterUpperLimit = 36°F and heaterLowerLimit = 34°F

Mistakes:

heaterUpperLimit = 70°F and heaterLowerLimit = 72°F

These inputs would be applied to the system and changes of state would be observed. Table III shows a matrix of test vectors for the Heater.

The following test procedure specifies how the test vectors will be applied.

1. Choose a cold day, e.g., outside temperature < 60°F.
2. Set Heater On/Off switch to On.
3. Set heaterUpperLimit and heaterLowerLimit according to the Test Case [ ].
4. Observe room temperature for 1 hour.
5. If it goes outside the heaterLimits, record a defect.

2.6.3. Testing Using Use Case Scenarios

The system can also be tested using the use cases. Particular instances of use cases (as in collaboration and sequence diagrams) can be used to test the system. For example,

Use Case Name: Heat House.

Scenario:

1. Pat Harris starts this scenario on January 8 with the outside temperature = 40°F, the heater off, the roomTemp = 70.5°F and the heaterLowerLimit and heaterUpperLimit at their default values.
2. The roomTemp falls below 70°F.
3. The system turns on the Heater and the Fan.
4. The roomTemp rises to 71°F.
5. The system turns off the Heater and the Fan.
6. For the next 15 minutes, the roomTemp must remain greater than the heaterLowerLimit [end of test].

This is called an instance or an instantiation of a use case, because particular names, dates, places, times and temperatures are substituted for the parameters in the use case.

2.7. Operations Model

The operations model (usually a computer simulation) should be built from the implementation model. It should reflect the structure of the system as it was actually built [Wymore, 1993]. It will be used to manage and improve the operational system. It will be updated anytime the operational system is changed. Most importantly, it will used to help with retirement of the system. Another UML tool called an activity diagram will be used to show the workflows, that is who does which tasks.

3. GLOSSARY FOR THE HVAC SYSTEM

A glossary is a mandatory part of a Unified Systems Engineering Process design.

Built in Test (BiT)—systems should be designed so that they test themselves and make the results of the test known.

coolerLowerLimit—when the room temperature gets below this, temperature the cooler should be turned off.

coolerUpperLimit—when the room temperature gets above this temperature, the cooler should be turned on.

Dead zone—When the room temperature is in this zone between the upper and lower limits, nothing should be turned on.

Evaporative cooler—an effective but inexpensive cooling system for dry climates. A unit has a box with water absorptive pads. When dry air is blown through the pads, water evaporates and the air is cooled.

heaterLowerLimit—when the room temperature gets below this temperature, the heater should be turned on.

heaterUpperLimit—when the room temperature gets above this temperature, the heater should be turned off.

HVAC—heating, ventilation, and air conditioning system

Piggyback system—a cooling system with both an air conditioner and an evaporative cooler. The evaporative cooler is used when the relative humidity is low, and the air conditioner is used when the humidity is relatively high.

systemStatus—a Boolean flag that indicates whether or not all systems are functional.

4. LEVELS AND ASPECTS

Most systems are impossible to study in their entirety, but they are made up of hierarchies of smaller subsystem that can be studied. Simon [1962] discusses the necessity for such hierarchies in complex systems. He shows that most complex systems are decomposable, enabling subsystems to be studied outside the entire hierarchy. For example, in studying the motion of a baseball, it is sufficient to apply Newtonian mechanics considering only gravity, air, the ball, and the bat. One need not worry about electron orbits or the motions of the sun and the moon. Forces that are important when studying objects of one order of magnitude seldom have an effect on objects of another order of magnitude. Engineers, therefore, study subsystems of large systems. The order of magnitude could be for forces, physical size, time intervals, or complexity. Often this dimension is called the *level*. It is important that interacting elements of a model be at the same level. If the use case is at one level and the classes are at a different level, the model will be hard to understand.

In our HVAC system, we modeled objects that the user is likely to interact with, such as the thermostat, the controller, the heater, the air conditioner, and the evaporative cooler. We did not worry about low-level details such as the colors of the wires interconnecting the equipment and the electrical voltages. Likewise, we did not worry about high-level issues, such as possible brown outs or heating of the city caused by operation of a million air conditioners. Our use cases were all at the user level, although we did find it useful break these up in to sublevels, as indicated in the use case descriptions. When dealing with use cases at one level, you might be interested in use cases one level above or below, but you would seldom be interested in use cases two levels above or below.

TABLE IV. Two Aspects of the HVAC System

Operations Aspect	
Level	Use cases
City	Heat the city, cause brown outs
Architect	Choose and install equipment and insulation, specify architecture of the house
User	Heat house, cool house, display system status, set temperature limits, choose equipment
Equipment	Turn equipment on and off, open and close windows and vents
Electricity	Specify colors of wires interconnecting equipment, choose electrical voltages
Element	Freon changes phases, helical spring expands, mercury capsule tilts
Maintenance Aspect	
Level	Use cases
Generational	Freon depletes ozone layer
Multi year	Vacuum ducts, caulk small openings, sweep chimney, bleed radiators
Seasonal	Light pilot, dust fans, add Freon, oil motors, change cooler pads, adjust storm windows
Periodic	Clean or replace filters, replace fuses
Continuous	Circulate oil, pump water

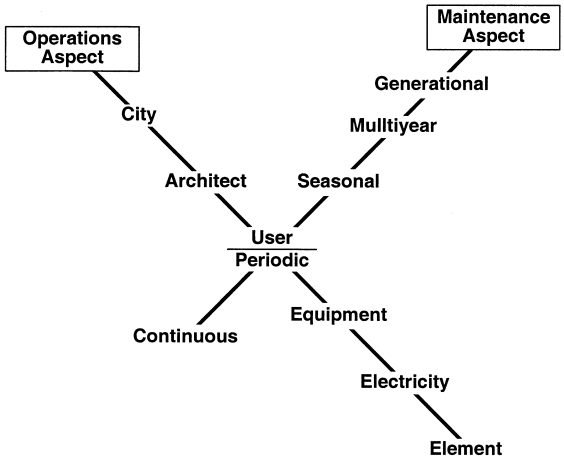


Figure 13. Most of the use cases in this paper exist at the intersection of the User Operations level and the Periodic Maintenance level.

It is important to select the correct level for each model. But is it is also important to identify alternative aspects of the system under study. For the HVAC system, we will first look at the operations aspect and the maintenance aspect as defined in Table IV.

The operations aspect is a physical decomposition, and the maintenance aspect is a temporal decomposition. Many other aspects of an HVAC system are possible, such as functions, forces, economics, recycling, and airflow. Often the aspects of the system will intersect at an operating point, as is shown in Figure 13. The use cases in this paper exist at the intersection of the user operations level and the periodic maintenance level.

In Zachman Frameworks, the levels are Scope, Business Model, System Model, Technology Model, and the Detail Model. The aspects (called perspectives) are who, what, when, where, why, and how. They emphasize that in order to understand an enterprise, each cell in the framework must be modeled [Zachman, 1987].

5. CHALLENGES

There are challenges in teaching hardware and algorithm designers to use these new software tools.

5.1. Changing from Finite State Machines to Statecharts

In the old days, we designed computer systems so that they sampled all of their inputs on every clock cycle. These systems looked at the present state and the present inputs and then determined their next state. That was fine when the clock was running at a megahertz. But now that the clocks are running at over a gigahertz, we do not want to load down the controllers like that. Instead, we want our state transitions to be event driven or at least sampled with different clocks. Often this means putting intelligence in the sensors. We no longer want the sensors to be simple switches that are polled by the controller each nanosecond. Instead, we want the sensors to interrupt when significant events have occurred. (However, the UML does have the ability to check guard conditions on every clock cycle, so designers could still use the UML in the old-fashioned polling fashion, if they wanted to.) Our old finite state machines were either Mealy (pulse outputs labeled on the transitions between states) or Moore (level outputs labeled with the states) machines. UML statecharts encourage both types of outputs in the same diagram. Finally, statecharts encourage concurrent state machines and hierarchies of state machines. Switching from finite state machines to statecharts requires a change in thinking for hardware designers.

5.2. Changing from Functional Thinking to Object Orientation

In the old days, we focused on the functions that the system had to perform. Functional flow block diagrams showed the functions the system had to perform and the inputs that were transformed into outputs by those functions. Some designers also listed the physical element that the function was assigned to, but, in general, there was no formality to the mapping between functions and the components that would implement them, Wymorian notation being the exception [Wymore, 1993]. Systems engineers seeped in the functional mentality might think that the UML ignores functions. It does not. The functions are listed in the bottom part of each class box. In fact, the purpose of analysis and design is assigning responsibilities or functions to candidate classes. This is the art form of system design. The way in which you assign responsibilities defines the characteristics of your system (flexibility, reliability, scalability, etc.) The result of responsibility assignment is the system architecture, or the essence of your system. So traditional hardware design tools focus on the functions and only mention the components in passing, and the UML techniques focus on the objects and merely list the functions in passing. This shift in emphasis from functions to objects requires a paradigm shift for hardware designers.

5.3. Progressing from Use Cases to Classes

UML designs are use-case based. Use cases capture the required functionality of the system. Designers use the use cases to discover the fundamental classes that the

system will require. But, because the use cases capture the required functionality of the system, it may be hard for a hardware designer to give up on the functions and concentrate on the classes. After all, the functions are what he or she wants. This also requires a different way of thinking.

Most of the classes created by hardware designers are abstractions of physical objects. Sometimes it is hard to discover classes that are not based on physical objects, an example of such a nonphysical class is the algorithm for selecting equipment in our HVAC system.

5.4. Changing from Waterfall Processes to Incremental Iterations

Our last paradigm shift is changing from waterfall processes to incremental iterations. In the old days, requirements engineers dug out the requirements, wrote them up, and threw them over the wall to the designers. They created designs and threw them over the wall to manufacturing, etc. The Unified Systems Engineering Process encourages designers to start with the high-risk and high-priority portions. Produce a skeleton design and try it out. Then in the next iteration add muscle to that skeleton and add skeletal designs for additional functions, etc.

5.5. Using UML for Real-Time Systems

The UML was not designed for real-time systems. However, many authors have discussed ways of dealing with timing issues. Douglas [2000] has several examples including a cardiac pacemaker. Axelsson [2002] used UML tools to model an automobile engine throttle system, and Neill and Holt [2002] created UML extensions to handle timing issues. Also old-fashioned transfer functions should be used in conjunction with UML tools when they help communicate the designs; see, for example, Gomaa [2000: Figure 20.1].

6. CONCLUSIONS

The Unified Modeling Language and the new software development processes can be used for hardware and algorithm design. This paper primarily illustrated the Unified Systems Engineering Process that was derived from these new software development processes. In passing, it showed rudimentary use of some of the UML tools. Finally, it presented some ideas that might help hardware and algorithm designers to use the Unified Systems Engineering Process and some of the UML tools. The UML tools are excellent communications tools. They are improvements of ancient systems engineering tools but they were designed to function together.

APPENDIX: UML NAMES AND RELATIONSHIPS

The following comments are generalizations drawn from a large number of sources. No source stated things exactly this way. Someone somewhere would probably disagree with each statement. But if these gross generalizations help, then use them.

Actions and **activities** are the outputs in statecharts. Actions are associated with transitions between states, and activities are associated with the states. They should be named with verbs or verb phrases. Typically, the first letter of the first word is lowercase, the first letters of subsequent words are uppercase, and there are no spaces between the words.

Actors are named with nouns or noun phrases. Their names should reflect the roles that are played and not the names of actual people. Names of actors are usually written with the first letter of each word capitalized and spaces between the words. Supporting (or secondary) actors are not a part of the system being designed, and thus cannot be designed or easily altered. They often represent external systems or commercial off-the-shelf (CotS) components.

Attributes help specify the state of a class. They should be named with nouns or noun phrases. They are usually written with the first letter of the first word in lowercase, the first letters of subsequent words uppercase, and no spaces between the words.

An **event** is an occurrence that has a location in time and space. In statecharts, events are the inputs that cause transitions between states. Events should be named with verbs or verb phrases. Typically, the first letter of the first word is lowercase, the first letters of subsequent words are uppercase, and there are no spaces between the words.

A **Function** converts an input into an output at a specified boundary. The function signature identifies the inputs and outputs. Functions (also called responsibilities and operations) should be named with verbs, verb phrases, or clauses. Typically, the first letter of the first word is lowercase, the first letters of subsequent words are uppercase, and there are no spaces between the words.

Inputs and **outputs** can be identified using an object (or context) diagram [Gomaa, 2000]. Arrows pointing into an active object represent inputs (events) that trigger state transitions. Arrows going out of an object show the functions (activities and actions) that are the outputs on the statechart.

Messages (also called commands) tell an object to do something that it knows how to do. The system inputs and outputs are often specified here. Messages should be named with verbs or verb phrases. Typically, the first letter of the first word is lowercase, the first letters of subsequent words are uppercase, and there are no spaces between the words.

Objects and **classes** are named with nouns or noun phrases. They are usually written with the first letter of each word capitalized and no spaces between the words. Object names are usually underlined.

The **precondition,** the **trigger**, and the first step of the main success scenario are interrelated and are often confused and interchanged. If you already have a statechart, then use it to help rewrite the use case. The precondition (and the postcondition) should contain, among other things, the state of the system and values for pertinent attributes. The trigger should contain the event that causes a transition from the precondition state. The first step of the main success scenario may contain actions and activities.

States should be named with historical statements or temporal phrases describing past or present activities. A state name should reflect an interval of time when something is happening. A state name can be a gerund, a noun phrase, a clause, or even a sentence. State names should not sound like combinations of input conditions. State names are usually written in sentence case with the first letter of each major word capitalized and spaces between the words.

The **type** of an attribute is initial letter uppercase, such as Boolean or Integer. Types are adjectives.

Use cases should be named with verb–noun phrases. They should relate to the problem domain and not to any particular solution. The verb should be in the imperative mood. Use case names are usually written with the first letter of each word capitalized and spaces between the words. Bahill's use cases template is available at http://www.sie.arizona.edu/sysengr/slides/templates.doc.

Some UML things on one diagram seem to be the same or similar to other UML things on other diagrams. Table V shows what things are allowed on what diagrams. Inputs, outputs, and functions are not UML things, but we have included them because systems engineers want to know about them. Once again, we warn that Table V presents gross generalizations. We doubt that anyone would agree with every detail. Also, we have not included all of the UML views; for example, we have omitted activity diagrams and packages.

TABLE V. Things That Can Be in Different UML Diagrams

	Use Case	Collaboration	Sequence	Class	Statechart	Deployment	Object
Actions					X		
Activities					X		
Actors	X	X	X	X		X	
Associations	X			X			
Attributes				X			
Classes				X		X	
Events					X		
Focus of control			X				
Functions	X				X		
Guard conditions							X
Inputs							
Messages		X	X				X
Multiplicities				X			
Nodes							X
Objects		X	X	X			
Operations				X			X
Outputs							
Responsibilities				X			
Roles	X			X			
States					X		
Visibility				X			
Number of use cases represented	Many	A fraction of 1	A fraction of 1	All	Many	All	A fraction of 1

Actions and activities are different, but each is synonymous with functions, operations, responsibilities, and outputs. Oliver, Kelliher, and Keegan [1997: 44] also treat function, method, operation, and activity synonymously. Some messages on collaboration or sequence diagrams can become activities or actions on a statechart, while other messages can become events, which are synonymous with inputs. Actors and roles are related.

REFERENCES

J. Axelsson, Model based systems engineering using a continuous-time extension of the Unified Modeling Language (UML), Syst Eng 5(3) (2002), 165–179.

A.T. Bahill and B. Gissing, Re-evaluating systems engineering concepts using systems thinking, IEEE Trans Syst Man Cybernet Part C Appl Rev 28(4) (1998), 516–527.

A.T. Bahill, M. Alford, K. Bharathan, J. Clymer, S. Dahlberg, D.L. Dean, J. Duke, G. Hill, E. LaBudde, E. Taipale, and A. W. Wymore, The design-methods comparison project, IEEE Trans Syst Man Cybernet Part C Appl Rev 28(1) (1998), 80–103; available, with additional commentary and examples, at http://www.sie.arizona.edu/sysengr/methods2.

M.P. Bienvenu, I. Shin, and A.H. Levis, C4ISR architectures: III, An object-oriented approach fro architecture design, Syst Eng 3(4) (2000), 288–312.

A. Cockburn, Writing effective use cases, Addison-Wesley, Reading, MA, 2001.

B.P. Douglas, Real-time UML, Addison-Wesley, Reading, MA, 2000.

M. Fowler and K. Scott, UML distilled: A brief guide to the standard object modeling language, Addison-Wesley, MA, 2000.

H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, Addison-Wesley, Reading, MA, 2000.

C. Hofmeister, R. Nord, and D. Soni, Applied software architecture, Addison Wesley, Reading, MA, 2000.

I. Jacobson, G. Booch, and J. Rumbaugh, The unified software development process, Addison-Wesley, Reading, MA, 1999.

D. Kulak and E. Guiney, Use cases: Requirements in context, Addison-Wesley, Reading, MA, 2000.

J. Marasco, Explaining the UML, 2002, http://www.therationaledge.com/content/apr_01_uml_jm.html.

C.J. Neill and J.D. Holt, Adding temporal modeling to the UML support systems design, Syst Eng 5(3) (2002), 213–222.

I. Ogren, Possible tailoring of the UML for systems engineering purposes, Syst Eng 3(4) (2000), 212–224.

D.W. Oliver, T.P. Kelliher, and J.G. Keegan, Engineering complex systems with models and objects, McGraw-Hill, New York, 1997.

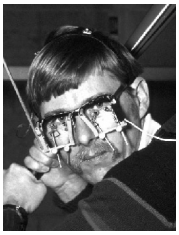
RUP, Rational Unified Process, 2002, http://www.rational.com/products/rup/index.jsp.

H.A. Simon, The architecture of complexity, Proc Am Phil Soc 106 (1962), 467–482.

U2 Partners, Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02: Unified Modeling Language 2.0 Proposal version 0.64 (draft), 2001, http://cgi.omg.org/docs/ad/01-11-01.pdf

A.W. Wymore, Model-based systems engineering, CRC Press, Boca Raton, FL, 1993.

J. A. Zachman, A framework for information systems architecture, IBM Syst J 26(3) (1987), 454–470.



Terry Bahill has been a Professor of Systems Engineering at the University of Arizona in Tucson since 1984. He received his Ph.D. in electrical engineering and computer science from the University of California, Berkeley, in 1975. He holds U.S. Patent Number 5,118,102 for the Bat Chooser, a system that computes the Ideal Bat Weight for individual baseball and softball batters. He is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE), of Raytheon and of INCOSE. He is chair of the INCOSE Fellows Selection Committee. This picture of him is in the Baseball Hall of Fame's exhibition *Baseball As America.*

Jesse Daniels is a systems engineer with BAE Systems in San Diego. He earned a B.S. and an M.S. in Systems Engineering in 1999 and 2000 from the Department of Systems and Industrial Engineering at the University of Arizona.