INFO TO GO

- Every modeling effort should begin with defined objectives.
- Turning diagrams into models is an effective way to define and understand requirements.
- Models help illustrate software's three dimensions: function, information, and behavior.

3 types of diagrams and how to turn them into powerful requirements tools by Becky Winant

VISUAL Requirements

"I prefer drawing to talking. Drawing is faster, and leaves less room for lies." **—Le Corbusier, architect**

I ONCE KNEW A SOFTWARE ENGINEER WHO NEEDED TO DEVELOP code to run a machine in a paper mill. The machine processed, pressed, and rolled pulp into large industrial rolls. The engineer was not an expert in the machinery; the people in the plant did not understand software or electronics. To bridge the gap, the engineer sketched a simple picture of the paper-rolling machine as she saw it. Then, as she talked with the people in the plant, she would point to a machine segment on her drawing and ask how it worked. She would add the segment parts and names to the drawing and record the information about them in her notebook. By making a picture, she developed a sense of the questions to ask. The conversations about the drawing helped the people in the plant feel included in the process of changing the equipment. That one sketch helped build rapport and understanding that lasted through to installation.

If interdependencies aren't obvious, requirements definitions won't be either. That's where models come in.

In the same way, simple models can promote our understanding of requirements. Software and system requirements are often vague and muddled. Software systems inhabit a multi-dimensional world with simultaneous activities, interactions, and rules governing when parts may or may not operate. If interdependencies aren't obvious, requirements definitions won't be either. Text's linear form is limiting—there isn't any clear view of the intersection of parts. That's where models come in. If people can *see* the system, they can better focus their questions and associate any answers with information they already have.

Software people have always built diagrams such as flowcharts and system block diagrams to visualize software abstractions. As software systems and their related risks have expanded and become more sophisticated, so have the tools to build these diagrams. Today, the Unified Modeling Language (UML) is intended to end the need for other tools; however, other notations may be equally effective and can serve purposes not addressed by UML. I won't be recommending one approach or notation. Process and notation expositions can be easily found elsewhere. My experience has been that *any* approach can work if it fits with the organizational culture and products, and if people find it useful. Whether you have built models before or never have, I hope to shed new light on models, what makes them work, and what could make them fail.

First Things First

"Organizations want systems. They don't want processes, meetings, models, documents, or even code." —Steve Mellor and Marc Balcer, eXtreme modelers and authors

Models are fantastic tools for ferreting out elusive requirements. But, before you even think about setting pencil to paper or finger to mouse, you need to know *why* you're doing this. Otherwise, your requirements team may never know if they're *done*—and done is everyone's goal.

Every modeling effort should begin with exit criteria, or modeling objectives. For any product, modeling objectives should be consistent with product objectives and the types of diagrams and documents produced. The objectives should be clear, and either measurable or observable. Sometimes evidence that a document exists is sufficient to satisfy an objective; however, you may need quantitative criteria or demonstrations via prototypes or executable models. For instance, if we were developing software to sell to airlines or travel agents, we might have the following objectives:

1. Produce a class diagram (with definitions) that identifies all information of interest and all policies and rules regarding reservations and ticketing. System and business analysts will review the diagram to determine if it addresses marketing, customer service, and regulatory concerns.

2. Develop use cases for the top-ten expected scenarios, and for as many unexpected scenarios as possible.

3. Assure that class data and associations can handle all defined use cases. Marketing, system, and business analysts will review them. If accepted, operations for classes need not be modeled. We will explore behavioral details in design.

4. Reservation subject-matter experts will review and approve all items specific to a reservation itself.

Now that you know how to begin, let's look at some sample diagrams and models.

Building Models with Three Perspectives

"We evaluate each model under both expected and unusual situations, and then we alter them when they fail to behave as we expect or desire." —Grady Booch, UML amigo and designer

A good model possesses the soul of a crash-test dummy: it tests our ideas, facts, and assumptions; it is infinitely adjustable; and it can be tested repeatedly until we discover what we need. People occasionally use the word "model" when talking about just one diagram. But more precisely, a model is made up of interrelated diagrams *and* text, where each provides unique information. The diagrams show a particular perspective, and the text defines distinct elements.

Most diagrams show one of three perspectives: function, information, or behavior. Imagine these as software's three dimensions. Functional views capture procedures, policies, and algorithms that state how things work. Informational views reveal the structure, meaning, and uniqueness of resources, such as contracts, equipment, records, and roles. Behavioral views describe desired and correct operation, as seen in event analysis, use cases, sequences, and conditions. Every diagram will likely fall into one of these categories, regardless of whether it's slanted towards analyzing system requirements or depicting software design.

It would take a book to list all of the possible diagrams devised for depicting software or analyzing systems—and many are available. The three diagrams presented here represent the most commonly used types for each of the three perspectives: data flow diagrams for function, class diagrams for information, and state transition diagrams for behavior. The figures I've shown are skewed toward requirements analysis, and typify initial diagrams. They describe the subject matter that expands on the system's purpose. For that reason, you won't see references to computer language, software design, or any technology that is the vehicle for construction.

A Functional View: Data Flow Diagrams

A **data flow diagram** shows a network of processes. Most people find them easy to follow and easy to draw. The data flow diagram scope varies: it might be a system, a software component, or just the functions for one object. These views

expose functional dependence and independence, which might suggest new possibilities to the software designer. Figure 1 shows a data flow diagram of airline reservation transactions.

In order to make the diagram a model, add

1. A **data dictionary,** which defines all information. Some entries might include:

```
passenger info = passenger name + passenger contact
passenger name = title + first name + middle initial + last name
passenger contact = either email address or daytime phone
title = a text string that allows for titles, such as Dr., Ms., Col., etc.
```

2. Text specifying the transformation steps for each process. Specification text for Check Flight Availability might include:

```
For each flight number, date, and time entered,
Check the Actual Flights for flight status and total seats available.
    If flight status is on schedule and seats available are >0,
    return the flight number and seats available.
    If flight status is canceled,
    return the flight number and flight status.
```

Having the definitions and specifications for all six processes in the airline reservation model would permit more questions, but even with the model fragments you can ask some very useful questions:

- Can a flight status be *delayed*—something other than *canceled* or *on schedule?*
- Where does class of service fit in?
- Do we ever need a passenger's address?
- How do we use the contact information? Where will it be saved?

Having answers might confirm that more detail is needed, or bolster confidence on requirements closure.

You can also use this model to postulate "what if" to explore the possibility that similar, but unstated, requirements might be desired by your customer. For instance, "*What if* the airline wanted to track charges and shipments for extra baggage?"

Another type of data flow diagram is the **context diagram.** A context diagram represents the system with a single black-box process, and our focus shifts to the boundary: all major interfaces, and the people and other systems with which this system interacts. While incredibly simple, this diagram is very powerful.

For instance, one client of mine needed to revamp several old, yet important, systems. Rather than build whole models, I suggested they draw a context diagram for each system, define all interfaces, and write a paragraph describing each system's purpose along with a list of software files used in the system—easy for the people who knew the systems. Once everything was finished, we posted the diagrams on a wall with description files as references. Everyone could see the inter-system dependencies. System A's context showed System B as an external. System B's context referred to System A, and so on. We reviewed descriptions and definitions, adjusting interfaces for consistency and accuracy. Managers used all of this information to establish who needed to be involved, which software needed renovating, and which schedule would cause the least disruption in work.

I recommend data flow diagrams for building project team consensus, gaining clarity on system or component boundaries and interfaces, or working out a troublesome process. When using these diagrams, be sure to define the interfaces in as much detail as possible and include some sort of functional descriptions. I do not recommend these diagrams for analyzing large or complex systems, as they can become unwieldy.

An Informational View: Class Diagrams

Class diagrams shift the perspective from active to passive. Because a system's most apparent feature is its operation, the class diagram's static posture forces you to delve deeper into the system. Class diagrams evolved from data models, which were first used to determine the relational structure of information, or system memory. Object enthusiasts don't like hearing this, but it is true: While you may not care about relational data tables, you should care about class *relationships.* Why? They resolve the rules and policies relevant to your customer's requirements and reveal the pattern unique to a given application. To

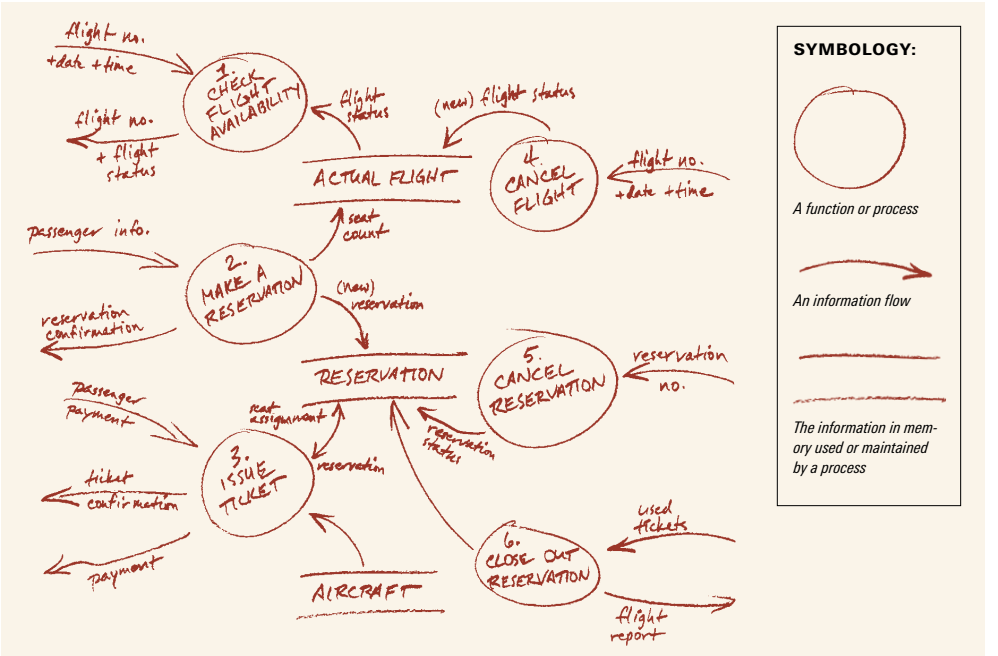

Figure 1: A data flow diagram, such as this one of airline reservation transactions, is easy to follow and easy to draw.

To build a class diagram, first identify the items of interest for your subject matter and their facts.

build a class diagram, first identify the items of interest for your subject matter and their facts. Then, arrange these items into related sets or "classes." Determine the relationship rules that govern class association, and iterate.

Figure 2 shows a class diagram for an airline reservation system. Actual Flight and Passenger are classes. A relationship between them reads "passenger reserves space on one or more (1..*) actual flights" and "actual flight has space reserved for zero or more (0..*) passengers."

To make it a model, add

1. A **class description** to explain the purpose and meaning of the class:

Passenger—an individual who books seats on our airline's flights. This person might travel with the airline once or many times. Information about the passenger is also required for reporting to the FAA.

Reservation—a record of every commitment to reserve space on a specific Actual Flight. The reservation is also the basis for ticketing and building flight records to go to the FAA.

2. Attribute definitions to focus on meaning and characteristics:

ReservationNo—a unique number that identifies an individual reservation in our system.

FlightNo—a number that identifies a flight from one airport to another airport at the same time every day.

PassengerID—a unique number that identifies an individual passenger.

Price—the price for a reservation based on when the reservation was made, the class of service, and the magic daily formula. Default format is U.S. currency.

3. An **association definition** for "passenger reserves space on one or more (1..*) actual flights" and "actual flight has space reserved for zero or more (0..*) passengers" to explain the rules that apply to each class. For instance:

Every Passenger that we keep on file must have at least one reservation association. A Passenger may have a reservation for more than one Actual Flight, and may have more than one reservation for that same Actual Flight.

Every Actual Flight doesn't have to be in a reservation association, since some flights might not have space reserved. An Actual Flight may have more than one reservation for a given passenger.

4. An **operation definition** such as:

Accept Payment (ReservationNo, Payment, Confirmation)
  Check the price for this reservation.
  If the payment = price,
    Issue a system-generated confirmation number as Confirmation.
  If payment ≠ price,
    Return a message that indicates the correct price.

Even with just these model fragments, you can ask the following questions:

- Even if a person doesn't hold a reservation, shouldn't we be keeping past passenger information for marketing purposes?
- Do we keep credit card information?
- How do seats get assigned? Are we missing a class?
- How exactly is the price calculated? Do we have sufficient information?
- Is every Actual Flight scheduled *every* day? What about weekends or holidays?

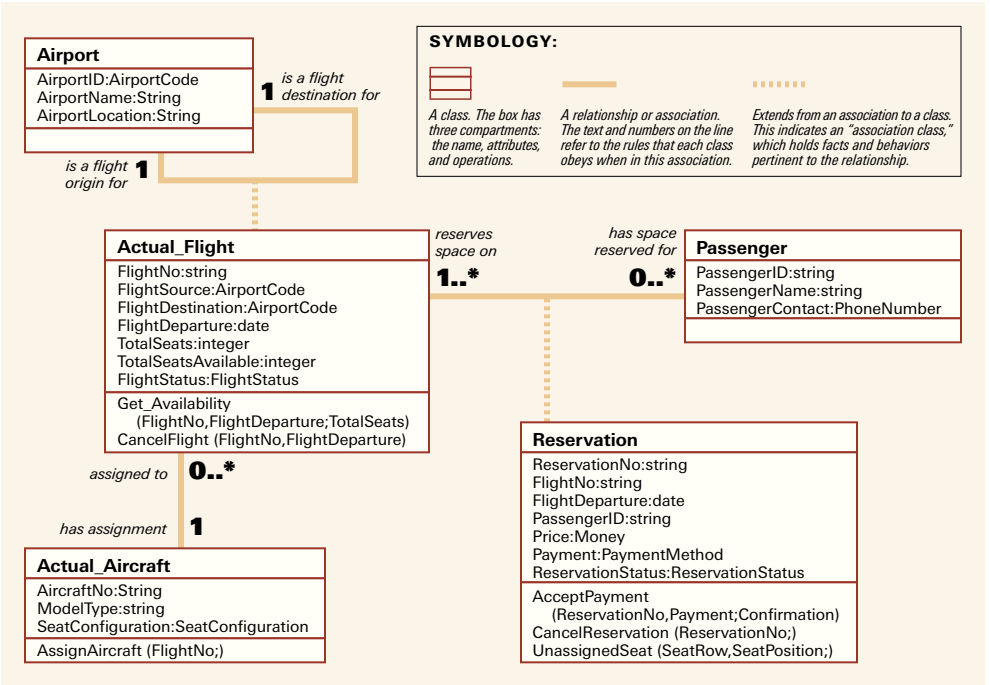

Figure 2: If you are going to build only one model, start with a class diagram such as this one.

Class diagrams aren't layered. A very large system might be divided by subsets of the larger subject matter. If we were to model airline operations, the reservation subsystem could be on one diagram, flight management on another, crew management on another, and so on. A specific class might appear on more than one subsystem. Couple this model with behavioral views and you would have a solid foundation for development.

If you are going to build only one model, start with a class diagram. Most system partitioning principles I have learned recommended separating functions by natural sets of data, or subject matter. Systems built on this principle are the cleanest—meaning easiest to maintain and understand. Clean partitioning comes from appreciating the information structure beneath the system dynamics. Building a class diagram provides the added benefit of establishing a common vocabulary among the development team.

A Behavioral View: State Transition Diagrams

The third perspective, behavior, explores necessary and expected system operation. If you want to describe modes of system or component operation or an object lifecycle, the **state transition diagram** is the tool for you. A simple example of a state is the power mode for your TV: the two states are *on* and *off*. Pressing one switch takes you from one state to the other. However, most states aren't that simple. Imagine what you would need to describe the modes of a space flight. Even the lifecycle of your mortgage involves more than you might initially think.

State transition diagrams have been around for a while. They were first used in software modeling in the mid-eighties for describing real-time software control. As object modeling arose, this diagram best described object or class lifecycles, and the events or conditions that triggered a change in those states. A state transition diagram may be part of a model definition, expanding on another diagram's perspective. In the

PERSPECTIVE

A Tester's View of Models

Harry Robinson is a self-described "big promoter" of model-based testing. He is officially a Test Process Engineer for the Operations Management Team at Microsoft Corporation. The model-based testing at Microsoft starts with requirements. "The first thing we do with the requirements is take natural-language text and convert that into what are typically state transition diagrams." These hand-drawn whiteboard diagrams help them understand system behavior, clarify ambiguities, and allow several testers to brainstorm simultaneously about the requirements. The whiteboard is digitally photographed for later reference, and the completed diagrams are manually entered into a test generation tool that converts them into actual test cases.

Microsoft's requirements modeling takes place well before development. Harry mentions that at first, spec writers were a little nervous about having their requirements reviewed in such detail, especially specs they didn't consider "done yet." But soon people realized that the testers were making the spec better, not making the spec writer look bad. Harry explains, "We can point to a state in the diagram and say, 'I'm at this point in the application. What action can I perform from here? What happens when I perform that action?' You can actually see places where you don't know what happens next." Looking at requirements in this way is helpful in finding holes and ambiguities before development ever gets involved.

"For example, we had one program that saves pictures to a folder, using a wizard. As we were tracing through the model, we realized there was a place in the wizard where it was possible for the user to go to the folder and delete the pictures that had been saved earlier in the wizard. Therefore, we needed to have some error checking put in, because even though that wasn't the way a typical user might approach using that wizard, it actually was a feasible scenario for someone to do. Something like that might not pop out at you looking at the natural-language text requirements, but as soon as you see it on the model, you think, 'What if those files weren't there?'"

Using models to improve software quality may not be considered traditional software testing, and that might make some testers nervous at first. "If you prevent a bug, it improves software quality but might not necessarily be considered testing. For instance, since we're forward in the process, the software that comes into system test officially has already been through this process, with lots of clarification up front. So people started delivering some beautifully clean software, with very few bugs in the module. One tester who had helped eliminate spec errors earlier in the process looked at one of these clean modules and said, 'Oh no! I'm screwed!' You have to remind the testers that their job isn't really to find bugs—it's to get good software out the door."

That re-education doesn't stop with testers; it extends to management as well. "Finding bugs is always a bittersweet thing. It's good that you found a bug, but how did it get that far in the first place?" For Harry, the best way to wipe out bugs is to find them before they're even created, when they're only drawings on the whiteboard. —*R.T.*

For more on Harry Robinson's approach to model-based testing, see his article "Intelligent Test Automation" from the Sep/Oct 2000 issue of this magazine.

case of Figure 3, you are learning more about the Reservation class. You can assess the actions, and also whether the class diagram detail supports the needs of those actions.

To make a state transition diagram into a useful model, define the actions that apply to each state and make explicit the transition conditions or events. If a state model doesn't correlate to another diagram, then add definitions for any information referred to by the actions. By walking through a complete state model's action statements and control paths, you will be able to look for missing or conflicting steps, spotting problems you might not have noticed before.

State transition diagrams also correlate to use cases and event analysis. A use case may require several classes to play it out. State transition diagrams for each class would need to accommodate those operational needs and also coordinate with each other to produce the desired results. If you started with a given event or trigger and followed the chain of necessary behavior, you would be looking at a pattern similar to a use case—the classes and their state transition diagrams need to account for the chain of events and responses.

Even with an initial diagram, you can ask the following questions about conflicts or missing pieces:

- How does a passenger get reimbursed when a reservation is canceled?
- How do we know whether a price is still valid or not? Is a create date needed in the Reservation class?
- Should the price calculation be in Establishing a Reservation?
- Is a ticket confirmation a "ticket"?

I have found state diagrams to be indispensable. They identify class operations that are contingent on conditions, versus ones that are totally independent. An independent action is: Get Flight Status. Regardless of what state Actual Flight might be in—canceled, on schedule, or unscheduled—you should be able to inquire about its status. Your software can get in real trouble if an operation's dependence or independence remains unspecified. State transition diagrams eliminate this confusion.

General Model Guidelines

"The specification should, as far as is practical, be free of ambiguity." —James and Suzanne Robertson, requirements and modeling masters

Whatever diagrams you use for modeling, these guidelines apply:

Define all terms on the model. A diagram alone may be interesting, but without supporting text, you can't be sure what the words mean. Definitions are intended to make meaning clearer. If others can't understand or find use in what you're saying, it's likely that you didn't fully understand that part either. Ambiguity and lack of definition are the top causes of modeling problems and wasted time!

Check for **consistency** among the documents. Conflicts or missing pieces are signals that a question needs to be asked and answered. For instance, what if the text for the Make a Reservation process contained a reference to passenger payment? Should the payment information be collected when a reservation is made? Or, is the diagram correct in showing that a reservation can be made without payment, and that a payment triggers the issuance of a ticket?

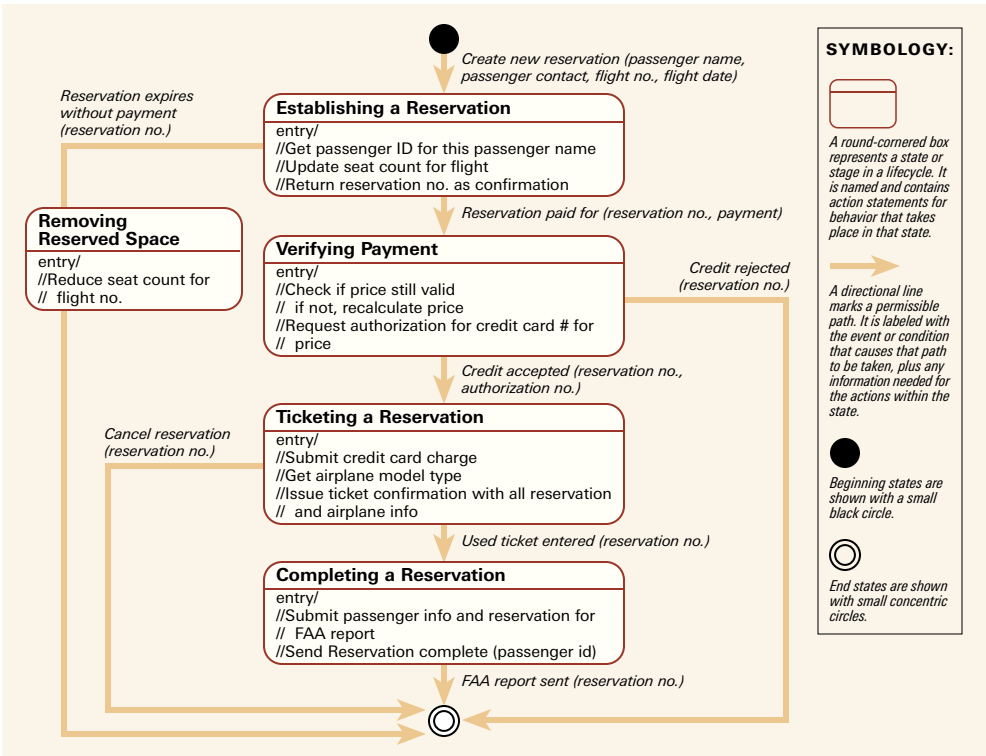

Figure 3: State transition diagrams eliminate confusion by identifying class operations that are contingent on conditions, versus ones that are totally independent.

Eliminate sloppiness with **exact references.** For example, what if you saw a class named Reservation and a state transition diagram for Passenger Reservation? Are they the same? Sometimes this points to a conflict in the understanding of what is needed or wanted, but sometimes it is an oversight.

All models should be **readable.** You need to have some sense of what is being presented. The icons usually don't get in the way of readability, but the words could. Take time with the names you choose. Keep in mind that notation can never save a poorly labeled model from disaster.

Practice Cultures and Model Usage

How you use models and declare them done depends on the environment you work in and the kind of software you produce. In "But Will It Work for Me," Kathy Iberle identified eight software engineering practice cultures, their business models, and their assumptions. For our purposes, I've organized these eight cultures into four groups.

PRODUCT	DRIVING FACTOR	MODEL USE
Open-source software or software for academic research.	Personal goals	Might draw pictures; probably will not use models.
Software intended for commercial or business use and consumer products, such as games and cell phones.	Time-to-market	Schedule pressure may discourage people from modeling, but models can be built incrementally and capture valuable market knowledge.
Software supporting company operations, including Web-based applications.	Policy and politics	Models can facilitate discussion, frame details objectively, open up communication, and provide documents for negotiation, agreement, and sign-off.
Custom systems on contract and commercial or mass-market firmware. Software is large, complex, and safety-critical, with examples being vehicles, medical equipment, or command-and-control software.	Risk containment (penalties, safety, security concerns, regulatory approval, or all of the above)	Natural fit for models, because work habits with high-risk systems must be methodical and thoughtful. Advanced practices, such as executable modeling and code translation, succeed here.

Most importantly, a model must be **useful.** If those people interested in using requirements don't see how the model relates to their work, then one of a number of things may be at fault: 1) no one really knows how to build a model (lack of training or support); 2) no one ever refers to the model (process and policies don't support models as exploratory documents or central references); or 3) no one understands what this has to do with producing code (the relationship between requirements documents and code has never been addressed).

Final Advice

"Poor analysis yields useless models. You have been warned."
—Leon Starr, modeling maven and engineer

To add to Leon's pointed advice, I recommend:

1. Perseverance pays off. Using a graphic language may feel strange at first. Read other people's models to learn about expression and style, and to evaluate what makes sense and what doesn't.

2. Be realistic. A model's quality cannot exceed that of analysis skills or practice culture. What do you want people to be able to do with models? Do they have the skills to do that? How will this change your current process? Can this work for your schedule, budget, and culture? Start with pictures if you aren't sure about modeling. Perfection is *not* your goal—usefulness is.

3. Dare to be wrong! This modeler's rallying cry from my colleague Linda Nadeau will remind you of modeling's purpose: You *want* to be able to find faults. If someone finds a flaw in your model, rejoice! You just discovered something to eliminate, reanalyze, or maybe just express more clearly.

4. Use tools effectively. Be pragmatic about what you expect, what you can afford, and how it would enhance your process and product. This may mean recording whiteboards and audiocassettes, or it may mean model compilers. Don't buy an automated modeling tool just because everyone else has it or it fits your budget. If you're just starting with models, it pays to be simple. Once you have experience, you will be a better judge of what you really need and want.

5. Manage your models. Expect models to change over time, and be able to change them. Set up version control even if it's just writing the date on the whiteboard that you copy.

I hope you appreciate *your* role in the effective use of models. As with software, the success or failure of a modeling venture will depend on the choices you make for your situation. Take methods advice, notation, and processes for what they are—something that worked for someone. Understand your culture, and incorporate advice, graphics, tools, and processes that could truly work for your project team. Just picture what you could do! STQE

Becky Winant has coached thousands of people to develop useful models, set realistic plans, and find the requirements they need. You can reach her at becky@beckywinant.com.

***STQE* magazine is produced by Software Quality Engineering.**