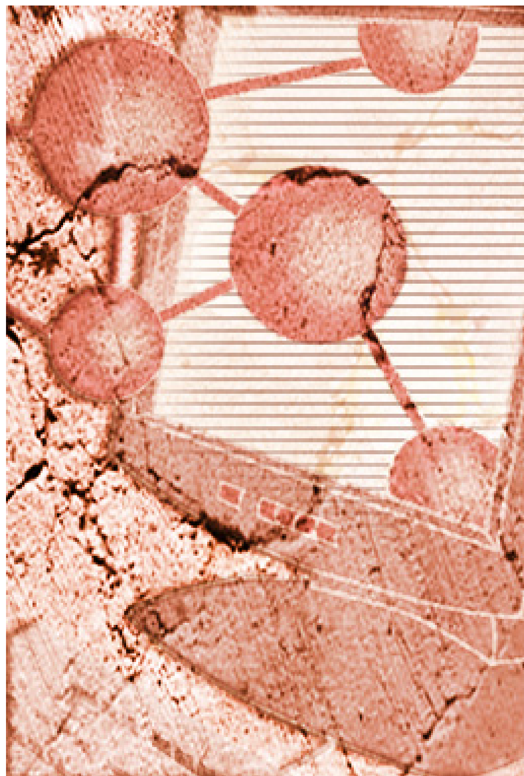
The art of modeling

Part II: Model organization and construction

by Ben Lieberman, Ph.D.
Principal Architect
BioLogic Software Consulting, LLC



In Part I of this series, we explored some of the theoretical underpinnings of modeling and discussed how to analyze and conceptualize models for particular audiences. In this installment, the focus will shift toward practical examples of model construction, with an emphasis on software modeling.

The form any model should take depends heavily on its intended purpose, which can be to investigate, instruct, evoke an emotional response, and so forth. For example, a model of an unproven scientific theory would be investigative, so its content should be based on prior knowledge of the domain as well as the predicted behavior that experimentation will be expected to verify. A model for the construction of a building is instructive; it should focus on the structure's physical systems (e.g., structural, plumbing, electrical, cooling/heating, etc.). We can think of many artistic endeavors as a form of modeling as well; the artist's intention is to convey a sense of feeling or emotion by copying shapes, movements, and sounds from the world around us. And artists choose the modeling medium best suited to their vision (e.g., sculpture, dance, music, performance). In addition to having a specific purpose, form, and theme, all effective models should have an organizing principle, or "pivot," that determines the focus of each model view.[1]

This article will describe how to create effective models, and how to discover and capture model elements. We will look particularly at software

development models, with an eye toward the third part of this series, which will cover the aesthetic qualities of visual modeling in software development.[2]

The organization of a model

Models take as many forms as there are people who envision them. We build external models based on the internal mental models we begin developing at birth, so each model is influenced by a person's individual personality and experience. Therefore, a modeler should understand his or her own biases, and also try to compensate for the differences in understanding between people. Otherwise, these biases can result in models that are overly complex and difficult to interpret, or models that are too vague to be useful.

Indeed, all models that we create to communicate with other people should be concise, correct, and clearly organized. Complex models should include an *index of views* (see **Model views and the view catalog** below) that describes the model structure and helps users locate specific information.

Establishing a purpose

In addition, every model should clearly identify its intent. As Table 1 shows, people use models of many forms for many purposes; some even use models to describe models (known as "meta-modeling"[3]). Ideally, a model will serve only one purpose, although in rare cases a single model may serve multiple purposes. For example, in software modeling, a use-case model is intended to provide information for testing, project management, design/construction, and customer acceptance.[4] However, this is the exception rather than the rule; a well-constructed model usually serves a single, well-defined purpose.

Table 1: Example models and their purposes

Example models	Purpose
Mathematical models for astronomical or economic predictions	Predict
Written and spoken language; icons; specialized terminology; charts and graphs	Educate/communicate information
Architectural building plans; UML diagrams for software development	Plan construction
Scientific theories; mathematical/logical symbology	Investigate/illustrate reasoning
Library indexing; geographical charts/maps	Aid navigation

Painting; dance; music; drama	Evoke emotional response
UML meta models,[5] cognitive models	Describe/explain other models

Establishing a form

As communication occurs between people via specific channels -- our five senses: visual (optical), sound (auditory), taste (gustatory), scent (olfactory), and touch (tactile) -- a model whose purpose is to communicate should leverage one or more of these channels.[6] Although most models are designed primarily for a single channel, some use multiple channels at the same time. An audio-visual presentation, for example, might include charts that use both spoken narrative and images to convey information. In software development, models use the visual channel almost exclusively to present detailed information.[7]

Table 2: Examples of models for specific communication channels

Communication channel	Model Examples
Touch (Tactile)	Braille
Scent (Olfactory)	Perfume formula
Taste (Gustatory)	Wine category, catering recipe
Sound (Auditory)	Morse Code, spoken language
Visual (Optical)	Written language, blueprints, painting

A great number of model forms have been created to take advantage of the human visual capability, because a tremendous range of differentiation is possible based on the shape, color, and texture of visual modeling elements. The most common form of visual model is a text-based representation. Examples include musical scores, scripts for screen/stage plays, manuscripts for novels and other publications, and virtually all software development documents. In addition to text-based representations, other common visual modeling systems include:

- UML (visual language for software development modeling)
- Molecular structural representations (e.g., spatial models)
- Civil construction blueprints
- Dance movements
- Semaphore signals
- Visual business models (e.g., organizational models, flowcharts)

- Icons (e.g., universal graphics for traffic control, bathrooms, laundering instructions)

Establishing a theme and pivot points

In addition to a clear purpose and form, a well-constructed model should have a central *theme*, or organizing principle. Think of how the theme of a novel helps the writer create a cohesive work and also guides the reader along. For a model, the theme sets boundaries and helps you determine what to include in, or exclude from, the model. For example, if your model's purpose is to help control air traffic, and the form is visual tracking via display screens, then the theme, or organizing principle, may be that you color code aircraft by type and activity (take-off, landing, climbing, descending, converging course, etc.) and provide textual descriptions of speed, direction, altitude, and so forth. Because the terrain over which inbound or outbound planes are flying is usually not relevant to the air traffic control tower, you would probably exclude (or filter) this information from the model. In a software development model, if the purpose is to describe object classes, and the form is UML class diagrams, then the theme, or guiding principle, will be to show structural information such as association and containment in a structural model view. You would exclude the dynamic aspects, such as invocation of a method, because they are not part of the theme for the current model or model view. Instead, you can capture this information in a separate model view focused on object behavior (e.g., a sequence/interaction model).

Finally, each model view (see **Model views and the view catalog** below) should contain a specific basis for information that I call a *pivot*. This is a category upon which the rest of the view information will depend. Select a pivot by deciding what information is most important to the model view. For example, in a musical score, the pivot points are the time signature (beat) and key signature (pitch). The remaining information on volume, speed, and so forth, is secondary to these two central elements. In a use-case model, the pivot may be a specific area of functionality, such as mutual fund trading in a financial services model, or billing in a telecommunications model. In Figure 1, the pivot is *trading*, so the diagram gives trade activity a central location and a "hot" color (yellow) to draw attention; the supporting use cases that surround the trading function are in a "cool" color (blue), indicating that they are not the focus of this view.[8]

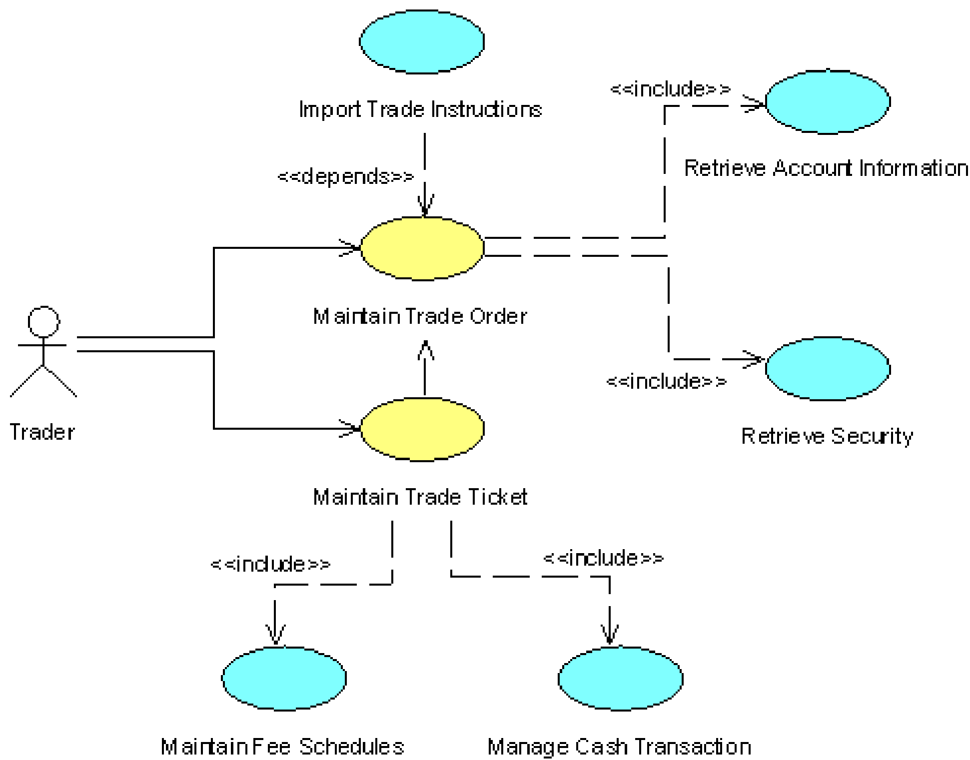
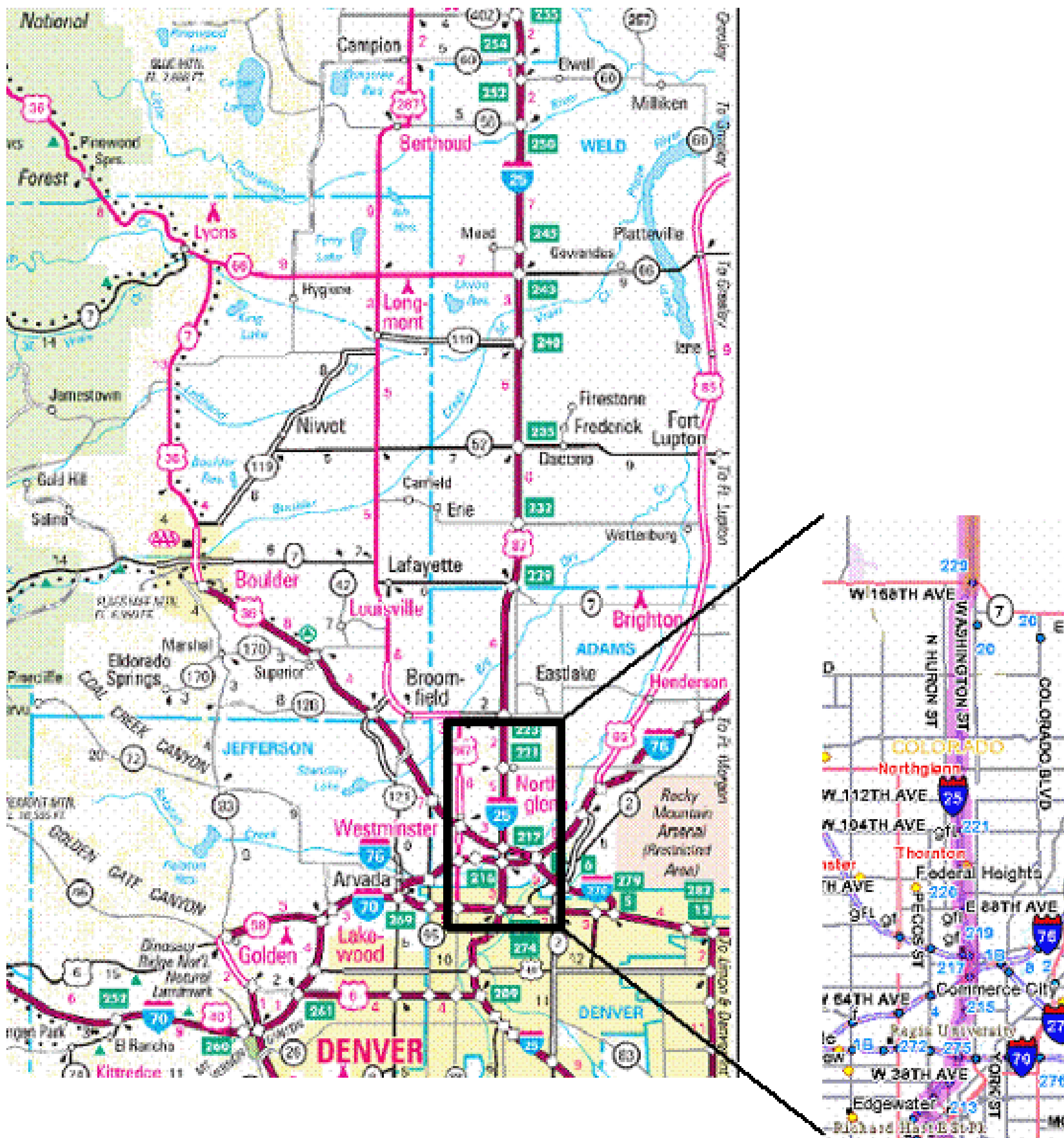


Figure 1: Trade use-case model

Time is often a pivot point for model views. Uses range from the obvious, such as historical timelines, to the more subtle, such as astronomical charts in which time is central but obscured by the description of stellar object positions.[9] Another common pivot, *position*, is often the organizing point for geographic models such as a road atlas (Figure 2) or a coastline navigational chart.



Source: American Automobile Association
Figure 2: Navigational model: road map and TripTik.®

Sometimes finding a good pivot for a model view is a challenge, but because the view will key off this organizational point, it is worth spending time to experiment with different pivots until you find the best one. As a rule of thumb, a pivot should be a common abstraction shared by all information in the model (see **Model construction** for more on finding these abstractions).

A final point to consider is context. The information in any model exists within a greater context, or environment, which you need to take into account. For example, you would model a car in the context of a highway in a significantly different manner from a car sitting in a showroom. Elements critical to the highway context include speed, position, road composition, traction, weather conditions, fuel consumption, engine temperature, etc. In a showroom context, however, critical elements for the same vehicle would include color, amenities, reliability ranking, design, fuel efficiency, and price. Similarly, for software development models the context of the project will determine which details to include or highlight.

A data analysis model in the context of a business domain should focus on the data elements and containment relationships of the problem domain, whereas the same data model in the context of a database implementation should focus on construction details such as the size and type of each data element, primary/foreign key relationships, data integrity validations, and so forth.

Model construction

How do you go about creating a model? Some people take a top-down, general-to-specific approach, whereas others start with detailed, specific examples and work up to the more general case. Still others start from some convenient place in the middle and work toward concrete examples and general abstractions at the same time. The most proficient modelers are comfortable with all three approaches and use them as the subject demands. (For more information on these three approaches to model design, see the section **Systematic approach: The case of software development**, in Part 1 of this article series.)

Depending on the nature of the problem and your familiarity with the subject, you should approach the creation of your model with a good idea of the purpose and context, and at least one or two candidate forms for capture and presentation. Choose one of the three approaches noted above for your initial investigation of the subject domain. I often use the middle-out approach to gain a rapid understanding of the scope of the problem and to get a holistic view of the different levels I need to capture. For example, if I am creating a business workflow model, I will talk with some of the workers and create a few business flow diagrams (i.e., variants on activity diagrams) to get an idea of where the more complex workflows are (middle-out). Then I will consult with the business managers to gain an understanding of where these flows fit into the overall business (top-down). Finally, I may return to the original workers for a more detailed look at the data and tools (e.g., software systems) that are currently used to conduct the work (i.e., get a bottom-up view). Once I have a top-to-bottom view of at least one representative part of the system, I can select a candidate model form and set of views to complete the work.

Find the key abstraction and select a theme

No matter which approach you select for investigation, it is critically important to look for the essential characteristics of the subject and determine whether they should be included or excluded from the model or model view. A well-chosen *key abstraction* can help you select the correct theme for the model and the most effective pivots for the model views. A key abstraction should represent the essential commonality within a collection of elements based on the modeling subject. In the trading example shown in Figure 1, the key abstraction is the movement of financial instruments from one owner to another. The mechanism that will accomplish this central task is the use of Trade Orders to define the trade details, and Trade Tickets to track the execution of multiple related orders. The Trade Orders and Trade Tickets will be supported by all of the surrounding functionality, such as Lookup Security and Manage Cash

Transaction (for tracking funds to pay for the trading transaction or receipt of sale proceeds). Thus, the model theme is based on trading functionality, and the pivot for the view in Figure 1 is Trade Order/Ticket.

Finding a key abstraction is often one of the most difficult aspects of modeling, since it requires consideration of the "big picture" to see how all of the system components interact. I have found that it helps to identify many of the data elements in a problem domain (e.g., the "nouns" of the problem domain) and the relationships between them (e.g., the "verbs" of the problem domain) when searching for common themes. By identifying these elements and the relationships between them, the overall nature of the problem to be modeled becomes clear.

Discover elements

[Editor's note: In Donald Bell's article on the UML class diagram, also published in this issue (November 2003), Bell uses the word entities *to refer to "the things that make up a model's contents." Here, Ben Lieberman uses the word* elements *to refer to the same things. In editing their respective articles, I saw no reason to insist that both writers use the same word.]*

The elements of a modeling subject are usually identifiable as the "things" that make up the domain. For example, a model of a water dam may be composed of elements such as Gate, Valve, Pump, Pipe, Sluice, Inflow, Outflow, and so forth. As you identify each element, it is a good idea to capture some of its relevant characteristics (i.e., in UML terms, these are the "attributes" of the element). For example, an attribute of the Gate element may be Position State, which may take the value of Open, Closed, or In Motion. Another attribute may be Pressure, to indicate the current hydraulic pressure on a closed Gate. In this way much of the problem domain can be systematically investigated and captured. If you are an astute reader, you may have noticed that we have already achieved a first level of abstraction by calling a problem domain element a Valve, rather than "regulator valve #14, inflow pipe #23a." This is important, because there may be hundreds of valves that share similar characteristics. Capturing a detailed description of each one would defeat the purpose of modeling, which seeks to capture the overall characteristics of a water dam system.

However, if you want the model to provide construction-level details (e.g., blueprints), then each valve does require a specific name, type, and location. Again, the model form and abstraction level you use will depend on the purpose of the model. If a software system for a valve control mechanism is under construction, then you can produce a domain model like the one in Figure 3 to describe how the control valves fit into the overall water dam system.

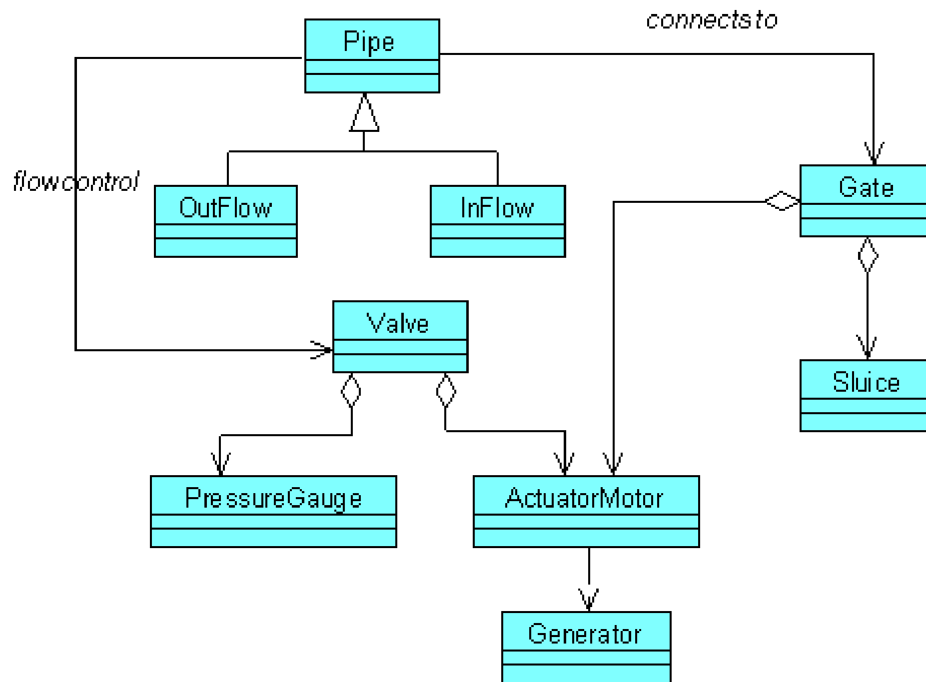


Figure 3: Domain model of a dam control system

Domain models are very useful in the discovery phase of a software development effort, because you can create and modify them quickly, with a minimum of development/design tools. Many domain models are assembled out of nothing more than a series of index cards taped to a board. Because data manipulation represents a vital part of defining system requirements, simple techniques such as this aid in developing those requirements. For more information on creating domain models, see the works by Evans, Silverston, and Menard, respectively, listed under **References**.

Discover relationships

As you discover the elements of a problem domain, it is likely that specific relationships between these elements will emerge. Often, you can organize these relationships according to the dependency between system components. Although there are many different types of relationships among elements, I have found four that are particularly useful for organizing system elements in software development (Figure 4).

1. ***Atomic relationship***
   In an *atomic* relationship, an element is completely independent of all other elements and is sensible entirely on its own. Atomically related elements are typically the subsidiary elements of a containment tree that itself represents a whole/part relationship (a type of relationship also known as containment).

2. ***Whole/part relationship***
   In the whole/part relationship, the related elements are not relevant to the system outside the context of the owner. For example, in the context of a purchase management system, an Address or Phone Number element is not sensible without some owning element such

as a Customer or Business.

3. ***Reference relationship***
   In a reference relationship, one element refers to another but is not the element owner. For example, an Order logically refers to a Customer, but the Customer can exist in the system without any associated Orders.

4. ***Abstract relationship***
   In an abstract relationship, a general element is refined to more specific elements that inherit all characteristics of the general element. This characterization of elements by their common attributes was investigated quite a long time ago (e.g., by Plato) and is a highly useful technique for object-oriented programming of software systems (see works by Lakos and Mattsson under **References**).

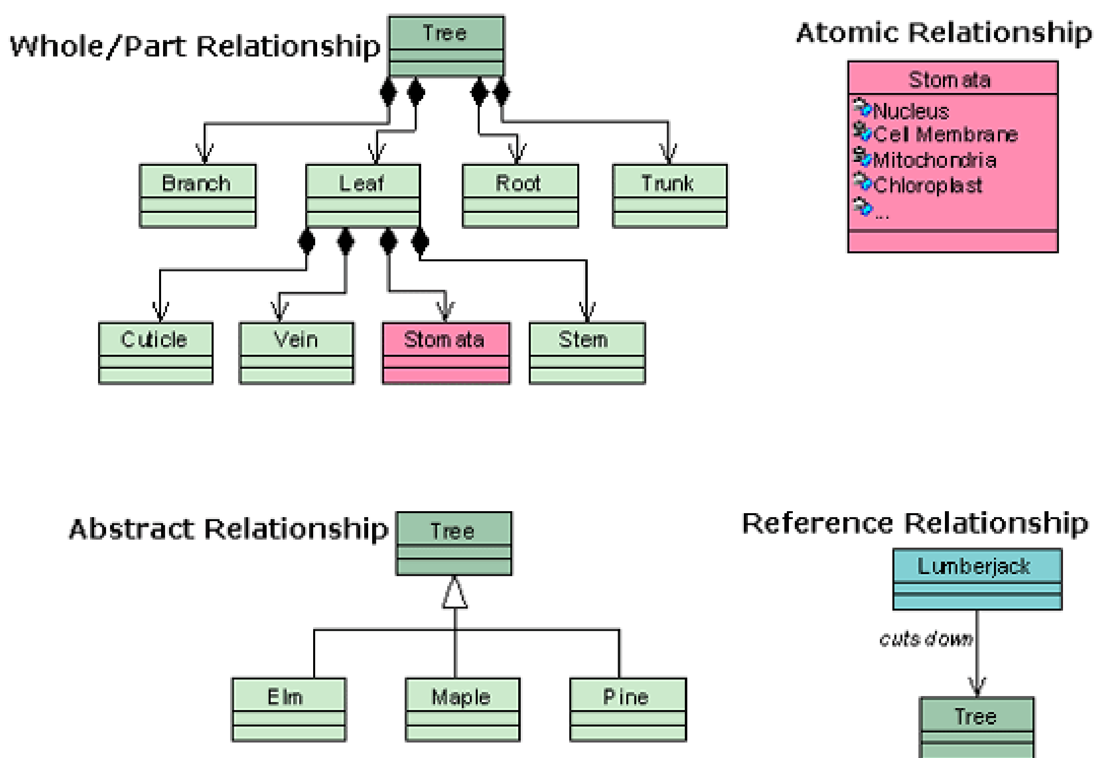


Figure 4: Examples of element relationships using UML class diagrams

As you discover elements, you can organize them into a domain model, using the relationships illustrated in Figure 4. As long as you have established the model's purpose and theme, you can determine and describe the necessary level of detail. For example, note in Figure 4 that *Stomata* (in the atomic relationship) is the only element shown with attributes. That is because this model was intended to illustrate relationships, and Stomata is the only atomic element that has related, defining attributes.

Capturing data and relationships will help you to rapidly capture the nature of the domain you want to model. This approach is particularly well suited to software development, because the modeler ultimately needs to

translate the discovered elements and relationships into a functioning software system.

Model views and the view catalog

For software development, most models will be composed of multiple views created for a widely varying audience; these views express related but independent aspects of the subject. Each of these views typically consists of a diagram narrowly focused on a single pivot, as we discussed above. In the UML, views are broken out as follows:

- Structural and dynamic views for system design.
- Use Case views for describing system functionality.
- Construction views for component creation.
- Deployment views to describe system installation.

Each of these views addresses a different aspect of the overall project, and each segments the problem domain to focus on a single aspect at a time.

A catalog of these views permits those who use the model to rapidly move from one related model to another. The catalog can be composed of either a simple figure index or a more complex listing of all model views. Modern software modeling tools often automatically generate such an index as part of the software functionality. However, if you are not using such a tool, or if the tool you have is incapable of automatically generating a catalog, then you should provide this listing manually for users.

Consider a model of a particular molecule. Depending on its physical characteristics, you can model it in many different ways: by filling visual spaces to represent atomic structure; by showing a wire-frame of chemical bonds; via a text-based chemical formula; or by reporting its reduction potential, hydrophobicity index (e.g. inability to dissolve in water), and so forth. The particular form and view you choose to model depend entirely on what information is of interest to you, the modeler, and your audience (Figure 5). The technique of separating software models into multiple specific views is discussed extensively in books by Kruchten and by Clements, Bachmann, et al., respectively, and in a committee-authored IEEE article on recommended practice for architectural description, all listed under **References**.

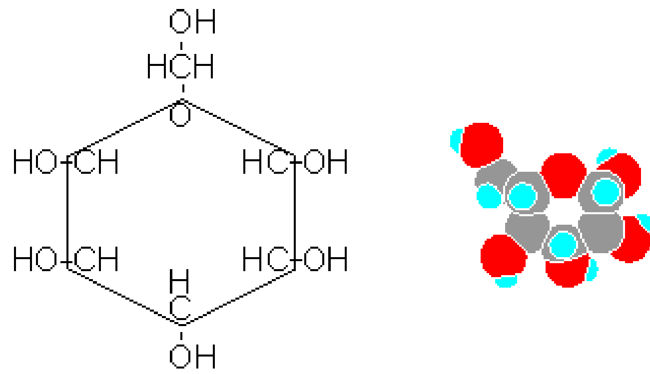


Figure 5: Chemical structure and spatial model of the sugar glucose

Editorial review

Models, especially software models, should typically be *created* by no more than three people, but they should be *reviewed* by as many as is practical. Remember that old joke:

> Question: What's a camel?
> Answer: A horse built by committee.

Indeed, a model that is constructed by more than a few people will become muddled and confusing because of differences in understanding and experience.[10] However, just as a manuscript can be vastly improved by a good editor, so can a model be improved by a thorough review. A software development model, be it for requirements, design, construction, or deployment, should be reviewed by *at least* two people during its elaboration. This will ensure that at least three points of view have been considered (those of the modeler and each of the reviewers), and that errors of omission or inaccuracies are addressed.

Reviewers should also look for clarity in presentation, as represented by well-organized model views. Each model should have an easily identified purpose and theme, and each of the model diagram views should have a clear pivot data point. When do you know that a model is of high quality? When a reviewer does not need to go back to the modeler with questions not answered by the model itself.

Conclusion

The first article in this series presented the *why* of modeling, focusing on the capture and communication of ideas from one person to another. This article has focused on the concepts behind model creation as they pertain to software development. Effective models are built to meet a specific purpose, have a well-defined form, and present information based on a common theme. Large models are typically displayed using a variety of views, with each view having an organizing principle, or pivot, around which the view is centered. Finally, model construction should be focused on the elements and relationships of the subject under study, especially in the early stages when, in choosing the best model form, it is critical to

determine a common abstraction between elements.

In the third and final article of this series, I will explore visual presentation techniques for software development models, using UML for examples to illustrate effective use of color, balance, structure, composition, and form.

References

P. Clements *et al. Documenting Software Architectures.* Addison-Wesley, 2003. Architecture Working Group of the Software Engineering Standards Committee, "Draft Recommended Practice for Architectural Description." IEEE P1471/D4.1, 1998.

E. Evans, "Deconstructing the Domain: A Pattern Language for Handling Large Object Models." EuroPLoP, 1999.

P. Kruchten, "The 4+1 View Model of Architecture." IEEE Software 12(6): 42-50.

J. Lakos, *Large-Scale C++ Software Design.* Addison-Wesley, 1996.

D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach.* Addison-Wesley, 2000.

B. Lieberman, "Putting Use Cases to Work." *The Rational Edge*, February 2002.

M. Mattsson, "Object Oriented Frameworks: *survey of methodological issues.*" Licentiate Thesis, Department of Computer Science, Lund University, 1996.

R. Menard, "Domain modeling: Leveraging the heart of RUP for straight through processing." *The Rational Edge*, June 2003.

J. Morgan, and P. Welton, *See What I Mean?* Edward Arnold, 1992.

J. Rumbaugh et al., *The Unified Modeling Language: Reference Manual.* Addison-Wesley, 1999.

W. Schramm, *The Process and Effects of Mass Communications.* University of Illinois Press, 1954.

L. Silverston, *The Data Model Resource Book, Vol. 1: A Library of Universal Data Models for All Enterprises.* John Wiley and Sons, 2001.

E. R. Tufte, *The visual display of quantitative information.* Graphic Press, 2001.

Notes

[1] The use of the term *model view* to describe the presentation of a particular section of a model does not imply a limited visual presentation. Just as musical scores are often split into movements with different tempos, keys, dynamics, and so forth, models are often divided into views that focus on different system qualities.

[2] As an aside, it is important to remember that a software program is itself a model of a real-world process; it is therefore imperative that software developers have a deep understanding of the principles of modeling.

[3] The Unified Modeling Language (UML) uses just this kind of meta-model in a self-referencing manner; see the work by J. Rumbaugh et al. listed under **References**.

[4] See **References**: Leffingwell and Widrig, 2000; and Lieberman, 2002.

[5] For example, models about the process of modeling.

[6] See **References**: Schramm, 1954; and Morgan and Welton, 1992.

[7] It is amusing to consider other forms of communication for software development. For example: What might a software program smell like? How should a use case taste?

[8] The third article of this series will address the aesthetic qualities of model presentation, including color, balance, composition, and layout.

[9] See **References**: Tufte, 2001.

[10] See the first article in this series, "Constructing an analytical framework," for more information on individual influences on models.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2003 | Privacy/Legal Information