2015

An examination of the complexity and comprehensibility of various software models

Taran Staal
Grand Valley State University

.

An examination of the complexity and comprehensibility of various software models

Taran Staal
Grand Valley State University

Introduction

In a discussion of the creation and evolution of the statechart, David Harel, the creator of the model, talks about his primary goals for making a good model. He emphasizes that a good model should be clear, precise, visualizable, and executable. A clear model is one that can be easily understood by someone unfamiliar with the system being modeled. A precise model shows the entirety of the system under consideration, without including extraneous information. A visualizable model is one that can be interpreted by a user using primarily visual information, meaning some words must exist for thing like labels, but the majority of the information contained in the model is expressed through symbols that are easily understandable. An executable model is one that can be developed and then applied to help users understand or even create the system that is modeled.

These ideas are indeed very important to the usefulness of a model, as a model that is lacking in one or more of these areas could be difficult to understand, use, or both. Today, however, there are dozens of different models that aim to show different aspects of systems, and many of these models may not have been designed with the same goals that Harel outlines.

This paper aims to examine the complexity of various software models, including UML models and other commonly used models. I will look at each model and attempt to determine areas of the model that are particularly hard to understand, or areas where ambiguity is possible within the model. I will discuss the implications that model complexity has on a persons ability to learn the model and to recall it at a later date. I will also discuss ways to improve a persons ability to remember the aspects of a model when they are an infrequent user or when they have not encountered that model in a long time.

Examination of models

It would be prudent to begin by examining the basics of UML, as that is the most commonly used method of modeling software. UML 2.5 separates diagrams into two kinds, structure and behavior diagrams. Structure diagrams include things like class diagrams and object diagrams. Structure diagrams are meant to show the structural components of a program, and how those components relate to each other. Behavior diagrams include things like use case diagrams and state machine diagrams. These behavior diagrams are designed to show the complex behavior of a system, and how the system is able to change over time.

Right away we find one example of an area where complexity may make things difficult to understand. Figure 1 shows how all the various UML diagrams are related to each other. The black diagrams are official UML 2.5 diagrams, while the blue diagrams are unofficial diagram types that see use. Obviously, in order to fully model a system, it is necessary to use many of these different diagrams in conjunction, and even an experienced software engineer could have difficulty forming an accurate impression of a system from so many varied diagrams. Unfortunately, while this issue is easy to diagnose, it is not easily addressed, as simplifying these diagrams to something that would be easily understood at a glance would likely lead to a diagram that does not actually convey any useful information.

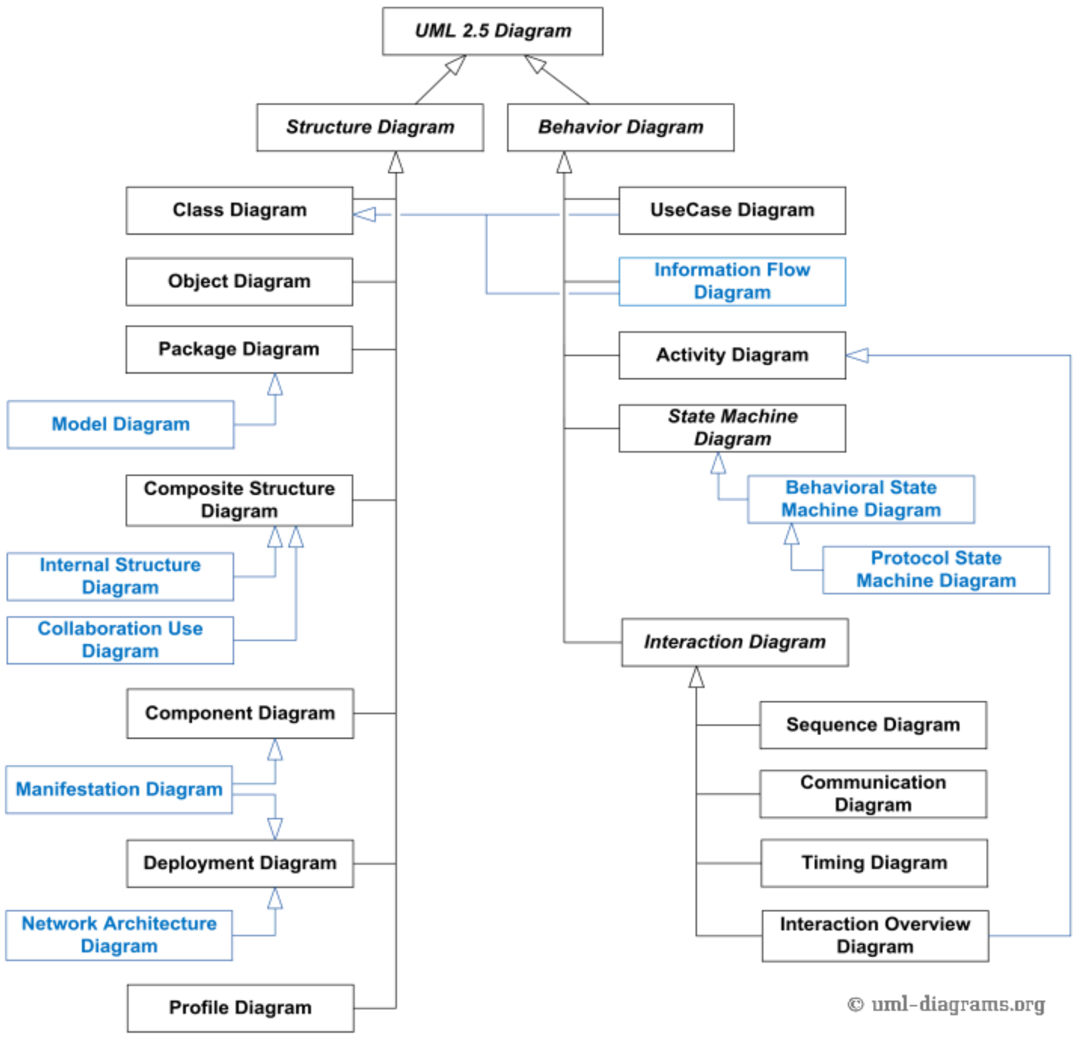


Figure 1

In order to make the overall view of a system as comprehensible as possible, it is important then to be sure each component diagram is clear and understandable, while still conveying all the necessary information. I will take a look at some of these UML diagrams and analyze their ability to accurately convey the information needed.

The Class diagram is one of the most commonly used diagrams in UML. A class diagram can be as simple as the name of a class along with a list of its attributes and operations. A simple example class is shown in Figure 2. This diagram easily communicates important information to the viewer, and it does not require the viewer to have much experience or familiarity with UML. Anyone with any experience in object oriented programming can understand this figure with little or no assistance.

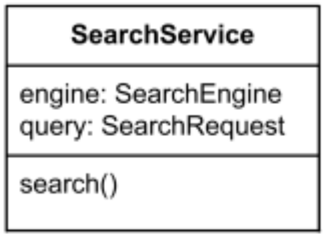


Figure 2

Of course, a class is rarely considered on its own. A class is usually presented as part of a larger diagram, like a domain model diagram. Figures 3 and 4 present two examples of domain model diagrams that include classes, entities, enumerations, and interfaces as well as all the relationships between those items.

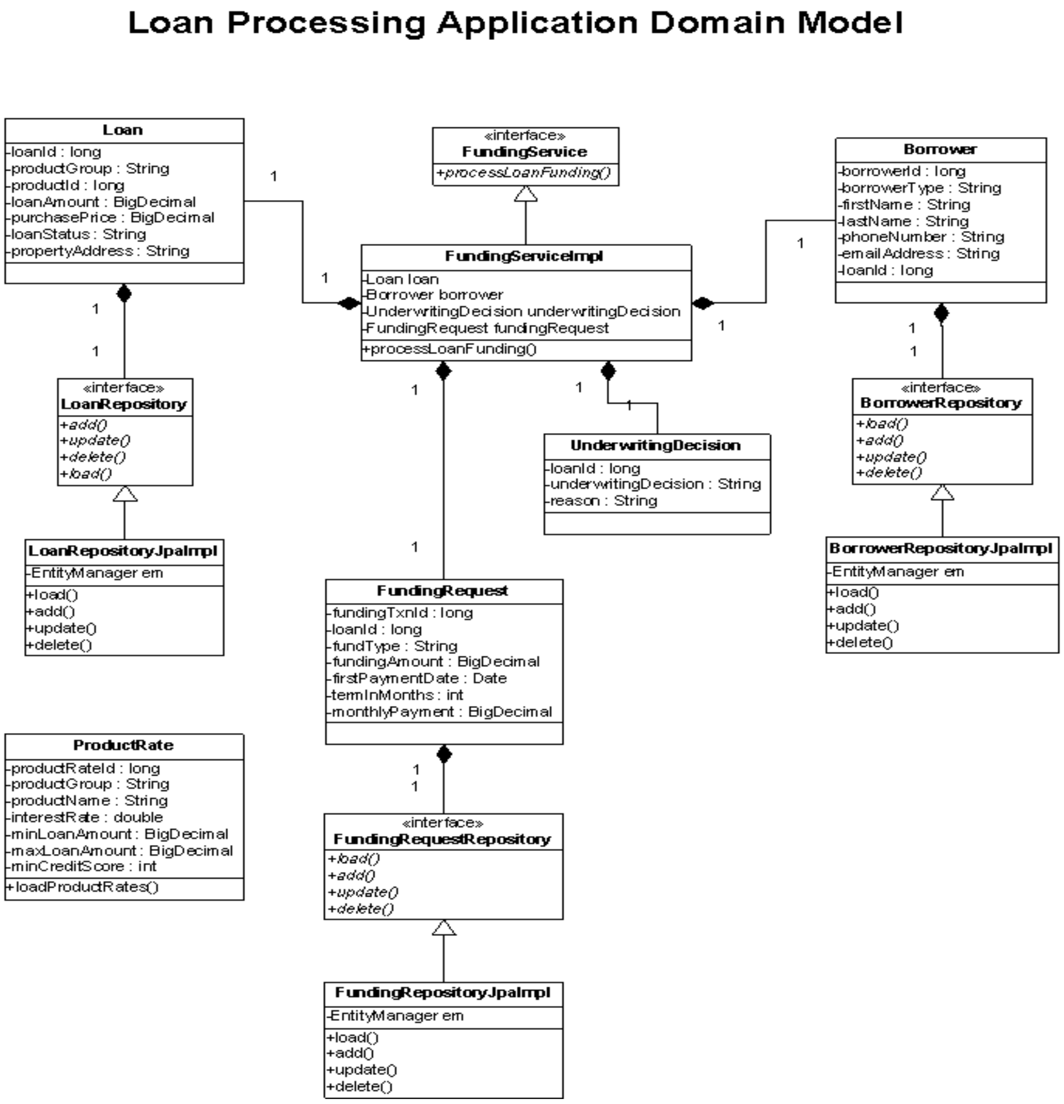


Figure 3

Taken from: http://www.infoq.com/resource/articles/ddd-in-practice/en/resources/LoanAppDomainModel_lg.gif

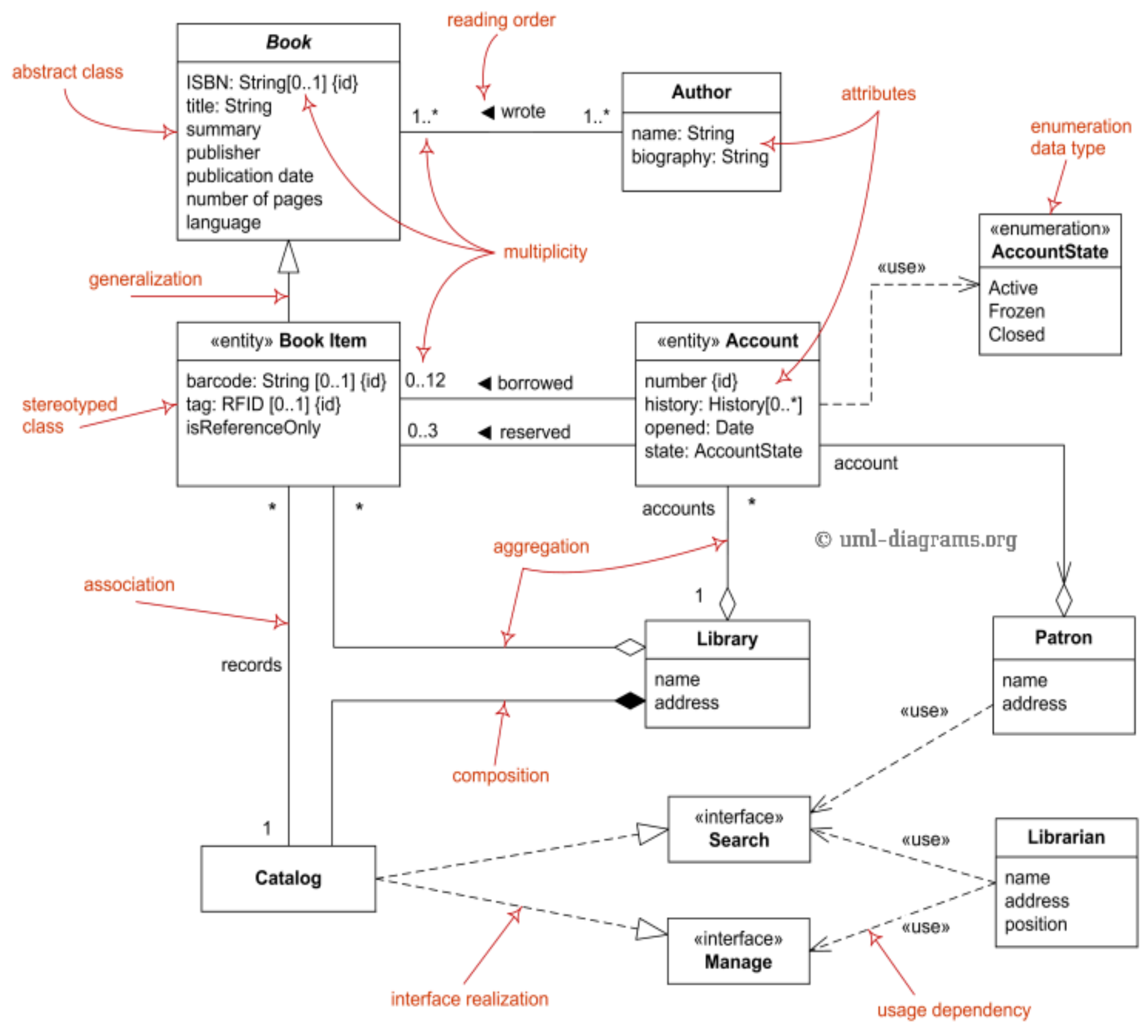
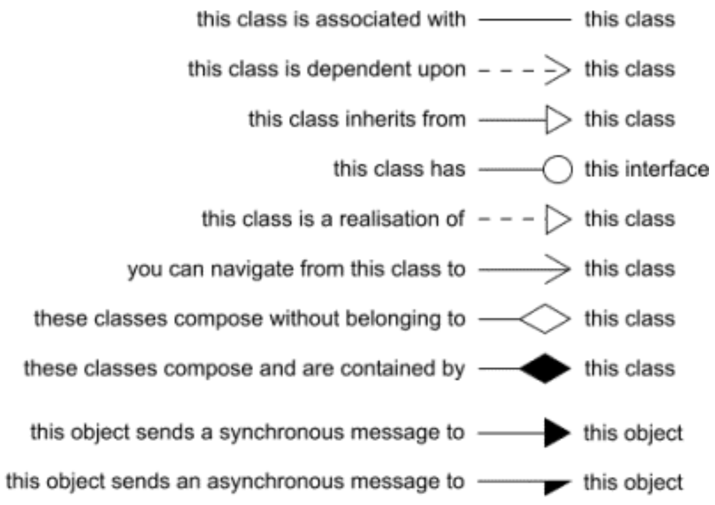


Figure 4

These diagrams are good examples of structural UML diagrams that, to an experienced user, can be very informative. However, to a student or a novice, these diagrams can be very hard to comprehend. Even an experienced developer will have to spend a little time deciphering the diagrams, and it is very possible that even they would have to look up what certain symbols mean to fully understand the diagram.

There are a number of ways in which these diagrams are very clear, however. One positive is that although the diagrams contain classes, interfaces, entities, and enumerations which all share the basic rectangular structure, anything that is not a class is labeled. In many cases it would be possible to figure out which object was what, but by labeling each object, it makes the overall diagram much easier to understand. Additionally, the use of arrows in a number of different situations also contributes to understandability.

Unfortunately, there are also a number of ways in which this basic type of structural diagram can be difficult to understand. For one, while the arrows can be useful, they can also be confusing. There are three separate types of arrows that denote different things. It would be very easy, especially for a novice user, to forget or confuse which arrow means what. In fact, there are even more possibilities for connections between objects than shown in these two diagrams- Figure 5 shows the range of possibilities to connect objects.



this class is associated with — this class
this class is dependent upon - - -> this class
this class inherits from —▷ this class
this class has —○ this interface
this class is a realisation of - - -▷ this class
you can navigate from this class to —> this class
these classes compose without belonging to —◇ this class
these classes compose and are contained by —◆ this class
this object sends a synchronous message to —▶ this object
this object sends an asynchronous message to — this object

Figure 5
Taken From: http://www.somanyword.com/wp-content/uploads/2014/06/uml-symbols.gif

Here we see that even the same arrow can have different meanings depending on the line it is connected to, dotted or continuous, and can also differ depending on whether the arrow is empty or filled in. The differences between all these are clearly far too complex for anyone but someone who regularly does modeling to remember. It is a well accepted fact in cognitive psychology that, as a general rule, a person can remember 7 +/- 2 pieces of information. Not only do the possible connections between classes come in at 10 separate possibilities, the rule from cognitive psychology also applies best to simple information, like numbers. Clearly, remembering all 10 possible interactions is not possible without training or extensive experience.

While the arrows do have some intuitive meaning when compared to a line without an arrow, each arrow does not have any meaning compared to any other arrow. In this case however, the arrows probably do as good a job as they can do. The arrow is needed to convey directionality, and having separate arrows is actually useful. However, this is the first instance we will see that simple labeling may have some value for comprehension- figure 4, with its red labels, is much more comprehensible than figure 3 because it labels the objects and their interactions.

There are a few other areas that could cause comprehension problems in these diagrams. The first are the diamonds used to connect objects indicating composition. The diamond itself has no inherent value, but the use of one filled in diamond and one unfilled diamond can actually be useful. Provided a person can remember that the diamond means composition, it is actually fairly intuitive that the filled in diamond would mean classes compose and contained by, while an empty diamond means classes that compose without belonging to a class. This is another case where labeling of some sort could be very useful for comprehension.

Another potential issue, albeit a significantly smaller one, is multiplicity. This is likely only an issue for beginner users, as multiplicity is seen in a variety of modeling methods. It is also fairly intuitive and easy to learn, so generally once a person has had it explained once, he or she will remember it without too much trouble. Of course, part of the reason it is easy to remember is because there are not other things in the diagrams that use numbers. The arrows can be especially difficult because there are so many different arrows used in different ways. The fact that the numbers only show up with multiplicity means that it is very unlikely that anyone will think they are related to anything else. Because of that fact, the multiplicity representation is probably fine the way it is.

One interesting UML model to look at is the sequence diagram. The sequence diagram is one of the rare models that manages to combine a structural view of a system with a behavioral view. Figure 6 shows a simple UML sequence diagram.

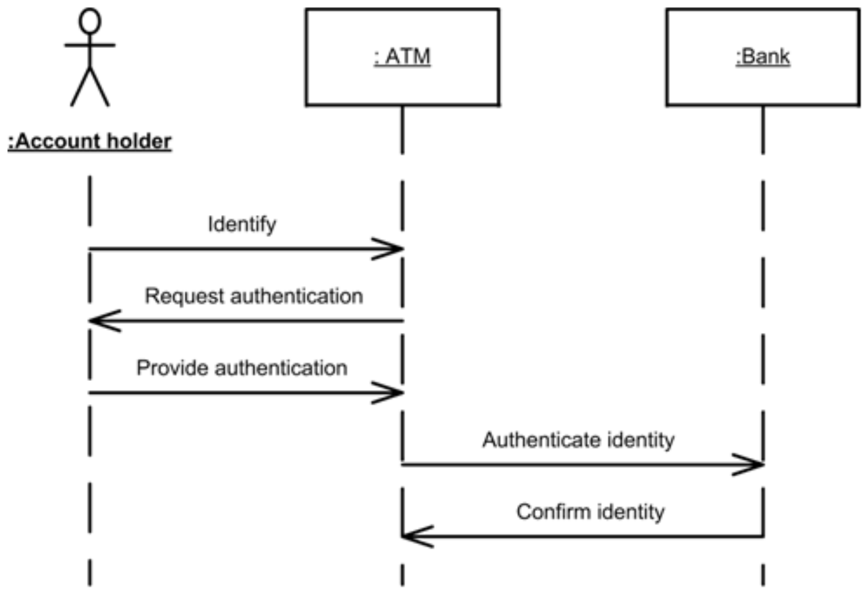


Figure 6
Taken From: http://www.jot.fm/issues/issue_2005_11/article5/images/figure2.gif

This diagram clearly shows the different objects and actors involved in the sequence, as well as the information and requests that flow between them. What is not clear to the untrained user is that going from the top to the bottom of the diagram represents the flow of time. Without knowing that information, it would be very easy to miss the main point of this diagram. On the plus side, that idea is quite easy to explain, and once explained, very easy to retain.

Another commonly used UML model is the use case diagram. Use case diagrams are behavioral diagrams that show actions that a system can perform in relation to outside actors, often users. These diagrams are fairly simple, but do a good job showing how use cases within an overall system relate to each other, and how outside actors can interact with the system. Figure 7 shows an example UML diagram for a store checkout system.

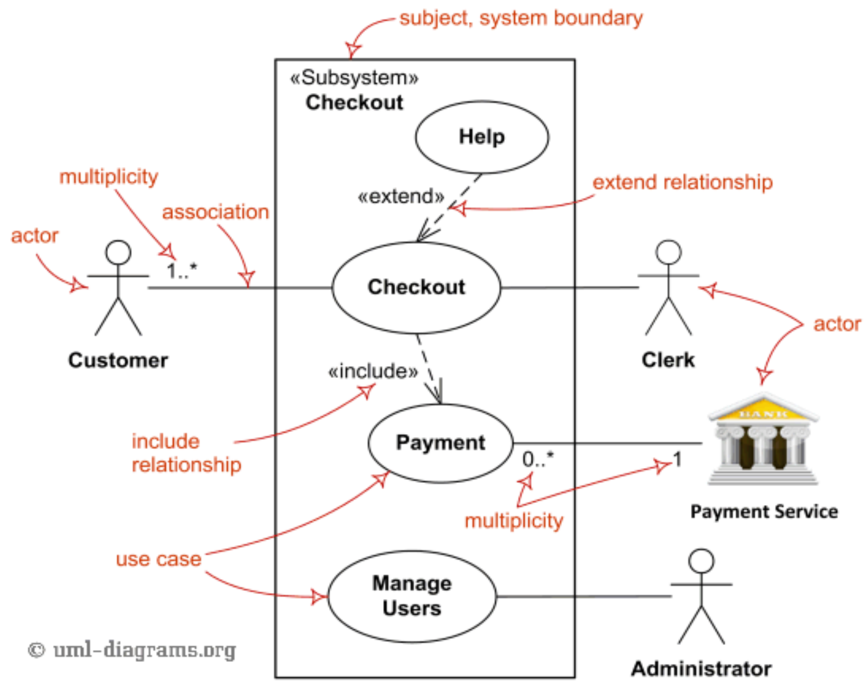


Figure 7

The fact that these diagrams are fairly simple means they are also very comprehensible with little or no training. The system is clearly delineated from the actors by a box, and the actors even have different symbols for whether they are people, services, or other systems. The relationship between use cases in the system and the actors is also very simple. One of the few points for potential misunderstanding is the multiplicity, but again, that is used in a large number of diagram types and is likely to be understood after only being explained once. Overall, the UML use case diagram is very understandable, with little area for misunderstanding. Of course, the price for that easy level of comprehension is that the diagram does not convey that much information. These diagrams are useful in large scale planning activities, or to explain systems to lay people, but has very little value in actually designing, building, or implementing systems.

UML also provides the possibility to dig deeper into an individual use case. The activity diagram shows the flow of the use case and demonstrates the role each individual actor, class, or device has in the system. Figure 8 shows an example of a UML activity diagram.

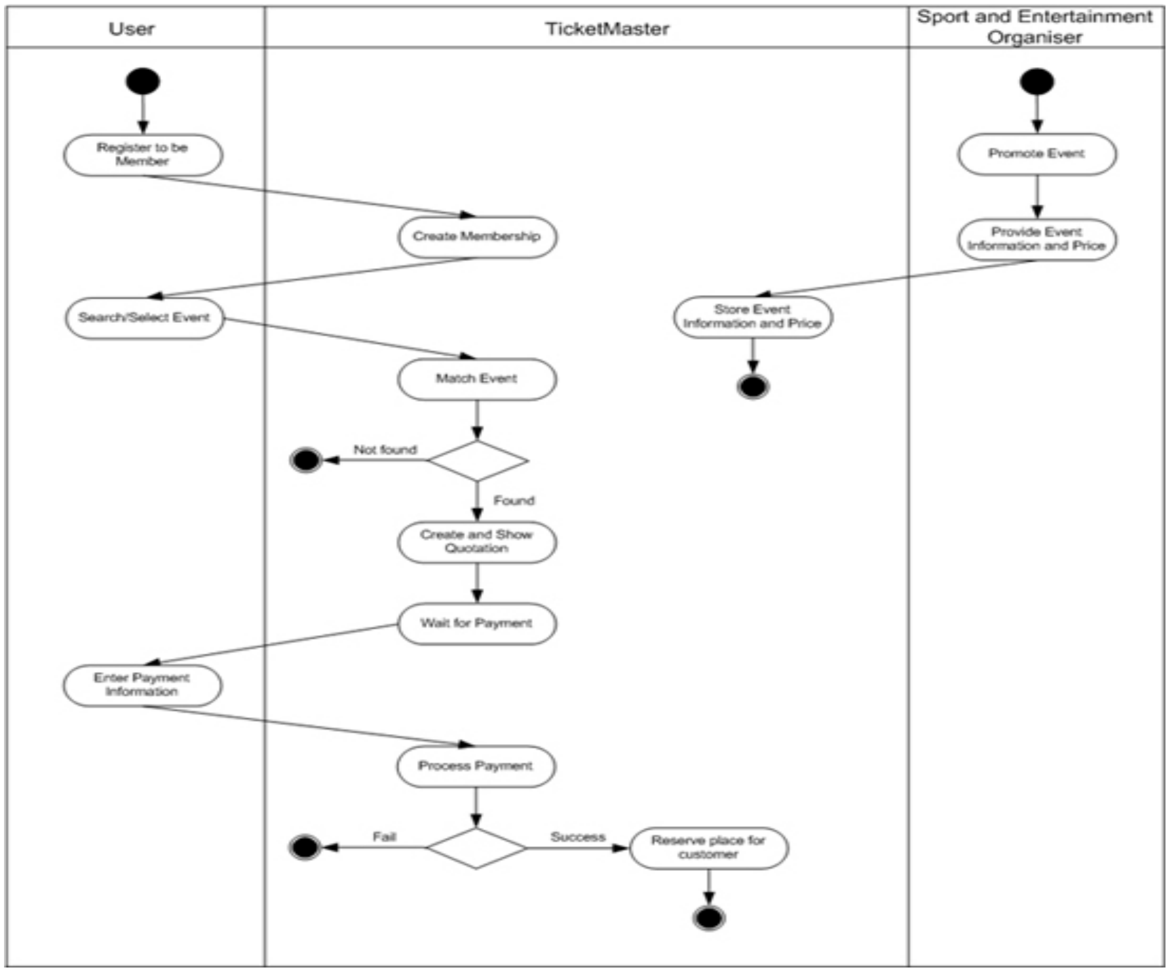

User
TicketMaster
Sport and Entertainment Organiser
Register to be Member
Create Membership
Search/Select Event
Match Event
Not found
Found
Create and Show Quotation
Wait for Payment
Enter Payment Information
Process Payment
Fail
Success
Reserve place for customer
Promote Event
Provide Event Information and Price
Store Event Information and Price

Figure 8

Taken From:http://ausweb.scu.edu.au/aw04/papers/refereed/tongrungrojana2/Figure5.jpg

These UML activity diagrams do a very good job showing the flow of activity in a system during a particular use case. Of course, they are limited that they only show the flow for one use case at a time, but they can be very useful to examine important use cases in depth. These activity diagrams also work well because they are very understandable. There is a clear flow of activity , and it is also clear which part of each system is involved in each step. There are only a few symbols in the diagram, and it is very clear what each one of them represents. The activity diagram is a good example of a clear, understandable diagram.

Another UML diagram that does a nice job conveying its information clearly is the information flow diagram. This diagram shows the flow of information of various sorts between different actors and classes within a system. Figure 9 shows a simple information flow diagram for a customer ordering something from a company.

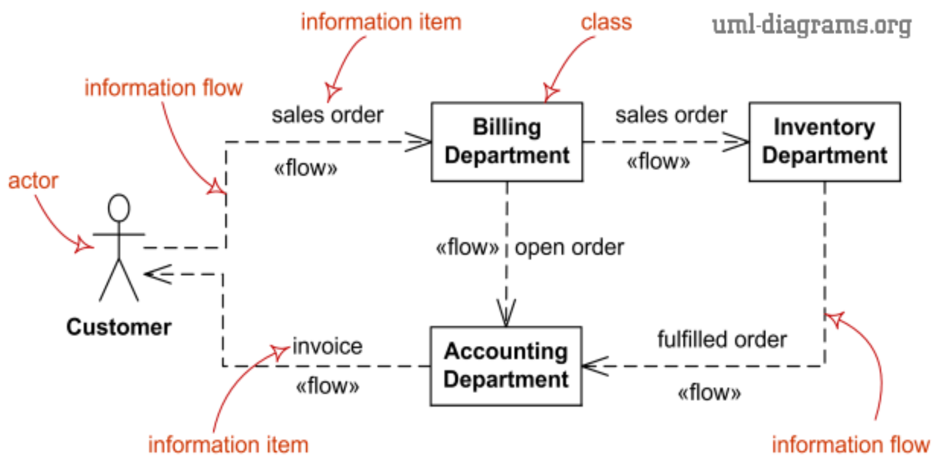


Figure 9

The information flow diagram is one of the best examples of a diagram that is probably comprehensible with no actual training or explanation needed. The lines and arrows only have one meaning, and everything is clearly labeled. Even a lay person would likely be able to figure out the main idea behind the information flow diagram. However, like the use case diagram, although the information flow diagram is very easy to understand, the amount of information conveyed is not that great. Although it is possible to get a sense of the information flow for a system, the simplicity of the diagram means that there is not much more information that can be shown in the diagram.

As shown in figure 1, there are a lot more UML diagrams than the ones I explored. However, one of the things that makes UML so appealing for so many people is that the various diagrams all use the same similar structures. The complexity level of most of the diagrams falls somewhere between the level of the domain model diagrams and the simple level of the information flow diagrams. Essentially, the observations made on the previous diagrams can apply to the other types of UML diagrams as well.

There are a number of other types of diagrams that do not use UML to model computer systems. Some of these diagrams have similar issues to those found in the UML diagrams, but they can also show improvements over some of the areas where UML presents comprehensibility problems. I will examine some of these diagrams and discuss their general level of understandability, and ways they could be improved.

One useful diagram is the state chart. The state chart shows the different states a system can be in, and the transitions that can take a system from one state to another. Figure 10 shows a UML state machine diagram, and Figure 11 shows another way of modeling states, the finite state machine.

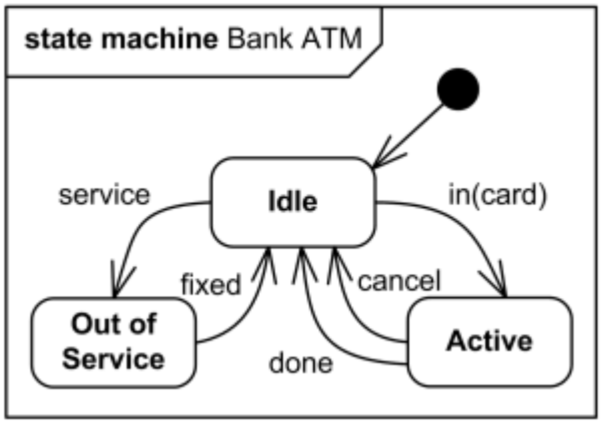


Figure 10

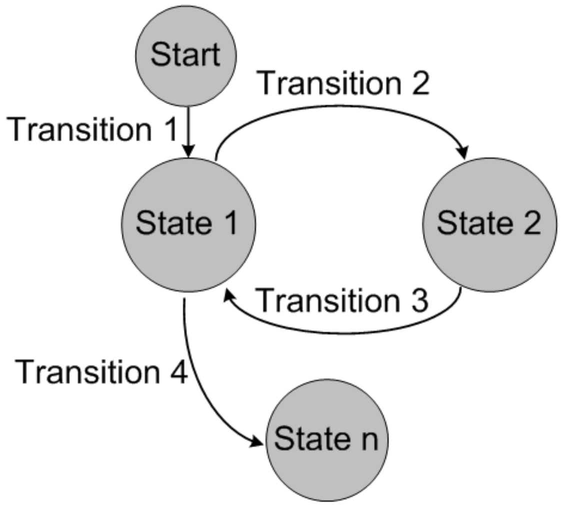


Figure 11

Taken from: http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/144986.gif

These diagrams are very similar, with one major difference that Figure 11 has start and end states, while Figure 10 has an idle state that it would spend most of its time in. These charts are quite simple, and their understandability is helped by the fact that each state is labeled, and the transitions between the states are also labeled. Like the information flow diagram, these state diagrams could likely be understood by someone with little or no training.

One way that these are not very clear, however, is showing any possibility of synchronicity. One assumption of these state diagrams is that a system can only be in one state at a time. However, this is not clearly shown in the diagram itself, and if a person unfamiliar with these diagrams were to take a look at them, that person might not understand that basic premise. That person might also think that multiple transitions could happen simultaneously, leading to multiple states. Although this would be possible for a person new to state diagrams to think, the facts that only one state is active at a time and that only one transition happens at a time are simple to understand and remember.

It is possible for state diagrams to represent concurrent behavior, but it does make the resulting diagrams significantly more complicated. Figure 12 shows a more complex statechart that allows for concurrency.

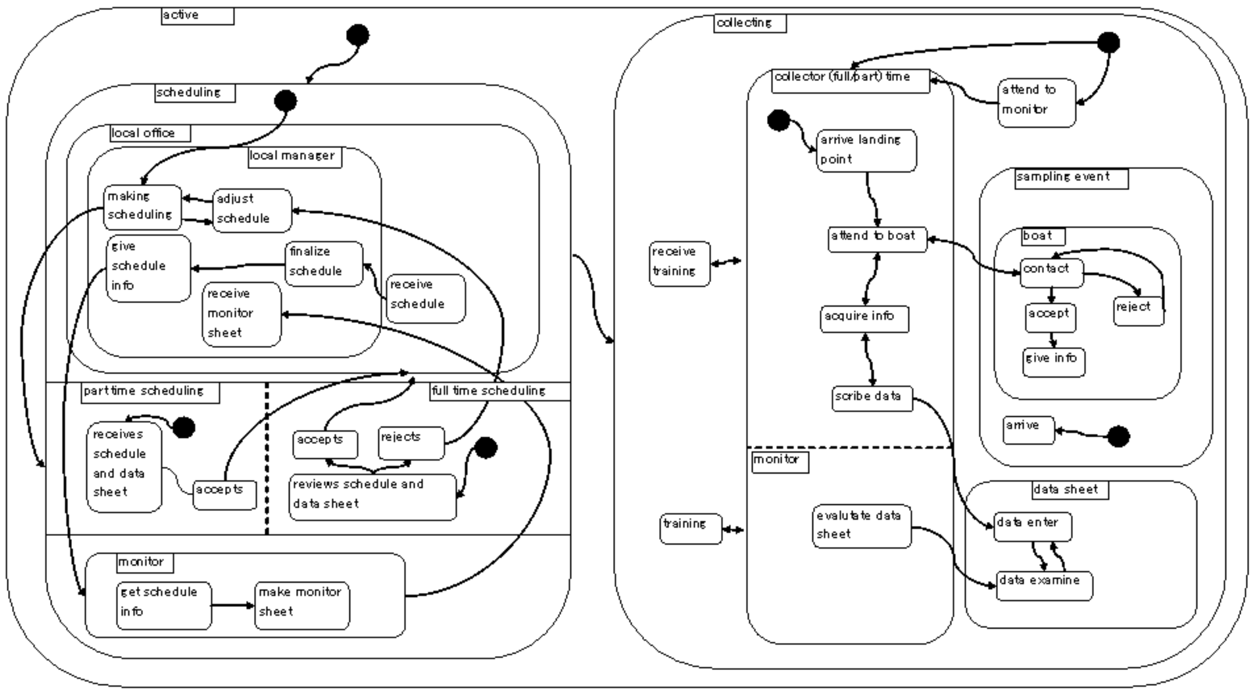


Figure 12
Taken From: https://www.isr.umd.edu/~austin/ense621.d/projects04.d/project-noriaki.html

With this diagram, it is easy to see the tradeoff of diagram complexity and comprehensibility. While using the same basic structures as the state machine diagram and the finite state machine, the statechart in figure 12 displays a hugely increased amount of information. However, it does so at the cost that even experienced users will have to spend some time examining the diagram to understand it, and it may even be incomprehensible to someone with little experience in software modeling.

One of the things that can be hard to understand about this type of statechart is the delineation between all the various states. We can see there are states within states and many ways to transition between the states. It is also not entirely clear that some of the states are states at all. An inexperienced user might look at five small states in a large box and make the mistaken assumption that the large containing box represents something else. Additionally, the delineation within some classes marked by the dotted line is not very intuitive. With a little investigation, it is possible to figure out that the dotted line means two exclusive states within the same state, but it is not very clear.

One downside with this type of statechart when compared to the more simple state machine diagram or finite state machine is in the transitions. Because of the simplicity of the state machine diagram and finite state machine, the transitions between the states are able to be labeled. The complexity if the statechart means there is not enough room in the diagram to label the transitions between states. This means there is less of an idea what is causing the transitions between states, whether it is initiated by an action or whether the transition will occur automatically.

The fact that all states are depicted the same way, with a rounded box labeled with a square box at the top does help the comprehensibility of the statechart. Of course, we would expect consistency like this, but because it is the same structure no matter what the size of the rounded box, it helps make the connection within the viewers mind that they are the same type of object.

Another type of behavioral model that can be used to model a system is the petri net. The petri net is largely used to model event driven systems. Figure 13 shows a simple petri net.

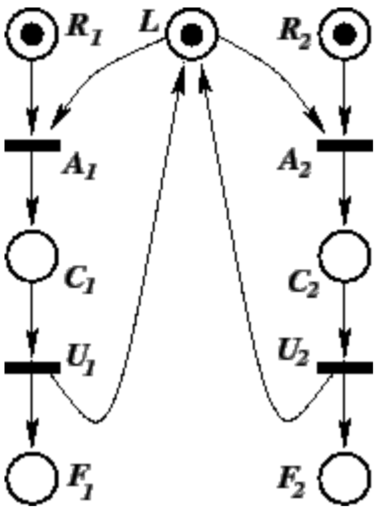


Figure 13

Taken From: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/wang/wang_html/img6.png

The petri net is another example of a model that will make little sense to someone without training. Although we can expect that any petri net will have labels or a key that indicates what the various places and transitions in the diagram all mean, it is likely that an inexperienced user will have no idea what the difference between the places and transitions are, and will not easily be able to follow the progression of the system. In particular, the sequence of transition firing and marker movement is not something that is intuitive, and would require training to understand. There are also some things that are not immediately obvious even after training, like the fact that when multiple transitions are enabled, only one will fire at a time.

On the plus side, once the basics of transition enabling, firing, and marker movement are explained, petri nets are fairly simple to grasp. There are really only two objects in a petri net, places and transitions, and there is only one type of connection between objects. The petri net allows for things like repetition and mutual exclusion, and is able to model those using the same simple symbols.

Petri nets can be expanded to have additional functionality by moving to the event driven petri net (EDPN) model. This model introduces another object to the model, representing input and output events. Using the symbol for events, EDPN are able to model things that the regular petri net cannot, like event quiescence. Notably, despite the addition of one object to the model, most of the functionality and diagramming of EDPN is the same as the petri net, meaning once the concept of the additional object is explained, a user experienced with petri nets will be able to understand EDPN with little or no difficulty.

General Complexity Examination

The first interesting thing of note after looking at all these various models is the tradeoff of information presented and comprehensibility. Some of the simplest models present things in a way that would probably be understandable to even to someone not trained in software engineering at all, but those models generally provide only a small amount of information. Conversely, some of the most complex models provide a wealth of data, but require even an experienced and well trained user to spend time deciphering them.

One crucial note is that we do not see any instances of complex, hard to understand models that do not convey a lot of information. Seeing a model like that would likely indicate nothing more than a poorly designed model. Thankfully, none of the commonly used models fall into that category, so we can safely assume those models are well designed.

One way to see what differentiates good design from bad is to look at a model that is not as well designed as most of the models that are commonly used today. One example of this is the Yourdon/Coad Object Oriented Analysis model. This model was developed in the early 1990’s, and the different types of objects are shown in Figure 14.

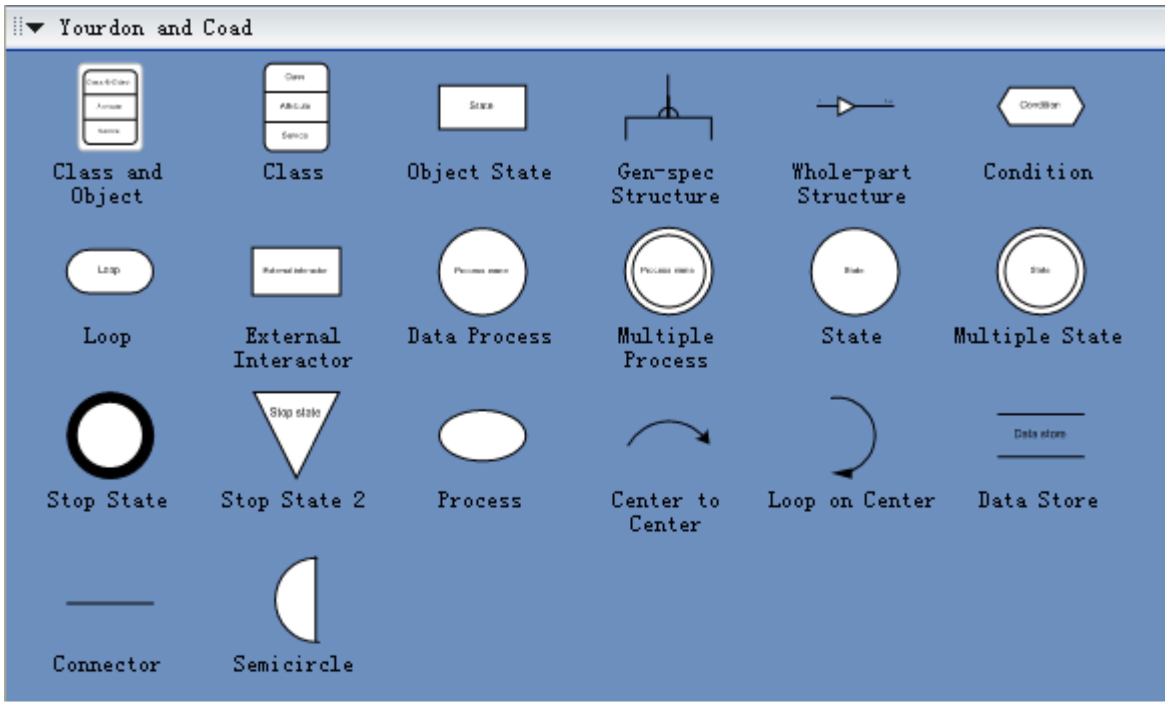


Figure 14

Taken From: https://www.edrawsoft.com/Yourdon-and-Coad.php

There are a number of issues with this model apparent from the start. First, many of the symbols are very similar to each other. For example, the class and class/object symbols are nearly identical. Even worse, things like data process and state are actually identical, with only context clues providing information as to which is which. This model is an excellent example of a model that may work well in use, but would be difficult to learn and even more difficult to use or analyze correctly as an irregular user.

In addition to examining models on an individual level, it is also interesting to look at some ideas of complexity that can apply to software modeling in general. One idea to use is borrowed from linguistics, called phonemic orthography. Phonemic orthography is an examination of how well the written symbols of a language correspond to the the spoken sounds of that language. A language with high phonemic orthography will have very few ways to write a given sound. Languages like Italian and Finnish have high phonemic orthography, and in fact, many languages in this category do not even have a word meaning 'to spell', since writing a word is so closely based on the pronunciation of that word.

On the other hand, languages with a low degree of phonemic orthography have many ways to write the same sound. English is a great example of a language with a low degree of phonemic orthography, which will of course make sense to anyone who grew up in an English speaking school and spent many hours on spelling lessons.

The general idea of phonemic orthography can also be applied to software modeling. A language with a low degree of phonemic orthography will be more complex (at least in this one way) than a language with a high degree of phonemic orthography. If we view a system as analogous to a spoken language, and the model of that system as analogous to the written version of that language, we can also discover complexity among those software models. A model with less complexity, like a language with a high phonemic orthography, will only be able to express an aspect of a system in one or very few ways. A model with a higher degree of complexity will look like a language with a low degree of phonemic orthography, and will be able to express an aspect of the software it is modeling in a number of different ways.

The reason this affects the complexity of the model is fairly intuitive. If a model can express part of a system in a number of ways, it is harder for the user to remember all of the different ways of expression, as well as apply them. Clearly, it is important to insure that models keep complexity low by not allowing multiple methods of modeling the same thing.

One example of a low degree of complexity in this area is presented back in Figure 6. This figure is the sequence diagram, and everything in it, including lines and arrows, only have one meaning. That can be contrasted with Figure 5, which shows the possible connections between objects in a structural diagram. This diagram shows two different kinds of lines (solid or dotted), three different kinds of arrows, two types of diamonds, and even a half arrow and a circle. While combining all the elements of each connection is unique, there is a problem that each aspect of the connection has little or no intrinsic value. A user will not know what a particular arrow means without also knowing what kind of line is connected to that arrow. Clearly, this adds a high degree of complexity to the model.

Another important implication of applying this idea to software modeling is that the complexity of the overall modeling world can also be increased when the same symbols are used. Multiple systems using the same symbols to mean different things will significantly increase the complexity of every system using those symbols, even if the symbol use is consistent within each system individually. Symbols with different uses across different models mean it could be harder to learn multiple systems and that users will be more likely to misapply symbols when working with unfamiliar models.

Another idea that can come into play when looking at model complexity is that of model hierarchy. Models can use hierarchy to display different levels of a system. This allows the entire system to be split up into multiple models, meaning a complex system does not have to clutter up one large model. A hierarchy would mean a top level view of the system can be presented in one model, and more detail can be presented in other models that do not have to worry about displaying the entire system. Of course, to keep things clear, there must be a clear and understandable method of relating the various models to each other. Without a good system for that, it could be very difficult for someone not familiar with the system to make sense of the multiple models. Some models like activity diagrams and information flow diagrams can already make good use of hierarchy, and it may be useful to use it for other models as well.

When discussing the general idea of comprehensibility of these models, it is important to think about the contexts in which they apply. The first context is that of a person learning the model, and the second is of a person putting the model to use. The complexity and general understandability of the models will contribute to success or failure in both of these areas.

One of the primary goals of software modeling is in fact to help the user with program comprehension. Obviously, if a user does not understand or misunderstands the model of the system, their ability to understand how the system functions is significantly impaired. There are three primary hypothesis for the way programmers approach program comprehension, top-down, bottom-up, or a combination of the two. Model comprehension can significantly impact these methods of program comprehension. For a programmer who approaches a program from a bottom-up perspective, a software model may not actually be that useful. A bottom-up approach is focused more on understanding the program at the code level, which a software model does not help present.

However, for a programmer who approaches program comprehension from fully or partially top-down, a good software model can vastly assist with that program comprehension. The top-down theory of program comprehension says a programmer will form a hypothesis of program functionality based on information besides the program code. In this case, a clear and understandable model should greatly improve the hypothesis that the user comes up with, and they can then use things like the code to support that hypothesis. However, if a user does not understand or misunderstands a model, it is likely that the hypothesis they come up with will be flawed, which means it will take the user additional time to examine code and discover the flaw in the hypothesis, as well as develop a new hypothesis.

One important finding of cognitive psychology and the psychology of learning is the fact that a person who truly understands what they are learning is more likely to retain the knowledge long term and be able to successfully apply it in the future. Rote memorization works fine for short term recall, which may help a college student pass an exam, but the best way to make sure the student remembers material is to make sure the student understands it. This means that the more complex a model, the more important it is that a person learning it actually grasps the concepts behind it. Of course, this is difficult since the complex a model, the harder it is to actually understand. Unfortunately, there is little that can be done to insure people are gaining true understanding when learning these models. The models are as simple as they can be, and making them simpler would dilute the amount of information they are able to convey. So, aside from checking over school curriculum and helping teachers teach the content, we have to look to people actually implementing the models to see what can be done to help their comprehensibility.

When thinking about people putting the models to use, another important concept of cognitive psychology comes up. Namely, the more someone is exposed to or uses an idea, the better they retain that idea. This means that a person who models or at least looks at models multiple times a year will have a better ability to comprehend what is going on in those models than someone who is exposed to similar models only once a year or less. This fact means that a person who is a regular user or viewer of the types of models discussed here is likely going to need little or no help understanding these models when presented with them, but a person who does not get as much exposure is less likely to comprehend what they are seeing. It makes more sense then to focus on the irregular user and see what can be done to assist that person with understanding the software models.

One important concept of memory and recall is the fact that memory prompts or cues can aid significantly in recall. Someone trying to remember a difficult concept can be helped with even small prompts, like a keyword related to that concept. The psychological concept that undermines this idea is called the spreading-activation model. The spreading-activation model of memory states that memories and knowledge in the brain are represented in nodes that are connected to related nodes with associations of varying strengths. So, a specific prompt will activate one memory node, which will in turn activate the memory nodes connected to it. That means any prompt about a software model will greatly help someone with recalling all the information about that model.

These prompts could be vital when someone is encountering a diagram they are not very familiar with or have not seen for a long time. The way to prompt them would be pretty simple, and would not take much effort. I see two potential ways to do it. First would be a text reminder integrated into the diagram. This could be something like Figure 4, where the interactions and many of the objects in the diagram are labeled with red reminder text. This goes a long way to making a complicated diagram more understandable. The downside to this approach is that the diagram will become more cluttered, which could be annoying to more experienced modelers who do not need the reminders.

The other way to provide prompts would be to include a legend with a complicated model. This legend could be pretty simple, perhaps resembling Figure 5. In fact, it would likely function even if it were simpler than Figure 5. For example, you could replace "these classes compose without belonging to" paired with the symbol with "compose without belonging" paired with the symbol. Potentially, even just "compose" paired with the symbol might be enough. The key is to present that prompt to aid the persons recall, and even just a keyword prompt is probably enough to give a significant boost to their recall and comprehension.

The advantage of presenting this information in a legend is twofold. First, it would be possible to present the legend with only the symbols used in the actual model- there would be no need to present a full legend with every possible symbol. Modeling software could easily automatically construct the legend based on the model built in the software. The second advantage is that the legend would be available for those who need it, but anyone who would not need to consult it would not have it in the way of the model itself.

Conclusion

David Harel outlined his four keys to a successful system model: clarity, precision, visualization, and executability. Many of the current software models in use today fail on examination in some of these areas. Some of this is perhaps unavoidable, as modeling a complex system necessitates a somewhat complex model, but there are a number of ways that models seem to have introduced complexity unnecessarily. Harel was spot on with his understanding of what makes a good model, and it would be good for those who create or regularly use some of these models to emulate his example and attempt to improve them.

In summation, models of computer software range from very simple to very complex. The more complex models could potentially be difficult to understand for inexperienced users, and are hard to learn initially. The best way to make sure people understand complex models they encounter is to provide them with a cue to remind them of what the different symbols mean, and the easiest way to do that is to present a legend with a complex model.

References

Bereiter, C., & Scardamalia, M. (1996). Rethinking learning. In D.R. Olson, & N. Torrance (Eds.), *The Handbook of education and human development: New models of learning, teaching and schooling* (pp 485-513). Cambridge, MA:Basil Blackwell.

Booch, G., Rumbaugh, J., and Jacobson. I. (2005). *The Unified Modeling Language User Guide*. Second Edition, Addison Wesley.

Coad, P. & Yourdon, E. (1991) *Object-Oriented Analysis*. Prentice-Hall.

Collins, A. & Loftus, E. (1975). A spreading-activation theory of semantic processing. *Psychological Review,* 82(6), 407-428.

Corritore, C. & Wiedenbeck, S. (2001) An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54(1), 1-23.

Derwing, Bruce; Priestly, Tom; Rochet, Bernard. (1987). The description of spelling-to-sound relationships in English, French and Russian: Progress, problems and prospects. In P. Luelsdorff (Ed.), *Orthography and phonology*. Amsterdam: John Benjamins.

Harel, D. (2007) Statecharts in the Making: a Personal Account. *HOPL III Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 5-1 - 5-43.

Jones, G. V. (1976). A fragmentation hypothesis of memory: Cued recall of pictures and of sequential position. *Journal of Experimental Psychology: General, 105*(3), 277-293.

Jorgensen, P. (2009). *Modeling Software Behavior: A Craftsman's Approach*. Boca Raton, FL: Auerbach Publishing.

Neath, I. & Surprenant, A. (2003). *Human Memory*. 2nd ed. Australia ; Belmont, CA : Thomson/Wadsworth.

Miller, G. A. (1956). "The magical number seven, plus or minus two: Some limits on our capacity for processing information". *Psychological Review,* 63(2), 81–97.

Pennington, N., Lee, A. V, and Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, 10, 171n226.

Pilone, D. (2006). *UML 2.0 Pocket Reference*. Retrieved from http://www.eblib.com

Port, R. (2007). The Graphical Basis of Phones and Phonemes. In Munro, M. & Ocke-Schwen, B. (Ed.), *Second-language speech learning: The role of language experience in speech perception and production.* Amsterdam: John Benjamins.

Raaijmakers, J. (2003) Spacing and repetition effects in human memory:application of the SAM model. *Cognitive Science,* 27, 431–452 .

Siau, K. (1999) Information Modeling and Engineering A Psychological Perspective, *Journal of Database Management,* 10(4), 44-50.

Siau, K., and Cao, Q. (2001). Unified Modeling Language - A Complexity Analysis. *Journal of Database Management*, 12(1), 26-34.

Siau, K., and Lee, P. (2002) Difficulties in Learning UML: A Concept Mapping Analysis, Seventh CAiSE/IFIP8 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'02), Toronto, Canada, 102-108.

Siau, K., and Loo, P. (2006). Identifying Difficulties in Learning UML. *Information Systems Management,* 23(3), Summer, 43-51.

Tyson, K., and Frank, W. (2002). Be Clear, Clean, Concise. *Communications of the ACM*, 45(11), 79-81.

Whittle, J. (2000) Formal Approaches to Systems Analysis Using UML: An Overview, *Journal of Database Management*, 11(4), 4-13

Unless otherwise noted, images taken from: http://www.uml-diagrams.org/