Demystifying UML

Stephen J Mellor

What is UML—exactly? How is it being used? By whom? Are there many ways to use it, or is there one true way? Does it apply to embedded and real-time systems? How? Really? What are the popular ways to use UML? How effective has it been? Why are there so many tools? Why do they seem to be so different—even though UML is a standard? What should I look for in a tool? Will using UML change my development process? Make it faster? Or not? Build better quality systems? Or is it all hype?

If these, and myriad more, questions bombard you when you think about UML, this article is for you. It answers these questions and more by laying out the provenance of the language; its various usage styles, subsets and extensions, including real-time profiles; where it is being used and with what degree of success. A brief self-assessment is included to help you determine whether your team is ready for UML, and if so, how.

What is UML—exactly?

The Unified Modeling Language is a graphical (and textual) way to describe systems standardized by the Object Management Group. The OMG is a not-for-profit consortium of users and vendors dedicated to interoperability of object-oriented systems.

For a user (i.e. a developer), the UML presents itself as a notation for common object-oriented concepts. For example, before the UML was standardized some folk represented a class as a box, and others a cloud, each with variations for attributes and other properties of a class. The UML standardizes on a three-box compartment with the name of the class in the first compartment, attributes in the second, and so on. This gives us “interoperability” in the sense that we can now have the same understanding of those icons on a screen or whiteboard.

The definition of UML is captured in a specification that is itself written in UML. Just as you might have a class Circuit with an attribute bandWidth in a model of telephone system, the UML specification has classes Class and Attribute to capture the concepts in the modeling language. Tools capture user models as instances of those classes (Class, Attribute, etc.), which encourages interoperability of those models by tools.

Where did the UML come from?

The UML grew from a surfeit of object-oriented methods in the early nineties. These methods defined how to analyze and design systems, but they all used different notations, which caused considerable confusion in the marketplace. To unify these methods, the “Unified Method” was born as a project in the OMG.

Unfortunately, as the joke I first heard from Martin Fowler has it, the difference between a terrorist and a methodologist is that you can negotiate with the terrorist, so the scope was cut to define a notation, not a method.

This is important: The UML defines a *notation*, not a *method* for analyzing and designing systems. UML offers *no* advice on how to go about building a system, only on how to describe it.

The designers of the UML took this decision seriously. Each diagram is well defined, but there is no one true way in which the diagrams must fit together. That decision is up to you.
So what diagrams make up the UML then?

There are thirteen diagram types, listed below with their usage.

Diagram Type	**Usage**
Use case	Shows requirements in terms of interactions between a system and its users.
Class	Shows classes, attributes, associations and generalization
Object	Shows selected instances of classes and values for attributes at run time
Sequence	Shows synchronous and asynchronous interactions between objects
State Machine	Shows behavior over time in response to events
Component	Shows large-scale components and their interfaces
Composite Structure	Shows internal structure, ports, collaborations, structured classes
Package	Shows groupings of elements into packages and dependencies between them
Communication	Shows communications between objects (used to be called "collaborations")
Activity	Shows parallel and sequential behaviors connected by data and control flows
Interaction	Shows an overview of activities and interactions
Timing	Shows timing in a variation of sequence/interaction diagrams
Deployment	Shows deployment onto nodes in a specific implementation

What do all these diagrams do?

That could take a whole book! For a breezy introduction, read *UML Distilled*, by Martin Fowler. If you're partial to cubical books, take a look at *Doing Hard Time: Developing Real-Time systems with UML*, by Bruce Powel Douglass. If that's too much, try *UML in a Nutshell*, by Pittman and X. For another view, read *Executable UML: A Foundation for Model-Driven Architecture* by Stephen Mellor and Marc Balcer.

Unless you are a tool builder or a masochist, I *do not* recommend reading the UML specification from the OMG, because it is (supposedly) written for clarity of specification, not ease of understanding.

Which of these diagrams are especially useful for real-time/embedded designers?

They all have their uses. The most commonly used diagrams are the class diagram, state machine, use case, and the sequence diagram. The timing diagram is new in UML 2.0, so we don't yet know how useful it will turn out to be.

Certainly, the state machine diagram is especially helpful in event-driven systems.

Why is UML so important for designing embedded systems?

Because it increases your productivity by allowing you to work at a higher level of abstraction. Studies going back to the days of the first programming languages show you can write the same number of lines of code per day, irrespective of the language, so a higher level language like UML will necessarily be more productive. Moreover, UML allows you to visualize concurrent behavior, which is difficult in a traditional programming language, which is expressed linearly.

Which of the different embedded markets (automotive, defense/aerospace, industrial, consumer electronics, communications etc.) are today the main users of UML? Are UML tools mainly used by big companies?

All these markets use UML, and the usage of UML is increasing daily. Companies both big and small use UML.

I have direct knowledge of UML being used in systems as varied as telecommunications switches (several million lines of C++), all the way down to pacemakers and drug-delivery devices. Consumer electronics, avionics, automotive, you name it.

Who is using it in real-time/embedded?

Venture Data Corporation, in its *Embedded Software Developers' Requirements and Preferences Report 2005*, says that 14% of projects in 2004 were using UML, and this will grow to 25% by 2007. Of course, it depends on what you mean by "uses."

What do mean by that?

There are many ways in which the UML can be said to be used. Most people use the UML informally. That is, they *sketch* out a few diagrams on the proverbial beer mat and discuss their abstractions with their peers (*sic*). From there, coding proceeds directly. Martin Fowler's book, *UML Distilled*, takes this point of view.

Others use UML as a *blueprint* that specifies software structure. There is an intended near one-to-one correspondence between the UML diagrams and the code. A tool can generate code frames from these models, and the developer fills in the rest using the models as a guide. Note that if the developer finds a better solution while coding, the model will no longer reflect the code. Users of this approach must have a detailed understanding of their code structure and the corresponding elements in UML. They will find *UML in a Nutshell* useful in this (though the authors do not explicitly promote this approach.)

Others go further. The developer can add code directly to the model making the model *executable*. Note that if the generated output is changed by hand, the model no longer reflects the code. This may then lead to a desire to "round trip" and generate a model from the code, which only makes sense so long as the model bears a one-to-one relationship to the code. This point of view is promoted by Bruce Powel Douglass.

The fourth use is *translatable* models that have action language as a part of the model. The action language is executable, but it can also be translated into *any* implementation. A given model may be translated into an implementation that has many tasks or one, many processors or one, or even into different hardware/software partitions. In contrast to the second and third uses, there is no necessary correspondence between the structure of the model and the structure of the implementation except that the behavior defined by the model must be preserved. Mellor and Balcer hold this viewpoint.

These four ways[1] of using UML (sketch, blueprint, executable, and translatable) enable different kinds of reuse.

What are the different kinds of reuse?

Sketches are useful mainly to help visualize a solution, and for communication between people, whether the sketches are drawn on a beer mat or using a drawing tool. This has the advantage that the sketches have low maintenance costs (because you don't bother), but they cannot be reused except from the base of knowledge you and your colleagues have accumulated in your heads.

Blueprints are useful to document a design, especially a large one that will be coded as a separate step. Depending on your process, you may code from the blueprint, or write code as you build the model incrementally. (The latter is generally to be preferred). However, as knowledge is gained from coding, the design may change. Going back to change the model is generally seen as "extra work" that does not contribute to getting a running system, so the model and the code may get out of sync.

Attempts to reuse the models often founder on that inaccuracy, leading to calls to "reverse engineer" the code. Leaving aside the fact that it is problematic to reverse engineer an artifact that was not engineered in the first place, the real engineering is smeared throughout the model. The fact that you decided to use three tasks, say, is evident, but your choice of communication mechanism is repeated, often slightly differently in each place, where ever communication is required. Lack of uniformity of design is one reason for the failure of reverse engineering.

These concerns can be addressed by making the blueprint executable, adding code to the graphical UML elements as required, and generating everything—always—from the model, a kind of high-level visual programming. These models *can* then be reused if *both* the application and the implementation structures will be largely the same in the new system.

In contrast, the distinguishing feature of translatable models is that they separate application from implementation so they can each be separately reused.

How can you reuse an implementation without reusing the application?

Translatable[2] UML models have two parts: the model of the application, and a set of rules for implementation.

We can capture what your system does and what data does it need to support that behavior as a UML model that of the application, with the proviso that we make no implementation decisions. Now let's decide to use, say, C structs, a state machine dispatcher, and a function for each set of actions triggered at once. The two sentences above represent two separate ideas. The former captures the behavior of the application; the latter captures the overall structure of the implementation.

You can reuse these latter (implementation) decisions in a completely different application, building a wholly different system that makes use of say, C structs, a state machine dispatcher, and a function for

[1] Martin Fowler has independently arrived at this characterization, though he does not call out "translatable" as different from executable.

[2] "Translatable" must always be read as "Executable *and* Translatable". We use this shorthand to distinguish translatable models from those that merely execute.

each set of actions triggered at once. After all, there is no mention of the application in those implementation decisions.

You could also make a different set of design decisions, say to use C++ classes, a switch statement for each state machine, and a sequence of linear statements for what gets done on each transition, and apply those to any application.

In other words, the application can be reused and redeployed separately from the set of design decisions, which can be reapplied to different applications. (See Figure 1.)

How effective has UML been? Does it decrease project time? Build better quality systems?

Frankly, that's hard to know in a scientific way—few people have the resources to build a system once, let alone twice, once using UML and once without. And if they do, it's often with small projects using students as guinea pigs. That's unconvincing.

However, it stands to reason that if we can visualize a design and talk about it to improve it, it's going to be better than if we just hack out the design. In any case, using UML this way is an option, and you would not diagram out everything, only those pieces that can benefit from sketching. Using UML will then decrease time to market because the earlier you find an error, the cheaper it is to fix.

It is *not* clear that using UML as a blueprint for software is effective. More accurately, it is not clear that building a blueprint, carefully documenting it, and then coding from that blueprint is effective. In fact, there is a whole movement, the Agile Alliance (www.aanpo.org), that argues against models and documents as being superfluous and costly. The most visible faction of this movement is XP, or eXtreme Programming, led by Kent Beck who published the eponymous book in 2000. Chief among the arguments made by this movement is that you cannot know a model is correct because you can't test it, nor does it execute.

This argument does not apply to executable or translatable models though. Because these models can be interpreted, they can be checked for correctness with the customer, and analyzed in a variety of ways for synchronization problems. This type of verification is especially important in systems that have to be known to be correct, such as medical instruments, airplane control, and the like. Executable or translatable models give you both the advantages of UML and the advantages of agility.

I hear a lot about UML 2. What's that?

UML 2 is the next major revision of UML. UML 1.0 was the first version of UML. UML 1.1 was the first updated version based on reviews of the initial specification. And so on. UML 1.5, the last UML 1.x added an action model to the specification.

UML 2.0 added features, such as the composite structure diagram and timing diagram, to model architectures better, and several features from SDL, the Specification and Description Language used for the specification of real-time applications with many concurrent event-driven activities, which is used in the telecommunications industry.

Moreover, UML 2.0 refactored and reorganized the underlying metamodel of UML 1.x with the goal of making it cleaner and smaller, but with minimum violence to the notation of UML 1.x.

Meta *what*?

A metamodel is a model in which the instances are types from another model. The UML metamodel has classes Class and Attribute. Instances of the class Class might be Circuit or Switch in a model of telephone system, and bandWidth an instance of the classes Attribute.

"Meta-", means "beyond" or "after", and it is a relative term. The class Class is "meta" to the class Switch, and the class Attribute is meta to the attribute bandWidth.

Why is the UML 2 not well suited to the design of embedded/real-time applications?

UML 2.0 *is* well suited for the design of embedded/real-time applications. But it's also well-suited to business process re-engineering, workflow analysis, and IT systems. This goes back to the idea that though each diagram is well defined, there is no one true way in which the diagrams must fit together. That decision is up to you.

So how do real-time developers fit the diagrams together?

There are many possible ways. Some people:

- build whatever seems to be illuminating just this minute
- build class diagrams, add attributes and operations, then code from that
- build class diagrams, then sequence diagrams for certain important scenarios, and then code
- start from sequence diagrams derived from the use cases, abstract classes from the scenarios, and then code
- build classes, then state machines for each class, then actions for each state, and then translate the models into implementation.

There are more approaches still for non-real-time work such as workflow analysis and IT systems. There is no agreement on which approach is best, even under specific circumstances.

Does this imply necessary connections between the diagrams? For example, you said "[build] classes, then state machines for each class" Does that mean there's one state machine for each class in the UML?

The UML does not prescribe this. You can build a state machine for anything that takes your fancy. But you can constrain the UML to follow this rule using a *profile*.

What's a profile?

Formally, a profile is a UML model that describes extensions and subsets to UML. Subsets are described using the Object Constraint Language (OCL). For example, the constraint above could be written:

```
      Forall Class.StateMachine.allinstances() -> size() <=1
```

Constraints like these can be used to define how the elements of UML fit together; they can also be used to subset the language. For example:

```
StateMachine.allinstances() -> size() = 0
```

requires that there be no state machines in a model.

Extensions are created by defining stereotypes, which are tags that can decorate any model element. For example, we may tag a class "persistent" and use the tag to identify a class whose instances are stored past the lifetime of the runtime of the system.

Informally—and this is ideologically unsound—a profile is any set of extensions and subset to UML whether written down using these mechanisms or not. Formally, a profile is the OCL and stereotype definitions that describe the rules, which in UML 2, are captured in a package.

What about the real-time profiles?

The *UML Profile for Modeling Quality of Service and Fault Tolerant Characteristics and Mechanisms* (http://www.omg.org/docs/ptc/04-09-01.pdf) defines stereotypes for properties such as performance, dependability, security, integrity, coherence (temporal consistency of data and software elements), throughput, latency, efficiency, demand, reliability, and availability. Each of these properties is itself broken down into specific stereotypes. Throughput, for example, defines stereotypes "input-data-throughput", "communication-throughput", and "processing-throughput". The profile defines extensions for fault-tolerance and risk assessment. My favorite here is "ThreatAgent" which can be denoted by a stick-figure icon holding an anarchist bomb.

You can decorate your model from this catalog of icons and properties to define the real-time requirements of your application problem. Tools may then use this information to construct implementations, though at present this is not directly possible. Consider a class with a "communication-throughput" value over some observationInterval. Now what? There are many ways of realizing this requirement; tools are not yet able to implement them. Nonetheless, having a standard set of concepts defined for characterizing real-time systems is valuable.

Further work is taking place with Modeling and Analysis of Real-Time and Embedded systems (MARTE), which requests a UML profile to add capabilities for analyzing schedulability and performance properties of UML specifications.

For the last eight or seven years, different "real-time" UML tools have been released? Why so many different solutions? Why such a lack of standardization over such a long time?

UML, as I suggested above, is a family of languages, and UML 2.0 also incorporates some concepts from SDL. Each tool set defines one member of the family. There are different solutions because these tools each target a different subset of the market, but they are all pretty much based on the UML standard.

What should I look for in a tool?

That depends on how you want to go about developing systems using UML. If you intended to use it as a sketching language, your requirements will be quite different from what you look for if you want execution. There's no substitute for doing the research, especially as tools grow and change over time.

Does this mean I have to have an agreed development process to use UML?

You need to have an agreed development process, with or without UML, if you're going to work together as a team. That does not require a "heavyweight" process—you could all decide that your process is as simple as "Talk to customer; Draw use cases; Write code; Test it; Feedback to customer." But if each person is doing something completely different, it will be difficult to work together.

How does the UML fit into this?

If two people were working together and one builds models in the order use cases, sequence diagrams, code, and the other builds them class diagrams, state machine diagrams, then actions, they would not be able to communicate as effectively as if they had some commonality. Bringing UML into a project is sometimes cast as an answer to the question "How do we build software?" ("We use UML!") In fact, it makes the question more acute.

Will using UML change my development process?

It doesn't have to, but often the introduction of UML is seen as an opportunity to make some changes. It is critical for your success that you consider what you want your process to be, and then make your use of UML fit that process.

Do you have a preferred process for real-time systems?

I build translatable models consisting of class diagrams, state machine diagrams for those classes, and write action language for the state machines. All of the processing in the system is housed in concurrently executing state machine instances, which helps capture concurrency in the application so it can be mapped into an implementation.

I didn't see use cases. Why not?

Use cases are remarkably popular, and with good reason; they help focus our attention on the requirements, and not on coding. Especially in real-time systems, with our hard-to-meet time and memory constraints, it's all too easy to fall into designing the system before acquiring an understanding of the problem. This is where use cases can help.

Use cases help us build executable and translatable models, but they do not themselves execute.

Tell me more about what you mean by "Translatable"?

An executable model executes, but it also embeds implementation decisions about classes, state machines, memory allocation, and so on. A translatable does not. Accordingly, the model must be translated into an implementation using a set of formalized rules that, for example, take a class and turn it into a C struct, a state machine diagram and turn it into a set of switch statements, a CreateObjectAction and turn it into a malloc(), and so on. Each set of rules must be internally self-consistent, and a complete set is called a *model compiler*.

What's a model compiler?

A model compiler is simply a set of rules that read a developer's model, as captured in a metamodel, and turns it into text. That text, of course, will often be consistent with programming language syntax (i.e. the text is a C program), that can itself be compiled onto a processor.

How is a model compiler different from a programming language compiler?

It's not. It's just that the level of abstraction of the "programming language" is higher, so much higher that the UML program is independent of its platform. You don't even need to make decisions about data structures. A model compiler, in fact, is exactly analogous to a programming language compiler, just at a higher level of abstraction.

You keep saying that, but what exactly do you mean "UML is at a higher level of abstraction."

When we write in C, say, `if a == b then` …the C compiler turns this into assembly language that loads registers , maybe subtracts `b` from `a`, and jumps if zero. On a different processor, it may do something altogether different. C has "abstracted away" the details of register allocation and even the concept `if`, turning it into GOTOs we would normally eschew.

When we write in UML that two classes have an association between them, we have abstracted away how the association is made. We do not say whether the implementation uses a linked list, a doubly-linked list, a table—just as we don't say anything about registers when we write C.

In blueprint-style UML, you abstract away details like this, but there is still a nearly one-to-one correspondence between the model and the code. In translatable UML, we go further and abstract away all data structure decisions, tasking structure decisions, language decisions, and leave these to the model compiler.

How many model compilers are there?

That's like asking "How many possible designs are there for the same problem?", which is actually a very interesting, if rather abstract, question. The answer is that there's one model compiler for each possible design; related designs may be housed in the same package using compiler switches. For example, we use a Java model compiler to build our own UML toolset, which is itself modeled. We offer a small footprint C-based model compiler for real-time and embedded systems.

Are the market leaders ready for model compilers?

Whether the leaders of the market are ready to adopt executable UML is a chicken-and-egg problem. Vendors won't build a solution until they see a market, and customers won't buy until they see tools on the market. Even a well-proven tool such as Mentor Graphics' BridgePoint, which has been in the market for ten years now, can be perceived as risky until there are standards.

But we already have a standard UML! Why do we need another standard?

Executable UML relies on a subset of UML, but each vendor could choose a different subset on which to base their model compilers. That would reduce interoperability. Moreover, the execution semantics of UML are not formally defined.

When can we expect that standard from the Object Management Group?

Initial submissions for the Executable UML Foundation (http://www.omg.org/cgi-bin/doc?ad/2005-4-2) were submitted in April 2006. The final submission is due in June 2007. It generally takes six to twelve months after that for the standard to be officially adopted, though of course you can use tools that conform to the standard before that. Assuming no delays of course…

Should I wait, then?

No.

OK. So what about applying this in the real world? How visible should my shiny new UML project be?

UML doesn't guarantee success, which, obviously enough, admits of the possibility of failure. In other words, it's a gamble: you have to decide how much you're willing to risk before you see how the project will turn out.

Most people instinctively go for the low risk approach, choosing a low-visibility project or a low-visibility piece of an important project. Of course with low risks, the rewards are lame as well. "Sure it worked for the logging facility, but no way would that stuff work for the real stuff I'm working on like the motor-control loop."

By the way, dilute this answer significantly for using UML as a sketching language, and a little for blueprint style. The investment—and therefore the risk—is less for sketching than for executable or translatable modeling. Consequently, it matters less how visible your project is. The same applies for the answers that follow.

How big should my first project be?

Adopting any new technology on a wide scale, all at once, is rarely a good idea. Pick a project that is manageable in size. It has to be big enough to matter, but it has to be small enough that you can steer it. Don't underestimate the force required to overcome organizational inertia.

What kind of application is best for a first project?

Don't pick a project that's basically algorithmic. Mathematics is a perfectly good “modeling language” for much of that. UML really helps in laying out the overall structure of an application, and it is particularly well suited for understanding concurrent behavior. So look for something that's complex enough in the control aspect that your best programmer would struggle to sort it out. Got concurrency? Model it.

What about legacy code?

If you have lots of it, you may spend more time trying to interface the generated code to it than it's worth. Pick a project where the return-on-investment for learning a new technology is high enough to matter. It doesn't make much sense to invest heavily to improve your productivity for a piece that represents 3% of the overall effort of the project.

How well should I know my application? Should I start with something new?

Start by modeling something you know really well. Don't try to learn a new subject-matter or push the envelope of one you already know while you're learning to model. After you're comfortable modeling, then use your new skills to explore more complex and unfamiliar areas, where you can get a greater return on the investment.

What kind of people should I get?

You need smart ones, and you need several different skill sets along with the intelligence. But more important than talent and wit is desire. No better way to ensure failure than to populate the project with a few folks who don't believe this new-fangled approach can work for their problem.

So how do I know if I’m ready for UML?

We never get to do anything that's ideal, so we end up making trade-offs between the desired project and what’s available. Only you can make these trade-offs. In some situations it's better to have a small, nearly invisible success than to risk a spectacular failure. In other cases, it might be just the reverse, where having success in a low-risk environment could just be the kiss of death for the approach. In the end, indecision is worse than failure. Pick a project, staff it with people and then make it work. It won't be easy. If it were easy, you wouldn’t be being paid the big bucks.

For some help in making the decision to proceed, take a look at the brief self-assessment below.

A Brief Self-Assessment

This self-assessment falls into several parts. The first is Where does it hurt?

Are you…?
…having difficulties capturing requirements?
…visualizing relationships between requirements?
…communicating requirements?

…communicating designs?
…visualizing interactions?
…understanding concurrency?

The more of these problems you have, the more helpful the sketching capabilities of UML will be.

Are you…?
…having problems communicating across groups?
…keeping track of your designs?
…finding some tasks repetitive?
…

Then the additional capabilities offered by code-frame generation in blueprint-style UML diagrams, will help.

Are you…?
…finding defects late?
…having quality problems?
…concerned about the model being out of sync with the code?

These pain-points call for executing the application early. Blueprints and sketches in UML will not help with this pain.

Are you…?
…deploying (nearly) the same functionality across multiple platforms?
…building products that need to integrate over time?
…and having the pain-points described immediately above?

These pain-points call for using translatable UML for separate reuse of the application and design.

Now let's move on to some conditions

How attached are you (and your team) to code. Do you believe that …
…code is the true system artifact?
…some code can be generated, but you'll usually have to change it after the fact?
…code can be graphical?
…change the models and the code will take care of itself, just like a compiler?

Do you (and your team) believe that…
…a model is to be thrown away?
…a model is useful for documenting the system?
…a model is the system?

…a model is potentially *several* systems

In the two questions above, if you plumbed for the first answer, sketching is for you; the second answer calls for blueprints, the third for executable models, and the fourth translation.

You also need to consider the following questions.

- Do you know how adopt new technology?
- Do you have a suitable project on which to use UML?
- How much are you willing to invest in tools?
- Can you quantify the benefits you could achieve in your current project?
- Do you have anyone who wants to make UML work inside your organization?

Finally, some questions for those of you interested in translatable UML.

- Are there any tedious portions of your application that suffer redundancy?
- Do you find that managing a distributed engineering team causes integration issues?
- Is true platform independence important to you?
- Would you like to be able to verify your design/models before going to target?

Conclusion

When considering moving to UML, you must consider what you hope to get out of it. If you're looking for visualization of software structure, low investment, and very low risk, using the UML informally as a sketching language is for you.

If you are looking to specify systems using UML, you will need to invest more in learning the language and in tools to support it. You will need to think carefully about your process, especially for any team effort.

If you hope to reduce defects and remove the potential for the model to be out of sync with the code, you need executable models

If you're looking to reuse either the models or the implementation, you need fewer defects, and the ability to optimize globally, translatable models are the way to go.

Acknowledgments

The form of this article was inspired by Pierrick Arlot who published an interview with the author in *Electronique International*. M. Arlot seeded the initial list of questions.

John Wolfe supplied several answers especially regarding projects preparing to use executable models. The team at Accelerated Technology provided some of the questions in the self-assessment.