Editor: Eoin Woods
Artechra
eoin.woods@artechra.com

Harnessing UML for Architectural Description—the Context View

Eoin Woods

THIS COLUMN DISCUSSES architectural description and the process of representing and communicating your designs. What might surprise you is that despite my previous criticisms,[1] I'm going to talk about UML and why I find it useful when creating architectural descriptions.

UML is widely taught in academia but in my experience isn't used so often in industry. So why do I persist with it? To explain why, I'll show you how I find it useful by using it to describe an important view of a system (the context view).

Why Use UML at All?

When you create a description of a computer system, you're creating a model of it, a simplified representation that abstracts away many details to leave a clear definition of its key concepts. The model then lets you record, communicate, and analyze the important aspects of the design.

One problem people have with UML is that it doesn't contain many of the building blocks you need to describe the design of a typical system. I think this is where people often go wrong when using it. They end up with diagrams like the one in Figure 1, which doesn't provide much information. You can tell that five "things" are related to each other, but that's about all.

So why persist with UML if it provides so little assistance with creat-

0740-7459/14/$31.00 © 2014 IEEE

ing an architectural description? For me, there are two reasons. First, you can easily extend it to add whatever concepts you need. Second, you can harness tool support to make your models much more useful than mere pictures. In addition, because many software practitioners understand its syntax and conventions, they can understand it with little explanation.

Let's consider those reasons in more detail.

A Basis for Other Languages

People are often frustrated with UML because they want to describe their system in terms of element types that aren't part of the language, such as servlet containers, SQL Server databases, and JavaScript user interfaces, along with equally specific relationship types such as message queues to link them together. Because the base UML language doesn't include these concepts, people give up and define their own informal notations. However, with a little work (and ideally a decent modelling tool), you can extend UML to contain whatever element types you need. I used MagicDraw when writing this column, but many mature UML tools are available these days.

Figure 2 illustrates this with a much more specific version of the diagram in Figure 1. This diagram communicates more information than the generic UML diagram. To do this, it uses specific element types with annotations and their own icons, along with specific connector types that clarify the nature of the interactions.

To create UML models like this, you create a UML profile containing the definitions of your new element types (such as Java Server or Java Database Connectivity (JDBC) connection), which are defined in UML as "stereotypes." A stereotype is a UML extension that has a name and can have a set of specific attributes (tags). You can also specify a new symbol for instances of the type on diagrams, which makes the diagrams much easier to understand than if everything is an oblong box. Daniel Moody's 2009 paper, "The Physics of Notations," is full of good advice for defining your own graphical notations.[2]

I've observed that most people who create effective box-and-line diagrams have a specific vocabulary of element types they want to represent. So, UML profiling gives you an easy way to capture those types once and reuse them as many times as needed.

Models Aren't (Just) Pictures

Every architect I know draws pictures to explain ideas to people,

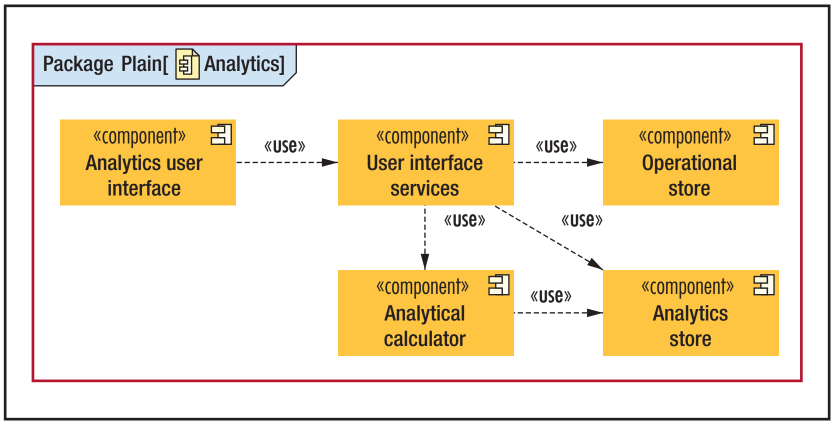
Package Plain[Analytics]
«component» Analytics user interface
«use»
«component» User interface services
«use»
«component» Operational store
«use»
«use»
«component» Analytical calculator
«use»
«component» Analytics store

FIGURE 1. A generic UML diagram. This diagram doesn't provide much information. You can tell that five "things" are related to each other, but that's about all.

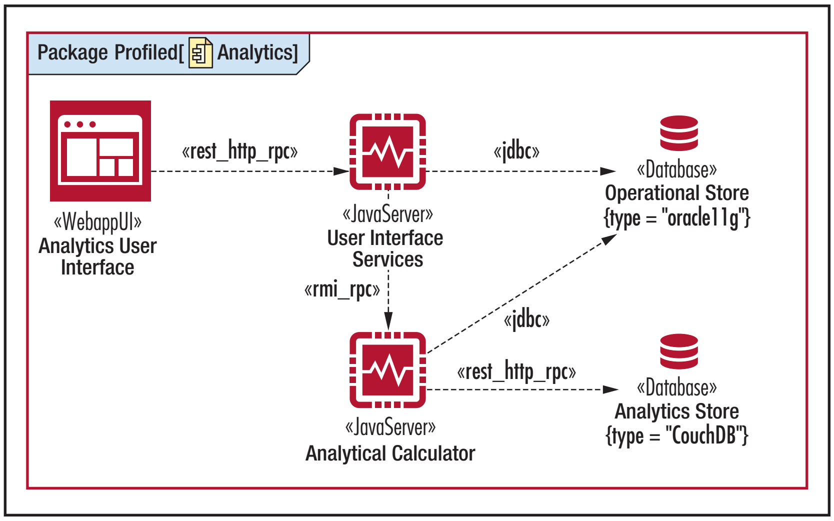
Package Profiled[Analytics]
«WebappUI» Analytics User Interface
«rest_http_rpc»
«JavaServer» User Interface Services
«jdbc»
«Database» Operational Store {type = "oracle11g"}
«rmi_rpc»
«jdbc»
«JavaServer» Analytical Calculator
«rest_http_rpc»
«Database» Analytics Store {type = "CouchDB"}

FIGURE 2. This specialized UML diagram provides a much more specific version of Figure 1. It uses specific element types with annotations and their own icons, along with specific connector types that clarify the nature of the interaction.

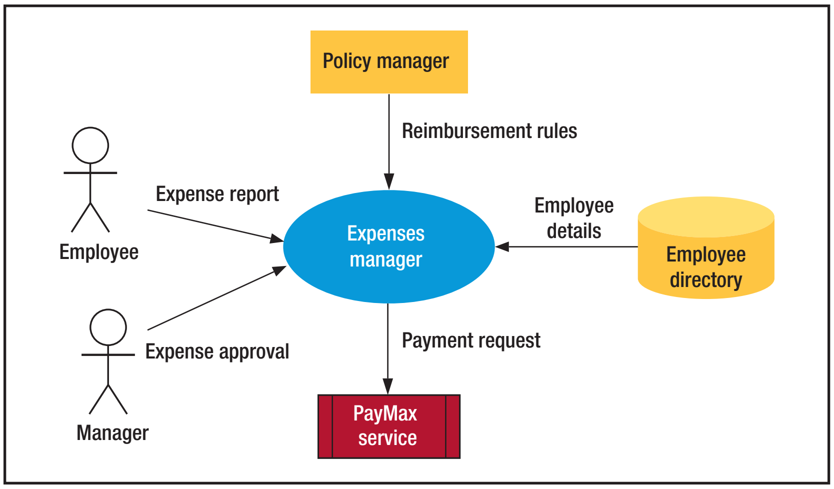

FIGURE 3. A box-and-line context diagram.

sometimes on paper and scanned, sometimes using a whiteboard with a digital photo for posterity, or sometimes using Visio or PowerPoint to capture something for longer-term use.

The problem is that all these technologies capture pictures, not data. If you need to create the picture and just make a few changes to it, they work well. But if you want to use the data it represents more than once, you're stuck (information like "A calls B" and "C accepts messages from D"). You've probably been in the position of having a Visio diagram that you needed two "views" of, so you had to make a copy and manually keep the two in step as they got updated.

If you can capture your model in a machine-readable form (in a UML tool of some sort), you have the underlying data as well as pictures. So, you know that if you rename an element, it will be renamed everywhere it's used (and you can retrieve more information from the model, such as the type of database in the example in Figure 2). This can help considerably if you're working with a model for a long period and changing it over time.

So how does this help you create better architectural descriptions?

The Context View

The context view describes the relationships, dependencies, and interactions between the system and its environment (usually the people, systems, and external entities that it interacts with).[3] It's an important view for a number of stakeholder groups, yet it isn't part of the base UML language. However, you can extend UML to solve this.

Figure 3 shows a simple context diagram with informal notation. As you can infer from the diagram, the system manages the making, approving, and paying of corporate expense claims. Employees create expense reports and managers approve them. The system imports a set of rules governing allowable-reimbursement policy from an external policy manager system. It then accesses the employee directory to retrieve employee details and sends payment requests to PayMax, an external service.

Given the base UML language doesn't include this diagram's element types, how would you go about creating it in UML?

The context view is simple and doesn't need that many concepts to represent it. You need a representation of the system, representations of the external entities it's connected to, and a set of relationship types indicating a connection's characteristics. As I showed earlier, you can add them to UML using a profile of stereotypes.

So for the context view, you can add stereotypes for the system, for different types of external applications, perhaps for data stores that the application accesses, and for the various sorts of connectors that the application has with these external entities. You can then bundle these stereotypes into a UML profile you can reuse whenever you need it.

Figure 4 shows a context view created using UML extended with a profile defining the additional concepts it needs (directory, external service, application and system elements and LDAP, message queue messaging, and flat-file connections). The two diagrams have the same underlying model; one has less detail (perhaps to show acquirers or end users), and the other has more detail (perhaps for development teams or infrastructure designers to use). The important point is that there's one underlying model and the diagrams are just representations of it. So, if you change something (perhaps rename the Policy Manager), you don't need to worry about keeping things in sync—the diagrams are automatically consistent.

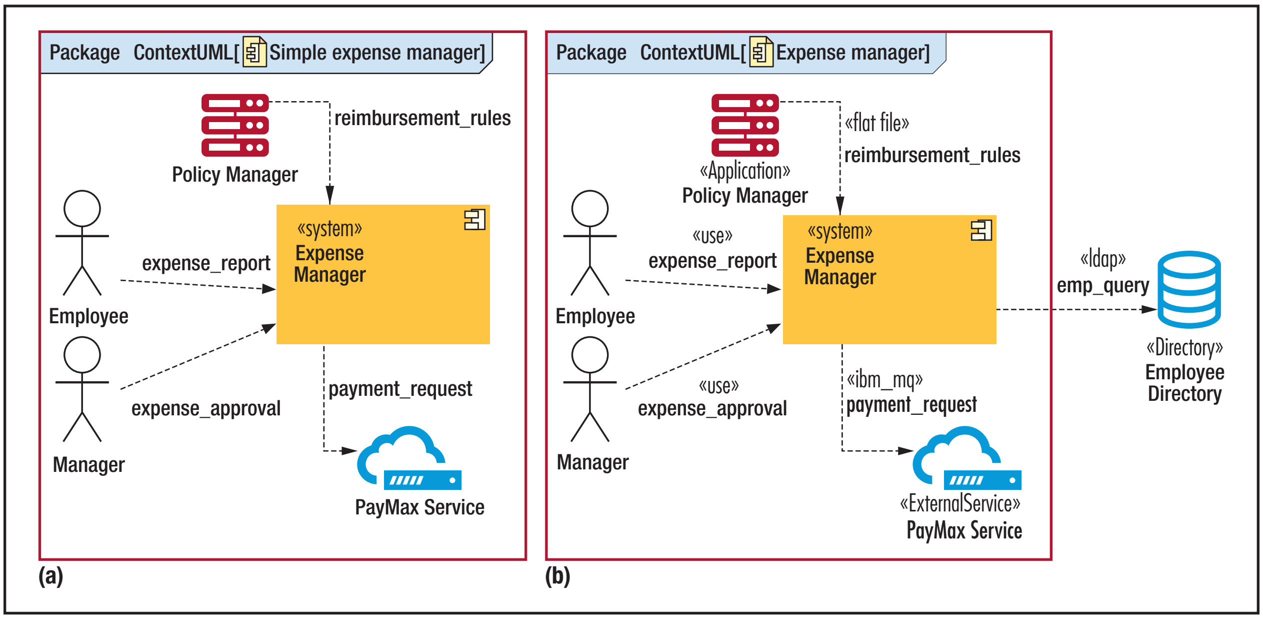

FIGURE 4. Context views using a UML profile, (a) overview and (b) detail. There's one underlying model; the two diagrams are just representations of it.

So what have you achieved with this? By extending UML, you have retained much of the expressiveness of a less formal picture, but you still have a model, not just a picture. So, you can use the underlying model data in different ways. It also means that if you need to update something, you know that it will be automatically reflected everywhere. This isn't a big deal in a simple model. However, as soon as models become large, keeping multiple representations in step when changing them becomes time-consuming and error prone.

UML has fallen out of favor in mainstream software development practice; I think this is due mainly to how we used it. We tried to create all-encompassing models that were far too detailed. We also tried to use the base UML language, with its limited set of constructs, to represent our specific and rich design domains. So, we switched to use richer informal notations instead.

However, although "boxes and lines" are definitely useful, they have the fundamental limitation that you end up with a lot of separate pictures rather than a model. This can lead to immediate inconsistency when you change things, and it prevents you from using the model as data (for example, to generate reports on your system designs).

Of course, it doesn't make sense to tangle with UML and a modelling tool if you just need a quick sketch of something; it's all about context. You wouldn't write your shopping list in JSON, but then you wouldn't store complex configuration data as free text. Architectural models are similar. Boxes and lines are great for short-lived models, but an extended form of UML can be a really useful addition to your toolbox when you have complex or long-lived models and you want to unlock their value by using them as data.

References

1. E. Woods, D. Emery, and B. Selic, "Point/Counterpoint," *IEEE Software*, vol. 27, no. 6, 2010, pp. 54–57.
2. D.L. Moody, "The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Trans. Software Eng.*, vol. 35, no. 6, 2009, pp. 756–779.
3. N. Rozanski and E. Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley, 2011.

EOIN WOODS is a software architect at Artechra. Contact him at eoin.woods@artechra.com.