Proceedings of the Twenty-Fifth Annual International Computer Software and Applications Conference (COMPSAC 01), Chicago, Illinois, pp. 113-114, October 2001. Copyright 2001 by IEEE.

From Software Architecture to Implementation with UML

Christine Hofmeister
Lehigh University
crh@eecs.lehigh.edu

Robert Nord
Siemens Corporate Research
rn@sei.cmu.edu

Abstract

Although originally developed to describe OO design, today UML is also being used to describe software architecture. However, using the same notation for different levels of abstraction can create confusion. In this paper we illustrate some of the important differences between software architecture models and implementation models in UML.

1. Introduction

Although the Unified Modeling Language (UML) is more commonly used to describe the implementation of a system, recently software architecture researchers have begun adapting and using it to describe software architecture ([1] [2] [3] [4]). This causes concern because the same language is being used to model different levels of abstraction, thereby potentially increasing the confusion about what the software architecture of a system is.

Here we sketch a simple example to illustrate a few of the differences between a software architecture model and an implementation model in UML. We use four views to describe the software architecture: conceptual, module, execution, and code architecture views [1].

The example is a registration system, RegSys, that handles students' requests to be added or dropped from a class. The requests are accumulated, and perhaps once a day the registrar updates the class lists. A class list can be retrieved at any time.

2. Software Architecture Views

First we design the conceptual architecture view, which describes how the system's functionality is mapped to *components* and *connectors*. We start with a single component to represent the entire system, and describe its interface as *ports* of the component.

The RegSys component is the outer component in Figure 1. It has ports update and read. RegSys is composed of components Registrar and Classlist, which interact with each other using the connector Batchupdate.

The Registrar component accumulates add and drop requests via port update. When it receives a request to update the class list, it sends the accumulated list of add requests using port addlist, and the list of drop requests using port droplist. Each of these ports will send a list then wait for an acknowledgement.

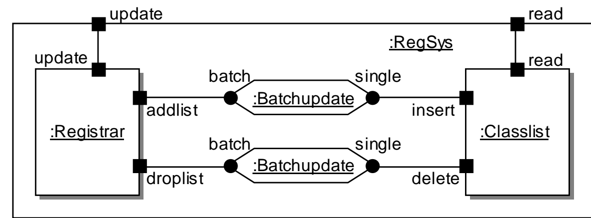

Figure 1. Conceptual Architecture View

The Classlist component accepts insert and delete requests on ports insert and delete. Upon receiving an insert or delete request, the component updates the database accordingly. It is also returns a text version of the class list when it receives a request on port read.

In this system, there are two occurences of connector Batchupdate, one for transmitting the add requests and one for the drop requests. Just as components have ports at their interface, connectors have *roles*. Batchupdate receives a list of requests on role batch, then it must send these requests one by one on role single, each time waiting for an acknowledgement before sending the next request. When it has successfully transmitted all requests in the list, it returns an acknowledgement on role batch.

We have described the interactions of the ports and roles informally here. In practice we define protocols to be obeyed by ports and roles, based on the work of [4]. Where appropriate, we also use state machines to describe component and connector behavior.

Next we turn to the module architecture view. It describes how the components, connectors, ports, and roles are mapped to abstract modules and their interfaces. A system is decomposed into subsystems and modules, and a module can also be assigned to a layer, which then constrains its dependencies on other modules.

For the registration system, the mapping between conceptual and module view elements is simple. Component Registrar, connector Batchupdate, and their respective ports and roles are mapped to module MRegistrar. Component Classlist and its ports are mapped to module MClasslist.

Module MRegistrar has one interface IRegistrar, but module MClasslist has two. The interface IClasslistUpdate

Proceedings of the Twenty-Fifth Annual International Computer Software and Applications Conference (COMPSAC 01), Chicago, Illinois, pp. 113-114, October 2001. Copyright 2001 by IEEE.

is for the system's internal use, and IClasslistRead is for external access, e.g. via a user interface. The details of these interfaces can be seen in Figure 2. This figure also shows the dependency of MRegistrar on IClasslistUpdate, derived by following the connections defined in the conceptual view.

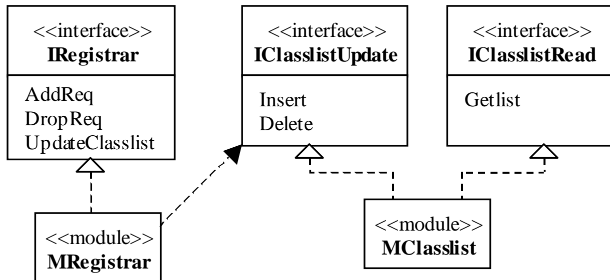

Figure 2. Module Architecture View

The execution architecture view describes how the system's functionality is mapped to runtime platform elements such as processes, shared libraries, etc. The code architecture view describes how the source code is organized. These decisions can affect the other architecture views: for example, the decision to combine the component Registrar and connector Batchupdate into one module was made because both are assigned to the same process in the execution view. The execution and code architecture views also provide crucial information for the implementation. However, because of limited space we don't describe these views here.

3. Implementation

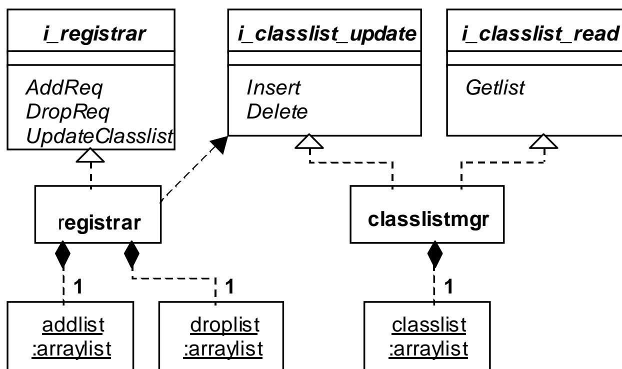

Figure 3. Implementation Class Diagram for C++

Turning to the implementation of this system, we start by creating one implementation class for each module and each interface. In general we expect a module to require multiple classes for its implementation. Here the developer plans to use an existing class arraylist from a utilities library. It will be used for the addlist and droplist objects created by the registrar, and for the classlist managed by the classlistmgr (Figure 3).

Figure 3 contains an abstract class with abstract operations for each interface in the module view. These interfaces will be implemented in different ways depending on the programming language to be used. If the language directly supports interfaces, then the <<interface>> stereotype can be used in the implementation model. In C++, a header file is not a true interface, so the developer must create an abstract class to serve as the interface.

4. Discussion

As similar as some of these diagrams look, they represent very different things. We have briefly described a few of the differences in this paper:

- A module is abstract, but the implementation classes are programming language specific. The module usually represents multiple implementation classes, and doesn't even have to be implemented in an OO language. The implementation classes, on the other hand, reflect the constructs that are available in the language, such as whether interfaces and multiple inheritance are supported.
- As this example illustrated, reuse is important for both the software architecture (connector Batchupdate) and the implementation (class arraylist), but reuse of implementation classes is not always relevant to the software architecture.

Expecting or forcing the architecture models to seamlessly evolve into implementation models can increase confusion and contribute to errors.

We do advocate the use of tools that couple the implementation models to the source code (generating code from models; extracting models from code). The extracted models are also helpful to a software architect when reverse engineering the architecture. But automated extraction of software architecture models is very much an open research problem.

5. References

[1] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, Reading MA, 2000.

[2] P. Kruchten, *The Rational Unified Process*, Addison-Wesley, Reading MA, 2000.

[3] N. Medvidovic and D. Rosenblum, "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures", *Proc. of the TC2 1st Working IFIP Conf. on Sw. Arch. (WICSA1)*, Kluwer, Boston MA, 1999, pp. 161-182.

[4] B. Selic and J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems", ObjecTime Ltd and Rational Software Corp., March 1998. (http://www.rational.com)