Third International Workshop on Component-Based Software Engineering: Reflection on Practice.

Educational Case Study – what is the model of an ideal component? Must it be an object?

Pat Hall
Computing Department, The Open University
Walton Hall, Milton Keynes
England MK7 6AA
+44 1908 274066
p.a.v.hall@open.ac.uk

BACKGROUND

I have been writing material for a second-level course on object oriented software development. The part I have been writing is on component-based software development, and patterns, frameworks, and product lines. To begin with I wanted to set the scene by defining an ideal component model, to break students out of the presumption that objects solved everything.

To do this I went back to earlier work on Module Interconnection Languages (MILs), following the ground-breaking paper by DeRemer and Kron (1976) but also drawing on other material from that stream of research, in particular the survey paper ten years later by Ruben Prieto-Diaz and Jim Neighbors (1986). This is an approach I have taken regularly in review articles of the area over the past two decades, for example in Hall (1999). Thus I see, and have always seen, it important to be able to treat component interfaces as first class elements, to break these up into coherent parts, distinguish between provided interfaces and required interfaces, and the need to make component interconnection a completely independent activity from component definition and implementation. I commented on the shortfall that I see in OO approaches.

As a matter of routine practice we send out drafts to critical readers, and my draft was sent to seven people, some of whom are very experienced practitioners, and others who are experienced educators. I got some surprising comment! Basically my critics believed that objects, particularly in the form of JavaBeans, are the ideal component. They could not engage with my view that making explicit the required interfaces is as important as making the provided interfaces explicit. Provided interfaces are equated with specification, a Good Thing. One claimed "OO has a longer history than Software Engineering ... in the guise of Simula 67, OO was, according to Bertrand Meyer ... predates Software Engineering by a few years".

It seems that the OO line is being swallowed, and that line is limiting our perceptions of what a component should be. If the view I have taken is right, then we should be actively be reforming OO models. Of course it could be that current CBD methods do do the right thing. But I have yet to find a proposed method that does – D'Souza and Wills(1999) and Jacobson Griss and Jonnson (1997) do not.

Here is the gist of my current explanation.

Begin Quotation

DEFINING THE IDEAL COMPONENT

Figure 1 illustrates the idea of components using an informal notation looking like electrical components with wires joining them. The white boxes are the components. A large outer component is shown composed of two interconnected inner components. Each component has a number of interfaces shown as shaded boxes: required interfaces are the boxes with 'pins' called 'plugs', provided interfaces are shown without pins and called 'sockets'. Interfaces of the same 'type' are shown with the same shading and dimensions as in the box at the bottom of the diagram. Interconnections are shown as 'wires' connecting a required interface of one component to the provided interface of another component, or to some interface of the encapsulating outer component.

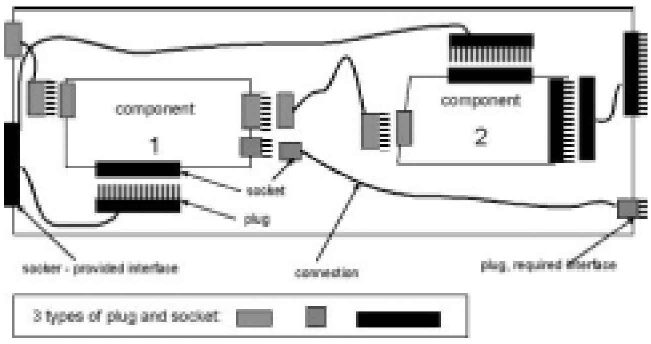

Figure 1. Components and their interconnection.

The critical point to note is that components have a number of points of interconnection: each point could be termed an 'interface', with some of these interfaces being provided for other components to use, while other interfaces are required from other components. Components are completely ‘encapsulated’: there is no way of doing anything with a component other than by using its provided interfaces.

This gives a set of requirements for any method of specifying and implementing components, which are:

- ❑ a *component* can have a number of *interfaces*, which must be able to be named and defined separately,
- ❑ each interface consist of a coherent set of *operations*, each of which can be separately identified, and specified with the input and output parameters named and typed. Together these are the *signature* of the interface.
- ❑ interfaces that provide the services of the component are called the *provided* interfaces,
- ❑ the interfaces of other components that are required to complete the services of the component must also be defined explicitly. These are called the *required* interfaces.
- ❑ the only way that a component can interact with other components and the rest of the software is through these interfaces. There can be no hidden or private connections. This idea that an interface can only be used through its public interfaces is known as *encapsulation*.
- ❑ the actual processing carried out by the component should in principle be capable of complete and unambiguous specification independently of any actual implementation of the component.
- ❑ Other components and software that uses a component may only rely on the defined interfaces and specified operations; no assumptions may be made about the implementation. This idea that no knowledge of the internal workings of a component can be used is known as *information hiding*.

What is required for component interconnection is:

- ❑ components can be interconnected to create a larger subassembly or ‘super-component’, thus making a component a *recursive* construct.
- ❑ required interfaces are connected to provided interfaces *explicitly*.
- ❑ interfaces that are connected together must *match signatures*, that is they must have the same operations with the same input and output parameters. (This can be relaxed a little to permit matching under inheritance rules.)
- ❑ some interfaces can be hidden as *internal* to the super-component, while others are made *external* and visible outside this super-component.
- ❑ additional *‘glue’ code* can be inserted between components to enable simple conversions to match interfaces where type substitutions are not sufficient.

Taken together these are an exacting set of requirements that several module interconnection languages have met, but no widely used programming language has ever met.

Example 1.1 shows a simple component built out of sub-components. Example 1.2 shows the simple component of Example 1 Figure 2, built out of sub-components, designed using UML.

Example 1.1
We are developing software for timetabling courses in a computer training school. Let us start by considering the requirements and from these develop a component model in the style of Figure 1.1.

We have identified that we will be concerned primarily with lecturers and classrooms and student-cohorts. All classrooms will be equipped with computers and a digital projector. We will need to timetable each of these, but also specialist devices like video players.

There are three basic “things” here:

- ❑ a Resource which can be a lecturer, classroom, or video player,
- ❑ a Schedule which allows us to determine when a particular resource is available or in use, and
- ❑ a Course, which brings the appropriate resources together for the appropriate length of time.

We will make each of these into a component, but before we can draw the component diagram, we need to explore the requirements a little further.

When we timetable a Course, we will need to find out for each resource required what its availability is and then select dates that suit all the resources involved. This timetabling could happen through interaction with a person who proposes dates, then checks that the resources are available, and then decides on particular dates. However the timetabling could also happen automatically: at this point we are not too concerned. This timetabling capability could be part of the Course component, but we will make a separate Allocation component that creates a new Course when asked to do so, and then timetables it.

Second, let us consider Schedules. Let us develop the interfaces required for the Schedule component further. We will need a set of facilities for checking the availability of a resource, and a set of facilities for reserving the resource for particular dates. At this point we do not need to consider in detail the actual functions of these interfaces. We may also want to cascade Schedules, so that the availability of a resource may be determined not only by its own schedule, but also by other constraining Schedules. For example, the availability of a lecturer is determined not only by his/her own commitments, but also by public holidays and by any company constrains like fixed closures over Christmas or annual conferences.

A Resource and a Schedule are intimately connected, and we will consider a composite of these as a Schedulable Resource. This leads us to the Component model in Figure 2 –note the rather more formal notation, still distinguishing between provided and required interfaces.

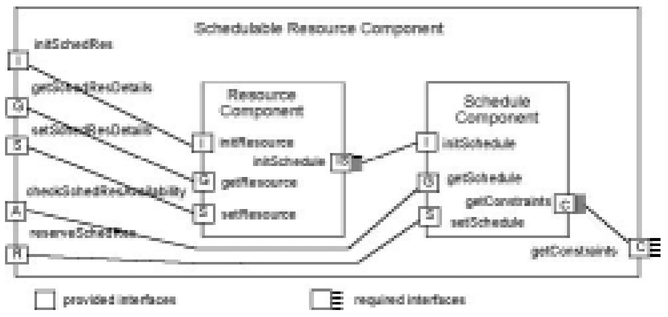

Figure 2. The generic Schedulable Resource components built from generic Resource and Schedule components.

The model of Figure 2 is highly generic, so let us now look at TTS, the timetabling system, built from these components for our client CTS, the computer training school. This is shown in Figure 3 which gives a partial instance diagram for CTS when it has 3 classrooms, 2 lecturers, and 2 courses. The Allocation object records within it the current courses and resources, and requires 3 interfaces, one for creating Courses, one for checking the availability of Resources and the other for reserving Resources once a set of appropriate resources has been found. We also see the cascading of Resource Schedules so that when the availability of any Resource is checked, that Resource in turn checks for more general constraints imposed by company closures, public holidays and weekends.

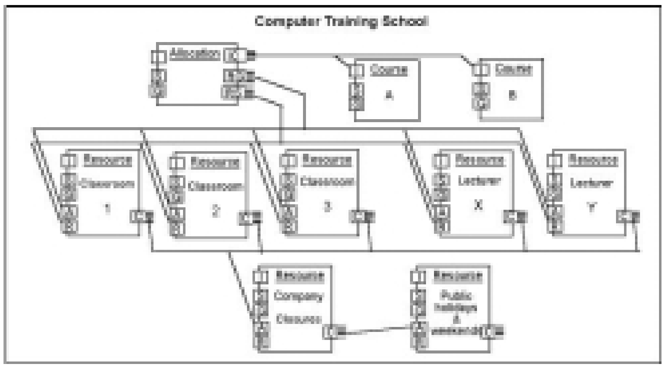

Figure 3. The generic Schedulable Resource components built from generic Resource and Schedule components.

Example 1.2
Let us now decide how the implement the component shown in Figure 1.2 and describe that implementation in UML. In principle we would like to make the heart of the component an object, but how do we make the interfaces explicit? A class's interface is a single collection of the provided operations that cannot be divided into subsets. We want to be able to divide up that interface so that we can control what can be connected together. And we also want to be able to make explicit the required interfaces. So a first shot might be to make all the interfaces, both provided and required, into classes as well, with the components themselves being packages. The connection between classes is shown by the realises relationship, where we are broadening what this means. (Note that we could have used UML stereotype <<interface>> for the provided interfaces.)

Figure 4 shows this for the component Schedulable Resource, while Figure 5 shows the result for the Resource component, where the interfaces are decomposed into their constituent operations.

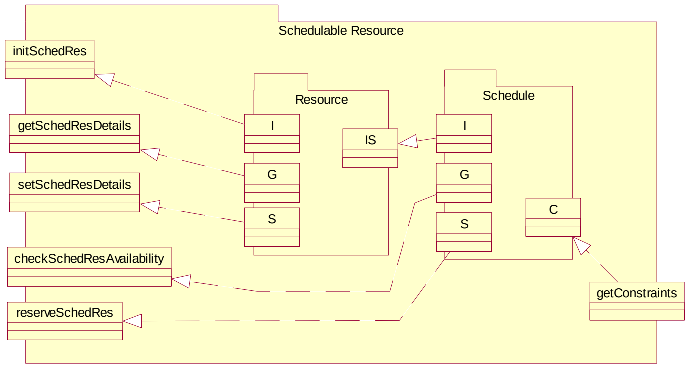

Figure 4 UML design for the Schedulable Resource component.

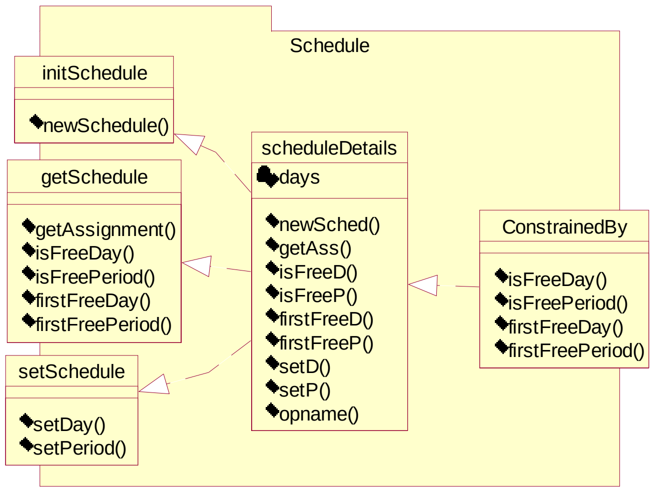

Figure 5 UML design for the Schedule component.

The interfaces are very simple. In the provided interfaces the implementation of the operations consists of the invocation of the operation that realises it. For required interfaces, in the implementation of the component, instead of invoking an operation outside the component the operation of the required interface class is invoked. Then to connect a component to another component the required interface operations are implemented to invoke the required interface of the other component.

End Quotation

THE OPEN QUESTIONS

What is the ideal component model?

If this ideal is not an Object in the current OO sense, then how can OO approaches be developed to embrace this ideal?

Does it really matter?

REFERENCES

Barroca, Leonor, Jon Hall, and Pat Hall (1999) (Eds) *Software Architectures. Advances and Application*. Springer.

Cox, Brad J. (1986) *Object-Oriented Programming* Addison-Wesley. ISBN 0-201-10393-1

De Remer, F and H.H. Kron. (1976) "Programming in the Large versus Programming in the Small", *IEEE Transactions on Software Engineering*, June 1976, pp312-327

D'Souza, Desmond and Alan Cameron Wills (1999) "Objects, Components, and Frameworks with UML. The Catalysis Approach". Addison-Wesley 1999.

Hall, Pat (1999) Architecture-driven Component Reuse" *Information and Software Technology*, vol 41 no 14, 15 November 1999. pp963-968

Jacobson, Ivar, Martin Griss and Patrik Jonsson (1997). *Software Reuse. Architecture, Process and Organisation for Business Success*. Addison-Wesley

Prieto-Diaz, Ruben and James Neighbors, (1986) "Module Interconnection Languages", *Journal of System Sciences*, vol 6, no, 4, November 1986. pages 307-334.