Modelling Software Components

Stuart Kent, John Howse and Anthony Lauder

School of Computing and Maths,
University of Brighton,
Lewes Road, Brighton, BN2 4GJ, UK
+44 (0) 1273 642494
Stuart.Kent@brighton.ac.uk

Abstract

This paper makes two contributions. (1) it argues that precise visual modelling techniques are important for modelling large-scale software components, as they facilitate the core activities of component-based software development (CBSD): building, finding, adapting and assembling components. The paper argues for a carefully selected set of techniques based on UML, to provide accessible yet precise component models. (2) it proposes a high level reference model for CBSD to tease out exactly what is meant by the terms 'component', 'component adaptation' and 'component assembly'. The paper illustrates this reference model by giving examples of components, and the transformations that can be applied to them, using precise visual models.

1. Introduction

It is widely recognised (so much so that it is built into low-level implementation technologies for CBSD) that components should at least come with a description of the interfaces they support, where an interface is a list of operations with their signatures. Whilst this may be sufficient for small-scale components such as GUI widgets, it does not take much reflection to realise that an interface as characterised is wholly inadequate for describing large-scale business components. How can I tell what the effect of performing operations is on the component just from the name and signature? (Answer: "Half the documentation of a program is in its names" – Boundy, 1991.) How can I search for components, let alone have a software agent go and do the searching for me without a more detailed specification? (Perhaps it will be quicker just to build my own… .) How can I be sure that the component actually does what is claimed of it in the marketing hype? (Well, this is all a bit imprecise and hype is hype, so you should expect it not to fulfil your expectations… .)

Of course, interfaces are usually accompanied by informal textual descriptions. Provided these are well written, accurate and with plenty of examples, these will definitely help the developer use the component, and may help her find components. However, without a more structured and precise description, it is doubtful whether these can support automatic searching and matching. And it won't help tools which automate the process of component assembly - they'll need much more precise descriptions. In addition, without a precise description, the component provider still has some leeway in providing component executables that do not quite match the marketing hype.

What is really missing is a precise description of a component's behaviour. If components can not be given a detailed precise specification and be guaranteed to meet that specification then little advance will have been made. Developers will still spend considerable time honing and customising their components to fit the requirements imposed by their design, rather than being able to rely upon and use the specifications when constructing the design in the first place.

However, precision is not enough on its own. The description must also be accessible to users of the component, which also means that it must abstract away from the internal workings of the component. In addition, it must be amenable to manipulation by automated tools if any of the promised searching and assembly tools are to be realised.

In the software modelling community a range of precise visual modelling techniques are beginning to emerge that could fulfil the four requirements of precision, conciseness, accessibility and automatability. Specifically, the Unified Modelling Language (UML – UML Consortium, 1997; Fowler, 1997) has emerged from the rich history of visual modelling, to become an OMG standard which promises to become the de facto industry standard modelling language. With the inclusion of the Object Constraint Language (OCL), it now looks like an integrated, precise and suitably expressive subset of UML can be identified. OCL is a precise assertion

language which has borrowed much from the extensive body of research into Formal Methods (e.g. VDM, Jones 1991, and Z, e.g. Woodcock & Davies, 1996). Added to this are techniques for composing and specialising visual models (see e.g. D'Souza and Wills, 1998) and even richer, yet precise, visual notations (Kent, 1997; Gil & Kent, 1998; Lauder & Kent, 1998).

The importance of precision and rigour in the context of CBSD can not be over stressed. The intention is that software components should be reused again and again, and that component assembly and adaptation should be largely automated. The former requires that components be of a high quality, precisely documented and rigorously developed and tested. The latter can only happen if component descriptions are sufficiently detailed, precise and unambiguous.

Similarly, models must be accessible and abstract, so they allow users of the components to quickly ascertain what the component does, how to adapt it and how to assemble it with other components. They must be automatable, to support the construction of suitable CASE tools.

This paper identifies an integrated and precise subset of UML, and shows how it can be used to model components. Some UML compliant notations are also described, which increase the visual expressive power of the notations, including an ability to visually describe the processes of component adaptation and assembly.

The terms component, component adaptation and component assembly, have begun to take on a wide range of different meanings (Is a component an object, a class, a framework? Must it always come with an executable part, i.e. are there pure design components? What is the difference between grey, white and black box components? What components are easily adapted and what are not? What does it mean to assemble components? Etc.)

A second contribution of this paper is to propose a reference model for CBSD, in particular discussing the definition of component, adaptation and assembly. The model is designed to unify many of the conflicting definitions, and provide ways of pinning down the differences between various flavours of CBSD.

The paper is organised as follows. Section 2 describes a fictitious example illustrating the kinds of activities that could be involved in CBSD. This example is revisited in section 5 to examine the utility of precise visual modelling techniques in CBSD.

Section 3 describes a high level reference model for CBSD, using precise visual component models to illustrate the various concepts introduced. Specific visual modelling techniques exemplified here are singled out in Section 5.

Section 4 briefly reflects on the notations introduced so far, and identifies a number of areas where work still needs to be done. This also feeds into the analysis of Section 5.

It becomes clear in Sections 3 – 5 that CBD is not possible without considerable automated support from tools which:

- assist with the locating and assembling of components
- enable the construction of components of sufficient quality

Tool support is discussed in Section 5. A final section draws some conclusions and identifies directions for further research.

2. CBSD in the Future?

This section presents a hypothetical example which aims to illustrate the effect that CBSD *could* have on Software Engineering practice. The example will be returned to in Section 5 to examine the potential impact of precise visual modelling techniques.

Big City Finance have a large legacy trading system which does not work well with their other applications and which has proven difficult to maintain. They decide to rewrite their proprietary trading system. They adopt a strategy of:

Using components to recursively subdivide the development

> From a high level description of the trading system's business model, "subject areas" are identified by the development team. Each subject area is built as a coarse grained component, with a clearly specified interface. Similarly, each subject area is typically broken down into sub-components which can be worked on by individual members of the subject area team.

As development progresses management recognise that they are able to:

Reduce maintenance costs

> When redeveloping the trading system special emphasis is placed on maximising the extendibility of the application. In particular, great care is taken to provide "plug points" where new or modified functionality can be added simply by "plugging in"

new or modified components. The result is a huge saving in maintenance costs.

The development team notice that a significant portion of the application does not relate to their proprietary business process logic, but solves a generic problem likely to be solved by ready-built software components. Consequently, they decide to:

Reuse existing components in the development project

Searching in a software catalogue, they find a specification of a component which appears to match their needs and which is available from XYZ Software. They adjust their solution design model to include the specification of this component and other parts of the application are designed and built. The code from the XYZ company is installed, and the whole application works smoothly.

Satisfied with the component from XYZ Software, Big City Finance decides to buy-in further generic components. Consequently, they architect their application to match the interfaces provided by XYZ Software with the intention of integrating appropriate components. However, at an industry conference, ABC Software announce a new range of components. ABC claim the code is 50% faster and 30% cheaper to lease than similar components from XYZ Software. Hence, Big City Finance recognise that they are able to:

Substitute cheaper components

By writing a thin wrapper round components from ABC the development team are able to replace XYZ components with ABC components with no knock-on effects. The business user notices only that response time improves, and the bank's bills are smaller.

In addition Big City Finance need to rewrite their Billing System. The development team realises that it would be cost-effective to purchase, from a component catalogue, a large package, designed as a set of interlocking components, which covers several key business processes.

This package must work alongside Big City's existing Settlement system, as well as with other finance companies. Hence, the need to :

Interface to legacy and external systems

Working at the specification level, for the bought-in components and for components they will develop themselves, they produce a model showing how the various disparate parts of the application suite can work together. This requires them to write wrappers for the Settlement system and the systems of other finance companies. They refine the specification trading off the thickness of the wrappers against the requirements of the final system.

A Big City developer is working with a derivatives trader to prototype and implement a workgroup application to meet a new business opportunity with a small time-window for deployment. This time constraint mandates:

Rapid application assembly

Having understood the trader's description of the problem domain, the developer browses a business model catalogue which contains standard process models, a few of which seem to closely match the new tasks. The catalogue entry for the first of these tells her that some standard user interface components work well with this particular process. She finds these in the software component catalogue and selects those that appear relevant. A smart assembly tool then gathers both the UI and business model component descriptions indicated by the catalogue entries, and executes the application with the user. Some modifications to the functionality are identified, which the developer enacts by specialising some of the base components. Later that day, the application is delivered to the trader. The following day, the developer adds her extensions to the base components as new component specifications to the company catalogue for others to use.

The trader finds that this new application supports his daily tasks well, but now needs to perform some additional processing on some data in response to an unusual problem. Since this is an urgent one-off requirement he cannot wait for involvement by the development group and thus must undertake:

Business user smart assembly

He browses a catalogue of business objects to which he has access, finds a couple which seem to offer the required additional data and processing and, using a simple graphical tool, connects the objects together and routes the results to his desktop spreadsheet program. The software components underlying the business objects execute the required processing on data stored on the spreadsheet and the results are passed back in the requested format, ready for manipulation using the standard spreadsheet facilities.

3. A Reference Model for CBSD

Throughout the example in section 2, four core activities are involved (adapted from Short, 1997, p19):

- build (and publish)
- find
- adapt
- (re)use

The way we choose to characterise and describe a

component is critical to maximising the simplicity and effectiveness of these activities. Finding, adapting and reusing components has priority over building them, as a component, if it is any good, will be found, adapted and reused many, many more times than it will be built.

This section aims to give some definition to the terms component, component adaptation and component assembly (plugging). It illustrates this with a running example expressed as a series of precise visual models.

3.1. Simple components

Figure 1 depicts a simple component. It has one or more plugs, where a plug comprises an interface and some behavioural specification of that interface. It has a revealed behaviour section which may include further specification of the plugs (e.g. behaviour that relates the plugs) and any aspects of behaviour that need to be revealed to assist a developer in adapting the component. It has a section of hidden behaviour, which will typically be design and implementation details which are irrelevant to users of the component. Finally it has an executable part. It is conceivable that components may be delivered without any hidden behaviour or any executable part, in which case the component represents a piece of reusable modelling.

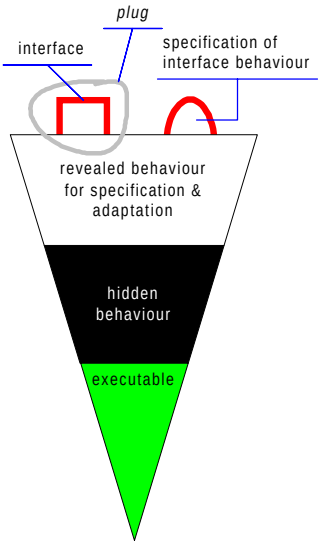

Figure 1: Simple Component

The shape chosen to represent a component is intended to give a sense of continuity in the behavioural model and a sense that as more behaviour is added to the model, the more constrained the component is in what it can do and how it is implemented.

To make this characterisation slightly more concrete, let us presume for the moment that views on a component are just class diagrams. Then the plugs may share some classes and some operations on those classes, but they may also have different classes and different operations. The revealed section must include the class diagrams for the plugs, suitably merged. The class diagram representing the whole model would include the revealed section as a sub-diagram.

Notice that we are making some assumptions about the nature of components. They are not (necessarily) single objects, or even single classes of objects. They are frameworks of classes which can be used to construct object configurations, where access to these object configurations is through objects and operations on those objects revealed in the appropriate plug. This definition incorporates the idea that a component can be a single object, as this corresponds to a framework with a single class that only allows one object of that class to be created. It also allows components where, e.g. one class is singled out as providing the object which initialises and controls access to all other objects. And we are happy to accept that there are some sub-categories of frameworks which are more appropriate than others for CBSD: some components are better than others.

The component depicted is a *grey box* component. A *black box* component is with no revealed behaviour except for the plugs. A *white box* component is one with no hidden behaviour.

Example

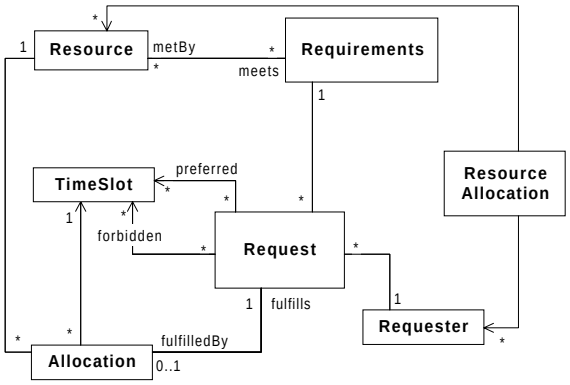

Figure 2: Resource allocation class diagram

Figure 2 is a class diagram describing the single plug of a generic resource allocation component. The diagram provides a precise language in which to describe the behaviour of the operations in the interface of the plug. For example, one of the operations supported by the interface is to allocate a resource. Its detailed behavioural specification is given in terms of pre and post conditions, which are written below both informally (essential for accessibility) and precisely in OCL (essential for quality and for automation).

ResourceAllocation::
allocate(Resource res, Request req, TimeSlot slot)

pre

slot is one of the preferred timeslots (if there are any) and not one of the forbidden ones

(req.preferred->includes(slot) or req.preferred->isEmpty) and not req.forbidden->includes(slot)

res meets the requirements associated with the request req

and res.meets->includes(req.requirements)

post

if no allocation already exists for req, a new one is created to fulfil req

(req.fulfilledBy@pre->isEmpty implies new->includes(req.fulfilledBy))[1]

the allocation of res to req in timeslot slot is recorded

and req.fulfilledBy.resource = res and req.fulfilledBy.timeslot = slot

The class diagram does not mean that this is how the class is implemented. Remember, this is a logical not a physical design. For example the diagram does not dictate that the implementation of ResourceAllocation must store a collection of requesters and resources. It may instead choose to store just the requests, and derive these two collections by navigating through the Request objects.

Another technique, UML object diagrams, is also available to document examples of typical behaviour. For example, Figure 3 strings two object diagrams in a sequence to illustrate what happens when an allocate action is performed.

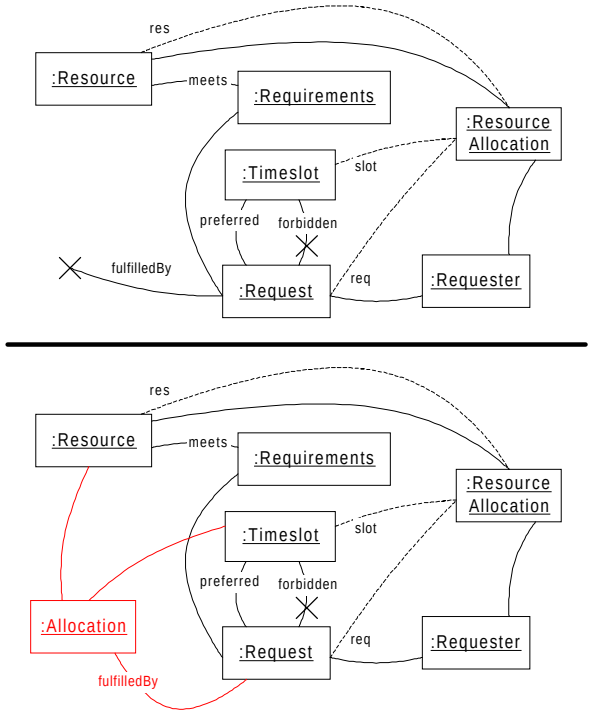

Figure 3: Filmstrip for allocating resource

Each object diagram depicts a (part of the) state of the component, showing, at some point in time, to which objects the component can navigate and how those objects are interconnected. Putting them in sequence shows how the depicted state changes as operations are performed.

A sequence of object diagrams is also sometimes referred to as a *filmstrip*, a term coined by D'Souza and Wills (1998) who first promoted the idea. Typically a filmstrip would be longer than Figure 3, and would be used to illustrate scenarios - how a bunch of operations in the interface are intended to be used together. Scenarios, at the specification level, may be documented using a variation on UML sequence diagrams (D'Souza and Wills, 1998).

Filmstrips only illustrate some aspects of behaviour, so, typically a collection of them is required to illustrate an operation or scenario. For example, Figure 3 only illustrates the case when no previous allocation has been made to the request.

Precise notation is also useful to place further constraints on the structures of objects permitted in a component. These are known as *constraint rules* in UML, though perhaps more widely known as *invariants*. Invariants are always attached to types (interfaces) or classes. For example, an invariant on ResourceAllocation would be:

ResourceAllocation

Given any allocated request req in requesters.requests

[1] new is shorthand for OclAny.allInstances–OclAny.allInstances@pre

requesters.requests->forAll (req | req.Unallocated implies

the requirements of req are met by the resource allocated

(req.fulfilledBy.resource.meets->
includes(req.requirements)

and the timeslot is one of the preferred ones - if there are some, but not one of the forbidden ones

and (req.preferred->includes(req.fulfilledBy.timeslot) or
req.NoPreferredSlots) and
not req.forbidden->includes(req.fulfilledBy.timeslot)))

This invariant assumes that states have been defined for the type Request, either in a state diagram or using dynamic subclassing on a class diagram. The state diagram (minus transitions) is given by Figure 4.

Accompanying this would be additional invariants relating states to each other and to the associations in the class diagram. For example, it is the case that a request can only be unallocated when it is pending, which is not enforced by the state diagram; and if it is in a state of NoPreferred then preferred must be empty.

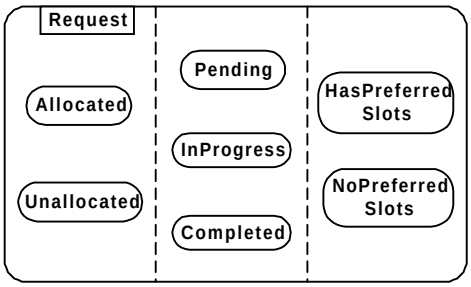

Figure 4: State diagram for request

Recently a diagrammatic notation, dubbed constraint diagrams (Kent 1997), has been developed which can be used as an approachable alternative or complement to the textually written constraints. Figure 5 is the constraint diagram which would be used to replace the first part of the above invariant, relating requirements of requests and resources allocated to those requests. This notation is currently being extended to a 3D modelling notation (Gil and Kent, 1998), which amongst other things, provides a clearer visualisation of operations over time.

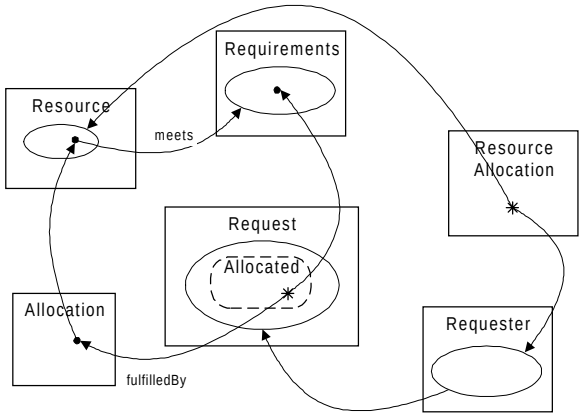

Figure 5: Constraint diagram for invariant

Object diagrams can be used to illustrate typical *snapshots* of the state of a component that are allowed and disallowed by invariants. Indeed for invariants, it seems to be disallowed states which are more useful.

A related set of snapshots and filmstrips may be referred to as a *specific model*, that is a specific example of behaviour. In contrast, the class diagram, invariants, operation specifications etc. make up a *generic model*. In general, the documentation of interfaces should include a repository of specific models illustrating typical behaviour of operations and typical component states. The repository should also include disallowed behaviour and disallowed states. Support provided by current CASE tools in this area is very poor.

These kinds of techniques facilitate far more rigorous integrity checks, with a corresponding increase in the overall quality of components. For example, specific models in the repository can be checked against the generic models. Do the links on object diagrams obey cardinality constraints on associations? Do they satisfy invariants, pre and post conditions? Etc. The integrity of generic models should also be checked. Do associations mentioned in invariants, pre and post conditions appear on the corresponding class diagrams? Are states mentioned defined on a corresponding state diagram? Do action specifications preserve invariants?

It may also be possible to make use of the diagrammatic nature of the notations by pattern matching e.g. between snapshots and constraint diagrams. Indeed one could even imagine this being done at run-time, much as components can be asked dynamically what interfaces they support. Instead, one could ask “Do you support an interface that conforms to mine - here are the diagrams to check against? If so create me an object”.

3.2. Multiple implementations

A component may have multiple "implementations": any

behaviour that is hidden (i.e. the black and grey parts) will not affect the users of the component; it is entirely up to the component designer which implementation she delivers.

3.3. Component assembly – plugging

Figure 6 shows the result of assembling components by plugging one into another. You will notice that the "A" component has a plug point which is the same shape as one of the plugs of the B/C component. The B/C component plugs into the A component by filling that plug point and bringing with it revealed and hidden behaviour and an executable that can be used to fill the hole beneath the plug point in the A component. In addition the B/C component has additional plugs and behaviour which also become part of the component resulting from this plugging process. In the result depicted here, the developer has chosen to reveal this additional behaviour. (S)he could equally choose to ignore it, in which case it would be hidden and play no further role in the resulting component, except as far as it contributes to the implementation of the B part of the B/C component.

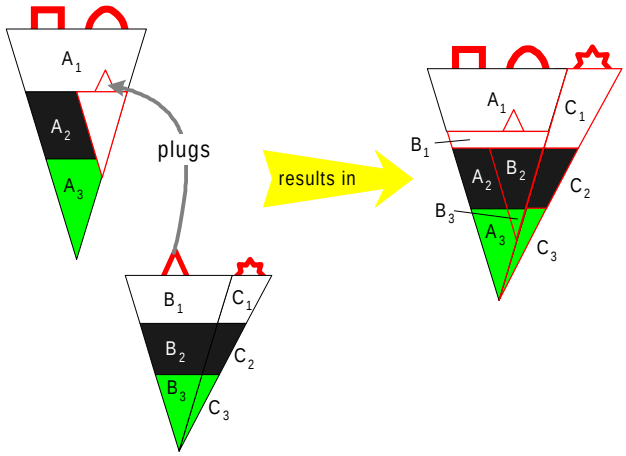

Figure 6: Component plugging

Example

Figure 7 shows a new component (InstructorScheduling) being constructed by plugging an InstructorQualification component into an InstructorAllocation component. The notation used is adapted from Catalysis (D'Souza & Wills, 1998). A dotted box indicates the boundary of the model of a component, which may include elements of all the various notations introduced so far. We have only shown a relevant subset of the elements. The "inheritance" arrow indicates that the child is the union of the two parents. By taking the union, common parts are merged. In this case the common part between InstructorQualification and InstructorAllocation is circled. This corresponds to the plug/plug-point.

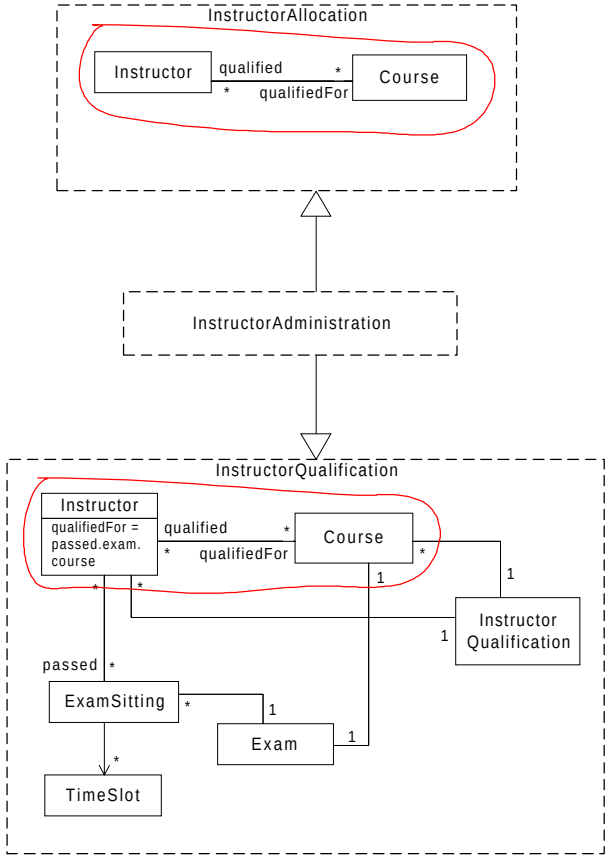

Figure 7: Plugging for instructor qualification

3.4. Component adaptation

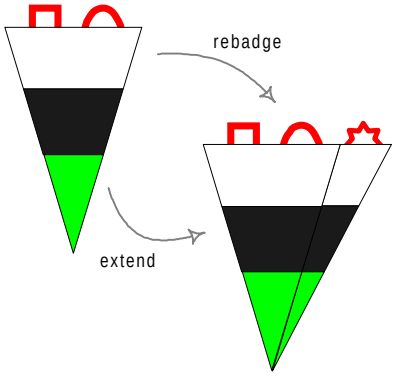

Figure 8: Adaptation through specialisation

Components may be adapted in at least two ways: through specialisation and through wrapping.

Figure 8 shows how a component can be adapted through specialisation. In this case, the adaptation is very simple: the component is extended with a new plug and with new behaviour, and some parts of it are rebadged. By rebadging we mean that e.g. some classes and operations

are given different names in the new component. There are variations on this. For example, specialisation may not involve adding any new plugs, it may just involve adding more to the revealed behaviour section. It also may be a more sophisticated affair: for example, the resulting component in Figure 8 is actually an adaptation of the A component, an adaptation that has been constructed by plugging in another component.

Although specialisation can be a powerful technique where one has some control over the both the definition of a component's plugs, it is highly unlikely that a third party catalogue will contain a precise match for the intended plug point. Thus we need a way of moulding the "best fit" component from a catalogue into a "perfect fit" component.

In the words of Shaw and Garlan: "Existing module interconnection systems typically require considerable prior agreement between the developers of different modules. To build truly composable systems, we must allow flexible, high-level connections between existing systems in ways not foreseen by their original developers." (Shaw and Garlan 1996).

One way to achieve the desired result is through *wrapping* (Bosch, 1992) Adapting a component by wrapping, essentially involves giving that component new plugs. Wrapping is actually a form of component plugging. Looking again at Figure 6, the B/C component is adapted by wrapping it with the A component.

The key to wrapping is to recognise that precision in specification is not the same as rigidity. Indeed, one of the major advantages of precise specification is that it enables adaptation of both the API and semantics of a component without fear of unknown (i.e. unspecified) side-effects. The idea is that we take precise specifications and manipulate them to exactly match our own requirements.

Example

Figure 9 uses the same Catalysis notation to show how the resource allocation component can be specialised. The dotted arrows between the models stipulate how elements of the parent are mapped into elements of the child. The mapping determines that resources are instructors, the requirements are courses, and an instructor meeting a requirement corresponds to the instructor being qualified for a course. Not all the mappings have been shown.

The child will obtain all elements of the parent, appropriately mapped, together with all specified behaviours, constraints etc. The child may, of course, add additional behaviour.

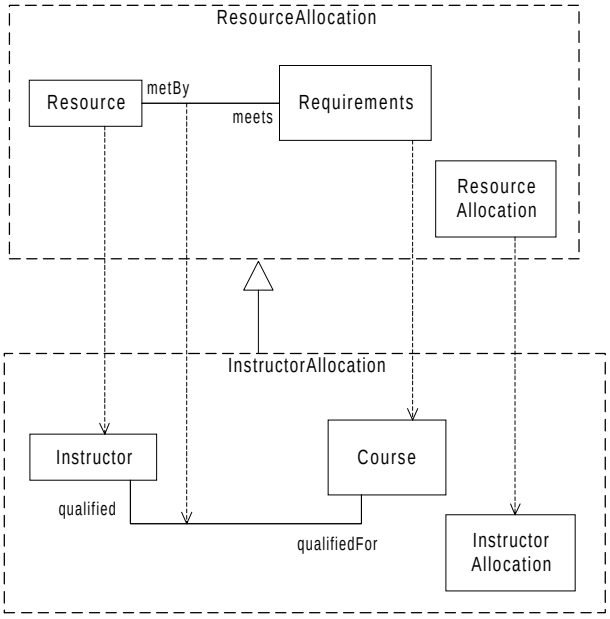

Figure 9: Specialising Resource Allocation

Further elements (e.g. invariants, new classes, new operations, strengthened operation contracts, etc.) may be added in the child component model provided behavioural conformance of child to parent is maintained.

3.5. Components as "model networks"

We mentioned earlier that it would be desirable for the behavioural model of a component to be continuous. A discontinuous model is one where some form of translation is required to go from one part of the model to another. For example, a model in which the component plugs are specified using an object-oriented techniques, say, yet the rest of the model is described using a relational database would be regarded as discontinuous: there is a significant translation step from one part of the model into the other. Removing discontinuities makes it much easier to describe behaviour and permits a more flexible and finer-grained regime for defining overlapping views on the model, as required to define plugs and revealed behaviour.

To handle discontinuities we propose instead to allow components to be characterised as a network of related models (Figure 10). Models in the network are related by projection mappings. A projection mapping is essentially a mapping between the interfaces (the edges of the plugs) of the two models: the interface of the more abstract model is a projection of the interface of the more concrete one. A model is more concrete if the granularity of the interface is finer than its more abstract counterpart. For example, a model for a rental component with an

interface that had a single operation abstracting the whole cycle of renting some item, would be more abstract that the a model which broke down that cycle into reserve, checkOut, checkIn, etc. operations. The projection mapping would have to show how various combinations of the more concrete operations mapped to the single rent operation.

Models may also be constructed from different technologies (e.g. UML versus a relational database model), in which case the projection mapping maps between those technologies.

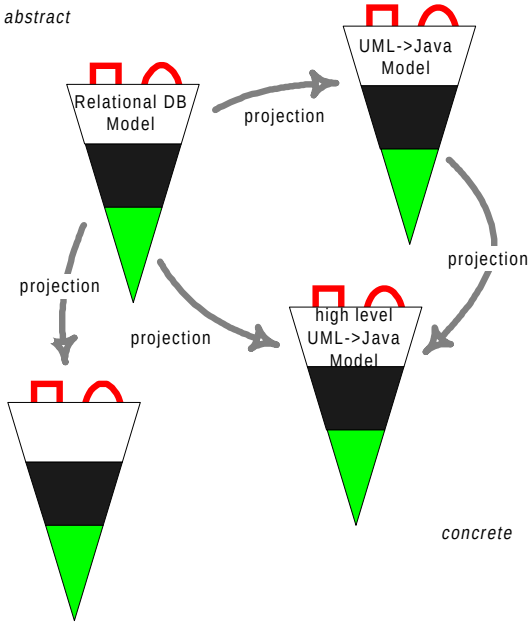

Figure 10: Components as model networks

4. Improving Modelling Techniques

Before relating the techniques and concepts described back to the example of Section 2, we identify some advances in PVM that still need to be made to support CBSD.

4.1. Describing Frameworks

We have already said that a framework, as understood in the OO literature, may be thought of as a component in our reference model. The latter has already distinguished between plugs, which detail how they may be used from without, and the revealed behaviour section detailing how the component should be specialised.

In our examples, we carefully avoided distinguishing plugs from revealed behaviour. This is because there is currently little support for managing this distinction. For example, the UML concept of packages is inadequate here, as it does not permit any overlapping. There is also no clear way of distinguishing between operations invoked from components outside the framework, and operations there solely for use within the framework. In our own modelling we have attempted to make this distinction using UML's *public* and *protected* declarations, but this is not ideal.

There are also few facilities for assisting with description of the revealed behaviour section. In particular, UML provides no way of highlighting "plug-points". For example, rather than reveal all the code for a white box component, all that needs to be revealed is a specification of a supporting method which encapsulates the single piece of special code required. Then the component would only list those operations, with their specifications, for which code needs to be written, possibly with a high level diagrammatic design indicating how those operations fit into the “big picture”.

4.2. Outerfaces

Another form of plug point is an outerface: something that identifies what components require from other components. For example, in Figure 7, the class diagram fragment shown for InstructorAllocation is an outerface or plugpoint. When implementing InstructorAllocation it will be assumed that the functionality represented by this outerface will be implemented by plugging in another component. UML provides no facilities for highlighting outerfaces.

D’Souza and Wills (1998) introduce the notion of *ports* as a way of providing this information. They introduce two genders of port: input ports (which include interfaces as described here) and output ports, which essentially replace associations. Connections are made by linking compatible ports of opposite gender. It may be that these ideas will (a) provide much finer control over connecting component objects at runtime, and (b) improve the description of frameworks, in the sense that it will be much clearer how components are intended to work together.

4.3. Quality

The documentation techniques clearly improve the quality of descriptions at the specification level, and, for white box components, at the design and implementation levels. One aspect that has not been addressed is the quality of components, in as far as the delivered design/ implementation or executable meets the “contract” with the user, as laid down by the precise specifications.

The approaches that could be taken to resolve this include, but are not limited to:

- Delivery of a test harness with a component, so that the user can run tests herself.

- A collection of pre-fabricated tests to run through the test harness, with supporting documentation explaining why these particular tests have been chosen.
- A certificate indicating the level to which the correctness of the design and implementation against the specification has been established.

There are refinement techniques available that can increase confidence of correctness up to "proven mathematically" (see e.g. Woodcock, 1991; D'Souza and Wills, 1998). It is likely that these techniques can be used to prove the correctness of certain refinement patterns, that, if followed, will guarantee correctness of the code against the specification.

In some safety-critical industries certification processes have been established, and perhaps these could be used as a model. However, a note of warning: these are usually of the form "formal methods must have been used", not "these specific techniques must have been used following these patterns".

It might be possible to develop a system of dynamic certification, similar to the certification system now being used for transferring information over the Web and implemented in popular Web software such as Netscape Communicator.

5. Enabling CBD

The techniques described have been motivated by a need for precise, accessible, and automatable models of components, which was in turn motivated by raising questions about how the activities of building, finding, adapting and (re)using components could be supported. These questions were loosely based on an evolving scenario set out via an example in Section 2. In this subsection we conduct a more detailed analysis of how the specific techniques described could help enable CBD as envisaged in that example:

Using components to recursively subdivide the development

Precise descriptions of the behaviour supported by components provided by each development team, and by each member of the development team, will help to clearly identify the boundaries of responsibilities and where there might be overlap. With appropriate CASE tool support, component assembly may be prototyped at the specification level using framework assembly techniques, and, because of the detailed and precise behavioural specifications, conflicts and interactions may be identified and dealt with early in the development process rather than be discovered only when coding begins.

Reduce maintenance costs

By using framework components, the application has built in flexibility for future adaptation through specialisation, in particular by providing points at which specialised sub-components may be attached (plug-points). Dynamic instantiation of plug points would be possible, where a framework is constructed so that some of these plug points could accept black box components.

Reuse existing components in the development project

A precise description of the behaviour of components in catalogues helps the development team considerably cut down the choice of potential components which match the precise and detailed specification of what they require. If all they had to go on was a list of operations, then the search would be entirely dependent on key word searches on names of operations and on reading the informal description that may accompany the component.

Precise, detailed descriptions could also mean that some sophisticated software agents could automate much of the searching and matching process, e.g. by matching on patterns in the diagrams or using automated reasoning techniques.

The certificate that comes with the component, backed up by trusted refinement and integrity checking, means that when the component is installed, it really does work as claimed.

Substitute cheaper components

Enabled by the same catalogue browsing and selection techniques detailed above.

Interface to legacy and external systems

Precise and detailed documentation allows much of the investigative work to be done at the specification level, avoiding the expense of building a complete system based on intuition and lucky guesses, only to find that it doesn't work. The sophisticated CASE tool support enabling integrity checks and simulation/animation of the specification gives the team a high level of confidence in the results of their investigation and allows experimentation with alternative solutions. All this can be done before spending money on buying in an expensive package.

Rapid application assembly

The effort put into specifying the components in terms familiar to the business domain expert, and the repository of specific models showing typical behaviour, allows the developer and business user to quickly ascertain whether the components match their requirements. Snapshots and filmstrips in this repository

come packaged with a visualisation scheme which plugs into the assembly tool allowing the examples to be viewed through a mock up user interface consisting of forms, fields etc. This scheme is also used as the basis for a test harness which allows the components to be experimented with.

The components actually come as part of a whole framework with clearly identified plug points, so the developer knows immediately which parts need to be adapted and which do not. She is also given considerable guidance into how to perform the adaptations. The framework is delivered with a number of different implementations, each of which is designed to work with different UI frameworks with implementations on different platforms. The developer selects the implementation and UI framework that suits her particular circumstance. The assembly tool depends heavily on the precise specifications of the components, and its built-in support for framework composition means that the developer's adaptations of the business and UI frameworks may be assembled quite rapidly to produce the desired application.

Unfortunately the certification process for components means that the developer may only submit an addendum to the component catalogue to indicate that her adaptations exist, but have not been certified or assessed for their reuse potential. This is noted by the catalogue maintainers who identify some potential in her extensions and contact her with a view to initiating the certification process.

Business user smart assembly

The components available in the catalogue are all black box and not intended to be specialised. Their specifications have been carefully crafted to be phrased in business terms and come with a substantial specific model repository to illustrate examples of use. This means the business user can be quite confident in his choice of components.

The desktop spreadsheet program has been designed to allow components from this particular catalogue to be plugged in dynamically as add-ons. They come with an electronic certificate guaranteeing this, so the assembly tool is able to validate the proposed extension by checking the certificates and matching the interfaces of these components with the ones that the spreadsheet program indicates it can accept. The certification scheme is supported by a rigorous process based on precise and detailed descriptions of the interfaces and component implementations.

Interestingly, the last point corresponds most closely to what is feasible today, mainly because no specialisation is required. This means use of black box components only, which are supported quite well by current implementation technologies, and require little expertise (for man or tools) to assemble, provided the component is certified to work with the proposed application. Examples are plug-ins in Web browsers such as Netscape's Navigator, connection of applications via OLE in Windows (e.g. the use of an external drawing editor or spreadsheet on objects in a document as it is being processed by a word processor), etc.

6. CASE Tools: Are they up to the job?

In Section 2, four basic processes for CBSD were identified: build, find, adapt, (re)use. For CBSD to be successful considerable and sophisticated tool support is required. Many CASE tool vendors (particularly OO CASE tool vendors such as Rational and Select) are repackaging their modelling and design tools as CBD tools.

However, the extensions to support CBD seem to be focused on providing management tools to assist with archiving and retrieval of components, i.e. the find process.

In the previous section we argued that the quality, precision and detail of component descriptions needs to be significantly increased to support *all* CBD activities. From the analysis conducted in Section 4.6. it is quite clear that some scenarios will be pure fiction without sophisticated CASE tool support for these descriptions. Thus the kinds of extensions currently proposed by commercial CASE tool vendors will only provide limited support for CBD.

The real technical challenge to be faced is the provision of CASE tool support required to enable the step increase in the quality of components and component specifications envisaged. Here are some examples of what is needed:

- In current tools, cross checking between models is poor. For example, checking a sequence diagram against a state diagram. Yet in some circumstances they are far too constraining, e.g. allowing only one sequence diagram per operation.
- Facilities for documenting specific models, snapshots, filmstrips etc. is extremely limited. One tool we've used doesn't even allow UML instance diagrams to be drawn. We would like a repository of specific models partitioned appropriately (e.g. filmstrips for a particular sequence diagram). We would also like to store counter examples, to show what behaviour is not allowed. There should be facilities for checking the integrity of instance diagrams and filmstrips against

the model, for example that links between objects obey cardinality annotations on the class diagrams.

- Connection to the code is very loose. You can generate outline code, but the process is really only one way. For example, if I change and/or add to the code, the changes are lost when the model is altered and the code is regenerated.

  More generally, it would be better to move to a situation where the CASE tool supported multiple levels of description as suggested by the "component as network" model in Section 3. Such a tool would need to support projection mappings, including an ability to maintain integrity between the different levels. Some CASE tools (though not UML compliant) can do this to some extent: the Bridegpoint tool from Project Technologies implements the Schlaer-Mellor method and generates all code from the model like a high level compiler; EiffelCASE from ISE implements Eiffel/BON, though this is largely achieved by making the projection mappings as close to identity as possible; COOL:Gen from Sterling (formerly Composer from TI), which implements IEF, supports two different levels of modelling, specification and design, as well as 100% code generation. Rational Rose claims to support round trip engineering, though this claim does not seem to stand up to close scrutiny (O'Brien, 1997) and certainly not for cases where the visual model is a high level abstraction of the code.

  Having a more systematic, rigorous and, in some cases, proven connection between different levels of modelling is essential for underpinning a certification system as discussed earlier.

- Animation or execution of *all* models. The execution could be visualised in terms of the snapshots and filmstrips, or visualisations of those. This is an extension of the idea of a specific model repository: animation effectively allows specific models to be generated dynamically.

- Better support for grey box components, including the ability to allow multiple, possibly overlapping views of the same model.

- Support for matching components and checking that one component conforms to the behaviour of another. This will at least require advanced pattern matching techniques on the various diagrammatic notations employed, and, to guarantee complete behavioural conformance, probably sophisticated model checking and automated reasoning techniques (e.g., Roscoe et al., 1996; Martin et al. 1994; Jackson, 1994).

- To assist with management and delivery of components, the documentation must be supplied with any executables as a self-describing, self-unpacking package. Indeed it may be necessary to supply two forms of a component: one for use when assembling components into a system, which would have all the documentation as described; a second delivering the executables and plugs required (e.g. that which is currently supplied in a COM component) for the component to be used as part of a running application.

7. Conclusions

The paper has made two contributions. It has described a reference model for CBSD, and illustrated this with examples of precise, visual models. It has shown how precise visual modelling techniques can be usefully employed in the description of components.

Future work is likely to focus on adding more substance to these ideas. This is likely to involve:

- Further work on defining an integrated notation set based on a subset of UML tailored to the specific needs of CBSD.
- Work on implementing a prototype CASE tool taking account some of the observations made in Section 6.
- Refinement of the CBSD reference model, including the construction of a mathematical model.
- The application of the techniques to modelling real systems, in order to provide some evaluation of what we are proposing and to inform the ongoing development of ideas. Currently we are collaborating with two companies, both concerned with the migration of legacy enterprise systems to component-based ones.

8. Acknowledgements

The authors acknowledge support of the UK EPSRC grant nos. GR/K67304, GR/M02606.

9. References

Bosch J. (1997) Adapting object-oriented components. In 2nd International Workshop on Component-Oriented Programming, pp. 13-21.

Boundy D. (1991) A taxonomy of programmers. Software Engineering Notes, 16(4):23-30, October.

Cook S. and Daniels J. (1994) Designing Object Systems: Object-Oriented Modelling with Syntropy, Prentice Hall.

D'Souza D. and Wills A. (1998) Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, to appear 1998, draft and other related

material available at http://www.trireme.com/catalysis.

Fowler M. with Scott K. (1997) UML Distilled, Addison-Wesley.

Gil Y. and Kent S. (1998) Three Dimensional Software Modelling, to appear in Procs. ICSE98.

Jackson D. (1994) Abstract Model Checking of Infinite Specifications, Proceedings of Formal Methods Europe (FME'94), LNCS 873.

Jones C. (1991) Systematic Software Development with VDM, Prentice Hall.

Kent S. (1997) Constraint Diagrams: Visualising Invariants in Object-Oriented Models, in Procs. OOPSLA97, ACM Press.

Lauder A. and Kent S. (1998) Precise Visual Specification of Design Patterns, to appear in Procs. ECOOP98, Springer Verlag.

Martin A., Gardiner P.H.B. and Woodcock J.C.P. (1996) Tactic semantics and reasoning, FACS, 8(4), pp479–489.

O'Brien L. (1997) Review of Rational Rose 4.0 for C++, Software Development Magazine, June.

Pree W. (1995) Design Patterns for Object-Oriented Software Development, Addison-Wesley.

Reenskaug T. with Wold P. and Lehne O. (1996) Working with Objects, Prentice Hall.

Roscoe A.W., Woodcock J.C.P. and Wulf L. (1996) Non-interference through determinism, Journal of Computer Security 4 (1996), pp. 27–53, IOS Press.

Shaw M. and Garlan D. (1996) Software Architecture: Perspectives On an Emerging Discipline, Prentice-Hall.

Short K. (1997) Component Based Development and Object Modeling, available from http://www.cool.sterling.com/cbd.

UML Consortium (1997) The Unified Modeling Language Version 1.1, available from http://www.rational.com.

Woodcock J.C.P. (1991) The refinement calculus, Procs. VDM Symposium 91, Delft, The Netherlands, Springer-Verlag LNCS 552.

Woodcock J.C.P. & Davies J., Using Z—specification, refinement & proof, Prentice Hall, 1996.