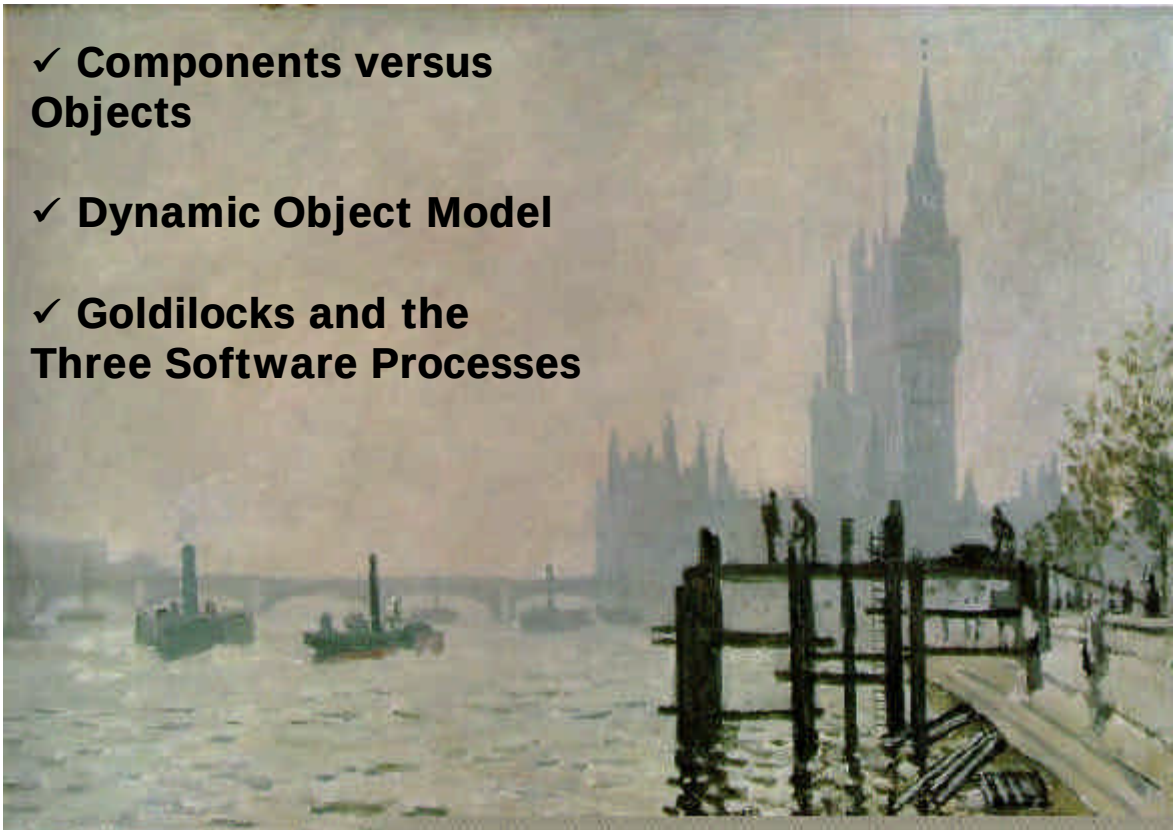
ObjectiveView

Objects, Components and eBusiness Development for Software Professionals



Claude Monet 1840-1926 "The Thames at Westminster"

Plus:

The RSI Approach To Use Case Analysis

Object Design Issues

e-Business Development

Published by

RATIO

OO consultancy – training – tools – recruitment

see www.ratio.co.uk for back copies

Got questions about an article?
Join the ObjectiveView discussion group – email:
objectiveview-subscribe@egroups.com

Further details on page 2

ObjectiveView

The Object and Component Journal for Software Professionals

CONTENTS

Personal Views

Interview with Ivar Jacoson
by Adriano Comai **30**

Object/Component Architecture

Components versus Object
by Clemens Szyperski **8**

Requirements Engineering

The RSI Approach to Use Case Analysis
by Mark Collins-Cope **17**

e-Business Development

Building e-Business Solutions
by Keiron McCammon **24**

Object Design Issues

Dynamic Object Model
by Ralph Johnson **31**

Software Process

Goldilocks and the Three Software Processes **36**
by Doug Rosenberg & Kendall Scott

IN THE NEXT ISSUE OF OV...

- ✓ **Interview with Bertrand Meyer**
- ✓ **John Daniels: *"UML Components"***
- ✓ **Alistair Cockburn: *"Use Case Analysis"***
- ✓ **And lots more…**

CONTACTS

Editor
Mark Collins-Cope
markcc@ratio.co.uk

Production editor
Karen Ouellette
karen@ratio.co.uk

Free subscription
email delivery:
objective.view@ratio.co.uk
(subject: subscribe)

hardcopy delivery:
objective.view.hardcopy@ratio.co.uk
(include full contact details)

Feedback / Comments / Article Submission
objective.view.editorial@ratio.co.uk
or join objectiveview@egroups.com

Circulation / Sponsorship Enquiries
objective.view.editorial@ratio.co.uk

We'd like to invite all ObjectiveView readers to join the recently introduced objectiveview discussion group @ egroups.com.
This discussion forum was created as a tool to encourage communication between ObjectiveView readers and authors, as well as between readers themselves.

Feel free to the email the list with questions about articles, as well as about object and component technical issues of general interest.

JOIN NOW! 2 EASY WAYS:

Go to http://www.egroups.com/group/objectiveview and click on the 'subscribe' button

Personal Views Series

Interview with Ivar Jacobson

Adriano Comai interviews Ivar Jacobson

[Adriano Comai]: Mr. Jacobson, you are widely known as the inventor of the "Use Case" concept. How was this concept born? Which were the forces that brought you to work on this idea?

[Ivar Jacobson]: It evolved over many years. First of all, I worked in telecommunications in my early days, where exists the concept of "traffic cases". The traffic case was like a use case, it was, in fact, a telephone call. There were many different kinds of telephone calls, many different kinds of traffic cases. That was something I learned there. In that time we had no cases for other things than telephone calls. In that early time, I mean back in 1967, I had another term that was similar to use case, and meant the same thing: we called them "functions". A function crossed the whole system. A telephone call was a function, but functions were also more abstract things, and the term was not really well defined. We used this approach, "function-driven development", that is called now "use-case-driven development". We identified the functions, and then we designed the functions, like we do with use cases today. So these ideas are very old.

I identified the use case concept in 1986, and when I had found that concept I knew I found something that solved many problems for me, because I could use this concept for everything that systems did, and for every kind of system. It helped me a lot to create a systematic methodology.

[Adriano]: The Use Case concept is like a filter that distinguishes between functions related to the user and functions internal to the system…

[Ivar]: Yes, it discriminates lots of functions that can not be use cases. It's much more specific. A function could be anything, that's the problem. Use cases cannot be anything.

[Adriano]: What are the roots, the ancestors to the UC concept in the software engineering literature?

[Ivar]: I don't know. Actually, I don't have anything like that. I think the closest thing was this idea of traffic cases.

But I want to make a point. It may be the truth that I am most known for the use cases, but we had component-based development in 1967, and use cases were not there, so component-based development is something I've been working in my whole life. The other thing is architecture, I mean really to identify an architecture before doing everything else. We talked about software architecture in 1968. We presented the software architecture when we went out to our customers, and I remember they had never heard about anything like that. They taught about architecture for hardware, but there was not an architecture for software.

[Adriano]: The use case concept seems, today, almost obvious, common sense...

[Ivar]: And I think it is.

[Adriano]: Yet it was marvellous to see how quickly and broadly it was accepted by other methodologists. How could this happen with so few resistance?

[Ivar]: Most of the methodologists went into the objects world, and there was a lot of competition. However, the use case didn't compete with anything, and it solved a problem that everyone had. Even the concept of scenario was about something internal to the system, about internal interactions, but was not really specified.

One thing I did late was to publish. If you look upon my 1987 paper for OOPSLA, there I had all these things, but the problem I had was that I could sell my book, the Objectory book, in 1990-1991, for $25,000 a copy, so why should I go and give it away to Addison-Wesley or any other publisher, to then get $3 a copy, even if that was selling many more? Now I understand that I should have done it a little bit differently, but it's very hard to say, you have to be at the right time. So I think the other books helped us, because they had a big problem, that was how to get from

requirements to start an OO analysis and design. But I refuse the idea that just the fact that people publish their book, they were first with the idea: they were older ideas.

[Adriano]: Use case specifications are mainly textual (although they can be complemented with UML diagrams). Previous methods (as Structured Analysis, or Information Engineering) proposed the use of diagrams as a "common language" to reach an agreement between customers ("users") and developers. What's behind this resurgence of the role of text?

[Ivar]: In reality, today, customers of software don't want to read diagrams. Use case diagrams are so intuitive that everyone can read them. Text is something people don't need to learn a special language to use.

We can use activity diagrams to describe use cases, and it's very nice, but there are two problems with that. First, they become very quickly very detailed, and it's not sure that they are more understandable because they are detailed, even if there is no doubt that at some point you need to specify in more detail. But we think that the best way to do that is in terms of the analysis model, where you describe every use case as a collaboration among objects, instead of trying to detail the use case without talking about objects. You can use activity diagrams, but activities can be very abstract, so it's very hard to understand them, you really need to understand what has to be done, to understand the activities. So I'm very careful in introducing a formalism in use cases. I think that when you go to analysis you get a much better formalism, a much better language to express details, because you talk about objects.

[Adriano]: Maybe activity diagrams cannot convey so much information as text …

[Ivar]: Yes, it's just a pragmatic thing, it's not a holy cow. In some cases it is maybe good to use activity diagrams, but I think I want to have a warning there, because it's better to be detailed when you have the right language to express details. And I think that in analysis when we talk about objects, and about collaborations between objects, even if these objects are conceptual, and not physical, implementation things, they are much more concrete and much easier to understand than just activities.

[Adriano]: What about the ambiguity of natural language?

[Ivar]: Yes, it is ambiguous, but I think there is a trade-off. … Language is understandable, it's ok to use just English. On the other hand, in situations when we have hard use cases, with a lot of interactions, you may need to go further. But it's better to view the analysis model as part of the requirements. In the new Unified Process book I've taken a little step in that direction, I view analysis as a part of requirements, and one of the things we get from analysis is the structure that we would like to see in the design and in the implementation, so we have some requirements on the architecture that we create through analysis.

[Adriano]: Use cases have a double role in your method. First, they are used to discover and to validate requirements coming from customers and users. Then, they drive the whole system development. Is one of these roles more important than the other?

[Ivar]: No, of course not. But many methodologists and many software developers are very technology-introvert. If the use case concept wasn't so good in describing interactions, and helping to define collaborations, they wouldn't have bought into it. So it does work as a very good sales argument to software developers: they would never have been accepted as widely as they are, if they hadn't this impact on the design, if they didn't drive the development. For me, anyway, both aspects are equally important, it's a very nice way to find the requirements, and to capture requirements in some kind of diagram, without going into the internals of the system. They are used to capture and to identify scenarios, and describe relationships between these things.

[Adriano]: In your book, you speak about feature list of requirements as a starting point to derive use cases.

[Ivar]: The feature list is something that will be translated to use cases, and the documentation will describe the use cases, so the feature list will grow and shrink, as you translate the requested features into use cases.

[Adriano]: Some years ago, you applied the use case concept, and other Objectory ideas, to the business process reengineering area. How well has been your proposal accepted by non-IT people? Are use cases used in business engineering so much as in the IT area?

[Ivar]: No, they are not, for several reasons. Rational has selected to work primarily in software, even if we understand completely the importance to do business engineering. However, we also know that the tools people need for business engineering are easily described in terms of tools for software engineering. If you have

Rose for visual modeling, you can extend it, to make it work for business engineering as well, but not the other way round. We still need to have a good tool for visual modeling of software. We have been working on Business Engineering, but we have done it locally, in Scandinavia, and we have in Sweden a Service Package from Rational, called Rational Business Engineering, with a specialization of Rose and detailed process descriptions.

Anyway, the customer base for software engineering is much larger. People who want to do business modeling are typically people who understand software, and who understand they need more. It's very sad that people from business engineering, like Hammer, didn't think about modeling so much, so the stream of people that come from that part is much fewer, most people come to business modeling from the software world. It's a much smaller business, but we have customers with hundreds of licences of Rose for Business Engineering, for example in one telecom company and in the Swedish pension system.

[Adriano]: Did you have any contact with people like Hammer or Champy about your business modeling proposal?

[Ivar]: No, I read their books, of course, and, we have been doing business engineering for many years, but when I read the books I said: "oh, here we have a guy who presents a problem, and he gives a sketch, an outline of the solution, but he cannot model it, and if you don't model it you don't understand it". Anyway, I feel that Hammer work and our work were very tightly related.

Another important idea is one-to-one marketing. One of the people that worked for me at Objectory is now working on it, and she thinks that our approach is perfect for it. This is an area in which I'm thinking to do more work in the future. I always work in the long term, on what happens five years from now and so on, and there are two areas in which I will work in the near future, one is business engineering, and how that is impacted by the new world, the internet world, and the other thing is software development in the context of the web, applications for the web. Even if the web changes everything, and it changes basically everything we do in business, the way we develop software for the web doesn't change very much. It's basically the same thing, but there is one thing that is different, and that's the user interface. The user interface design is very important.

[Adriano]: I saw you quote from Larry Constantine in your last book about this issue. Do you like his approach?

[Ivar]: Very much. His last book is a very good one. The only problem I have with it is that, instead of using the UML, he uses his own notation which is much weaker, not so well defined, and he has a different approach to what a model is, but there are lots of good things in that book. I like it a lot, it's the best book I've seen in 3 years in software.

[Adriano]: Is it more difficult to persuade IT- or non IT-people of the importance to do business modeling as a starting point for a new project or for the evolution of an existing system?

[Ivar]: The problem is that we don't have the time. Time-to-market is today… it's more important that you get something out than that it's a good thing, and that means that these approaches must be very tightly integrated. IT people know that to do business models takes 6 months, 12 months, and when they start to build the software, the business has changed. What's unique about our approach is that business modeling is part of the process, so if you have a software that takes 6 months to develop, than you do business engineering for 6 months. I can understand that people hesitate to do business modeling: if we think quality is not so important in order to get it out, then we will always have problems with any structured approach to develop software. But with iterative development we get something out according to the plan, and I think that will help people to understand the need for business modeling, continuously, during all the time.

[Adriano]: The UML was a collective creation. And so the Unified Process. But in the latter, your own contribution is clearer, more apparent. UP roots are more in the Objectory / OOSE ground than in the Booch method or in OMT. Does this reflect a sort of "division of labor" among the Amigos?

[Ivar]: I don't think that we have divided on purpose. Some people are experts on everything, and it's hard to see that anyone of us three would agree that there is an area in which we don't have any expertise. Honestly, I think there is no division of work. It's a fact that we started from Objectory, when developing the Rational Unified Process, and from there we have evolved. And of course, you cannot move from object modeling, just object modeling. There is not a simple way to go from approaches like OMT, or Booch, to do what we did in Objectory. So it is easier to go the other way round, thinking about use cases and then you have objects and classes and subsystems to design.

[Adriano]: Booch and Rumbaugh moved from software, while you moved from customers and users…

[Ivar]: Yes, but there is also another aspect. Components are what we have to start with. I actually started with components. In 1967, when I was introducing this approach at Ericsson, the main objection I had from developers was that these components, that we developed, were not easily related to the "functions", or the "use cases". If you take the use case, the use case crosses many components: that was an objection. They were thinking in terms of "one function, one module".

Whereas I was saying, well, that's not ok. Most of one function, or one use case, will be implemented by one component; but then the other components will play a role in that use case. So that was one of the objections. And I said: that's exactly this objection that I will turn into something positive: this is exactly what you need, you need to have that complexity, because that's how it is. So the outside world talks about use cases, but inside use cases cross these components - subsystems.

Just having objects and components, and don't care about things that cross them, is a smaller problem. One of the difficult things is to make use case realizations, and to manage dependencies between subsystems, and that's much harder. Thinking just upon objects is a much smaller problem, it's a sub-problem.

No, I don't think there is any conscious decision on dividing work - we think that the UML was a big task, and we had to work together to get it done. Now we are working on different things; we just don't think it's meaningful to work together on everything.

[Adriano]: Do you expect for the Unified Process a success and an impact on the IT industry, analogous to that of the UML?

[Ivar]: Yes, absolutely yes, and we have very good reasons to believe that. We are making inroads into many corporations today, and it's our goal to get there. We don't think it would be an easy thing to make the Unified Process a standard, it would be so much hard work and so much opposition, so we'd rather do it in small steps. Instead of going and forcing people through a standard, let people convince themselves. And I think that everyone that looks at the Rational Unified Process will become convinced this is the way they've got to do it, as soon as they have started to look at it. There is nothing even close to it. Many people tried to say that there is, but what is that they have? They have something that can be compared with my book, but they don't have anything that can be compared with our process. If you just look upon it in terms of substance, and depth, and experience, and so on, and if you compare … How old is Approach A, or Approach B? Do we know that it works, for large projects? We know that our works. It's really very different.

[Adriano]: How much of Objectory is left in the Unified Process?

[Ivar]: If you look just upon the basic ideas, we basically only covered requirements, analysis and design in Objectory. If you look upon these things, what was in Objectory in the old days is still there. But there's a lot of new stuff that has been added. We had very little about implementation, very little about testing, nothing about configuration management and version control, nothing about project management. Iterative development was primarily something we recommended, but it was not enforced by the process, we didn't really tell about the differences among the various iterations, so I think the core ideas are still there, but there are lots of other things that have been added. The Rational Unified Process is really a teamwork, we have a lot of people that have been involved. Whereas the Objectory Process was primarily my ideas, my work that we implemented. But, given the smaller resources we had, it was quit a lot.
[Adriano]: You present every iteration like a mini-waterfall …

[Ivar]: Yes, we think of it as a mini-waterfall, but we have a lot of parallelism. Within the waterfall, the people who develop subsystems work concurrently on their subsystems. So people who work on use cases rather independently talk to one another so they don't invent new things and so they reuse the same components, but they work concurrently on subsystems during an iteration. But that is still waterfall, because you always start with requirements, then you go through analysis, and then through design activities.

[Adriano]: You come from Sweden. Do you feel there is a European specificity in system and software engineering? Maybe more concern, more care about organizational issues than in the US?

[Ivar]: I have not been able to find any systematic difference, because I found people in the US very interested in starting with the business, in understanding the business, before they develop the software, and in Europe too. There is no systematic difference. It would be more funny… There has been a lot of research, in Europe, in areas that are more at an abstract level, and less in the concrete, physical world, but I must say that a

lot of research has been done, and with useful results, both in the US and in Europe - and also a lot of work done that never led to anything.

[Adriano]: Now the greatest part of the "unifying" effort is done. Are you going to rest, and capitalize on it, or are you moving forward to other areas of interest? What next?

[Ivar]: There is one part of me that says: I want to go ahead, and look for what needs to be done, to create a much better world, and we have a lot of things to do. In a way, UML is a standard, and that's wonderful. But it doesn't mean that these are new ideas, we just got them consolidated. In the last years, I don't think I've done anything really new, I pushed the adoption process more than the creation process. Now a part of me wants to take a next step. What is beyond the Unified Process? What is beyond UML? I think it is still an evolution, not a revolution, but there are some important steps that need to happen in software, to get up to the level of extremely high quality which we need to develop the systems we will want to develop in the 2020, or something like that. These are much larger systems than we can think of today, and more complex, and we need to be able to develop these systems. We need a much better infrastructure than we have today, in terms of operating systems, programming languages, UML integrated with programming languages, maybe part of the UML will be a programming language, with action language semantics and so on. That's one thing I'm constantly thinking of.

The other thing is to capitalize on business engineering. There is something really interesting to get done, there. The Rational Unified Process is very well prepared for the Web. Many of the companies who develop websites are using the Rational Unified Process today, specializing it a little bit, so it fits for their special purposes, but it's the same process. I would like to see that we extend and make the right decisions to make the required model improvements, in the Rational Unified Process, changes that make it clearly, without any doubt, "the" process for web sites applications design.

I'm also going to write a revision of my book "The Object Advantage" for the end of this year. The Internet, and ideas like one-to-one marketing, will have a lot of impact on this revision. We need to make the book more approachable for business people, and not only for software people. We will show how to use it in the context of business, not only in the context of software. The basic ideas are already there, it works very well, customers are happy, but today we need to take that through the barrier of IT, solving the problem existing with the acceptance of technical notation. Activity diagrams are very useful for business modeling.

Adriano Comai is an Italian methodologist. This interview is also published at http://www.analisi-disegno.com and in the October 1999 issue of the Italian magazine ZeroUno.

Object/Component Architecture Series

Components versus Objects

Clemens Szyperski, author of 'Component Software', discusses the similarities and differences of objects and components…

Introduction

Components are on the upswing, while objects have been around for a while. It's understandable but not helpful to see object-oriented programming sold in new clothes by simply calling objects "components." The emerging component-based approaches and tools combine objects and components in ways that show they are separate concepts. In this article, I will examine some key differences between objects and components to clarify these muddy waters. In particular, you'll see that approaches based on visual assembly tools really assemble objects, not components, but create components when saving the finished assembly.

Why Components?

What is the rationale behind component software? Or rather, what is it that components *should be*? Traditionally, closed solutions with proprietary interfaces addressed most customers' needs. Heavyweights such as operating systems and database engines are among the few examples of components that have reached high levels of maturity. Large software systems manufacturers often configure delivered solutions by combining modules in a client-specific way. However, the interfaces between such modules tend to be proprietary—at most, open to highly specialized independent software vendors (ISVs) that specifically produce further modules for such systems. In many cases, these modules are fused together during a linking step and are no longer distinguishable in deployed solutions.

Attempts to create low-level connection standards or wiring standards are either product- or standard-driven. The Microsoft standards, resting on COM and now the .NET Framework common language runtime (CLR), have always been product-driven and are thus incremental, evolutionary, and to a degree legacy-laden by nature.

Standard-driven approaches usually originate in industry consortia. The prime example here is the Object Management Group (OMG)'s effort. However, OMG hasn't contributed much in the component world and is now falling back on JavaSoft's Enterprise JavaBeans standards for components, although attempting a CORBA Beans generalization: the CORBA Component Model (CCM). The JavaBeans standard still has a way to go; so far it is not implementation language-neutral and bridging standards to Java external services and components are only emerging.

At first, it might surprise you that component software is largely pushed by desktop- and Internet-based solutions. On second thought, this should not surprise you at all. Component software is a complex technology to master—and viable, component-based solutions will only evolve if the benefits are clear. Traditional enterprise computing has many benefits, but they all depend on enterprises that are willing to evolve substantially.

In the desktop and Internet worlds, the situation is different. Centralized control over what information is processed when and where is not an option in these worlds. Instead, contents (such as web pages or documents) arrive at a user's machine and need to be processed there and then. With a rapidly exploding variety of content types—and open coding standards such as XML—monolithic applications have long reached their limits. Beyond the flexibility of component software is its capability to dynamically grow to address changing needs.

What a Component Is and Is Not

The separate existence and mobility of components, as shown by Java applets or ActiveX components, can make components look similar to objects. Indeed, people often use the words "component" and "object" interchangeably. Objects are said to be instances of classes or clones of prototype objects. Objects and components both make their services available through interfaces. Language designers add more confusion by discussing namespaces, modules, packages, and so on. I will try to unfold, explain, and justify these terms. Next, I'll browse the key terms with brief explanations, relating them to each other. Based on this, I'll look at a refined component definition. Finally, I'll shed some light on the fine line between component-based programming and component assembly.

Terms and Concepts

Components. A component's characteristic properties are that it is a unit of independent deployment; it is a unit of third-party composition; and it has no observable state.

These properties have several implications. For a component to be independently deployable, it needs to be separated from its environment and from other components. A component therefore encapsulates its constituent features. Also, since a component is a unit of deployment, you never partially deploy it.

If a third party needs to compose a component with other components, the component must be self-contained. (A third party is one that you cannot expect to access the construction details of all the components involved.) Also, the component needs to come with clear specifications of what it provides and what it requires. In other words, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces and platform assumptions only. It's also generally useful to minimize hard-wired dependencies in favor of externally configurable providers.

Finally, you cannot distinguish a component without any observable state from copies of its own. (State that isn't observable, such as serial numbers used for accounting or caches, is permissible.) A component can be loaded into and activated in a particular system. However, in any given process, there will be at most one copy of a particular component—multiple copies would not provide any additional value. So, while it is useful to ask whether a particular component is available or not, you don't need to ask about the number of copies of that component. (Note that a component may simultaneously exist in different versions. However, these are not copies of a component, but rather related components.)

In many current approaches, components are heavyweights. For example, a database server could be a component. If there is only one database maintained by this class of server, then it is easy to confuse the instance with the concept. For example, you might see the database server together with the database as a component with persistent state. According to the definition described previously, this instance of the database concept is not a component. Instead, the static database server program is and it supports a single instance: the database object. This separation of the immutable plan from the mutable instances is key to avoid massive maintenance problems. If components could be mutable, that is, have observable state, then no two installations of the same component would have the same properties. The differentiation of components and objects is thus fundamentally about differentiating between static properties that hold for

Components are Binary Units

In this article the general point is made that a software component needs to be a unit of deployment—or, to be more precise, a unit of potentially separate deployment. Any software that is ready for deployment needs to be in binary form. While I have made this point many times, confusion seems to prevail as to what it is that I mean when I say "binary."

For example, Bertrand Meyer, in our ongoing exchange published as part of the *Beyond Objects* column in Software Development Magazine (www.sdmagazine.com), expressed that he finds the qualifications "source" and "binary" confusing, pointing out that in the "good old days (a long, long time ago—1992, perhaps) 'source' meant something like C or Pascal, and 'binary' meant code for some processor." Well, in the really old days, Fortran source, once completed and packaged into libraries, would be shipped as binary components. These components consisted of a deck of punched cards encoding the source (!) of the Fortran code. Job Control Language (JCL) statements on leading cards would instruct the loader of the machine to first compile the cards. (Yes, nothing is new on the face of the earth…) In this case, the deck of cards, used as a software component, is in "binary form"—ready to be used by an automatic execution environment. The Fortran source is included verbatim in the deck, but the leading JCL commands provide the necessary closure to allow load-time compilation.

So, a binary unit's main characteristic is that it can be used directly by the execution environment that the unit targets, whether the unit is a component or not. If the target environment contains an interpreter or compiler, then a binary unit can look very much like source code. However, true source code serves a different purpose: It is written by programmers to be read by both programmers and tools, with an intention to build things. Almost always, source code units are not self-contained. For example, they textually include files from locations specified using file system paths, contain references to build-time variables (conditional compilation), do not contain explicit specifications of what build-tools they require and so on. In fact, source-code unit are quite regularly unusable outside of their delicate build environment. It's true that source code fragility depends on the language and development environment. For example, XML, in combination with XML namespaces, can be seen as a world of "source" that can be directly used as a "binary" as well. The same is true for many scripting languages. However, the fact that the same form can serve both purposes, that of source and that of binary unit, is not a reason to go soft on distinguishing between the two.

To summarize: a unit is a binary if it targets an execution environment; whether the form of that unit is human-readable and whether it is textual or machine code is irrelevant. A unit is a source if it targets human readers as well as development tools. The choice of ahead-of-time, just-in-time, or continuous online compilation or interpretation is one of execution technology that is unrelated to these terms.

a particular configuration and dynamic properties of any particular computational scenario. Drawing this line carefully is essential to curbing manageability, configurability, and version control problems.

Objects. The notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, an object's characteristic properties are that it is a unit of instantiation (it has a unique identity); it has state that can be persistent; and it encapsulates its state and behavior.

Again, several object properties follow directly. Since an object is a unit of instantiation, it cannot be partially instantiated. Since an object has individual state, it also needs a unique identity so you can identify it, despite state changes, for the object's lifetime. Consider the apocryphal story about George Washington's axe, which had five new handles and four new axe-heads—but was still George Washington's axe. This is typical of objects: nothing but their abstract identity remains stable over time.

Since objects get instantiated, you need a construction plan that describes the new object's state space, initial state, and behavior before the object can exist. Such a plan may be explicitly available and is then called a *class*. Alternatively, it may be implicitly available in the form of an object that already exists, that is close to the object to be created, and can be cloned. You'll call such a preexisting object a *prototype object*.

Whether using classes or prototype objects, the newly instantiated object needs to be set to an initial state. The initial state needs to be a valid state of the constructed object, but it may also depend on parameters specified by the client asking for the new object. The code that is required to control object creation and initialization could be a static procedure, usually called a *constructor*. Alternatively, it can be an object of its own, usually called an *object factory*, or factory for short.

Whitebox vs. Blackbox Abstractions and Reuse

Blackbox vs. whitebox abstraction refers to the visibility of an implementation behind its interface. Ideally, a blackbox's clients don't know any details beyond the interface and its specification. For a whitebox, the interface may still enforce encapsulation and limit what clients can do (although implementation inheritance allows for substantial interference). However, the whitebox implementation is available and you can study it to better understand what the box does. (Some authors further distinguish between whiteboxes and glassboxes, where a whitebox lets you manipulate the implementation, while a glassbox merely lets you study the implementation.)

Blackbox reuse refers to reusing an implementation without relying on anything but its interface and specification. For example, typical application-programming interfaces (APIs) reveal no implementation details. Building on such an API is thus blackbox reuse of the API's implementation. In contrast, *whitebox reuse* refers to using a software fragment, through its interfaces, while relying on the understanding you gained from studying the actual implementation. Most class libraries and application frameworks are delivered in source form and application developers study a class implementation to understand what a subclass can or must do.

There are serious problems with whitebox reuse across components, since whitebox reuse renders it unlikely that the reused software can be replaced by a new release. Such a replacement will likely break some of the reusing clients, as these depend on implementation details that may have changed in the new release.

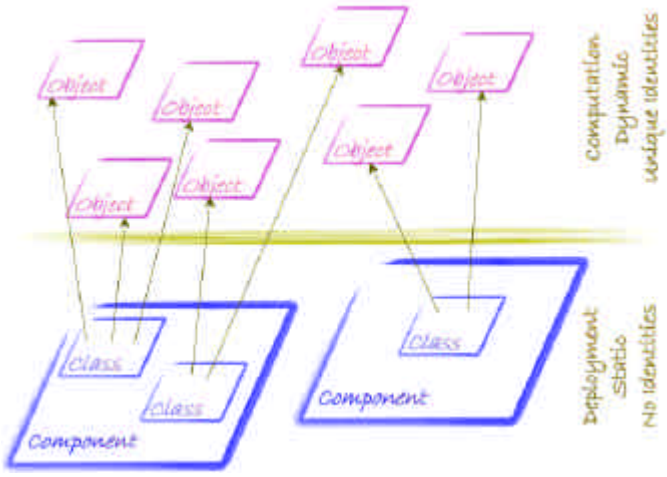


Figure 1. Components are the deployable static units that, when activated, can create interacting objects to capture the dynamic nature of a computation.

Object References and Persistent Objects

The object's identity is usually captured by an object reference. Most programming languages do not explicitly support object references; language-level references hold unique references of objects (usually their addresses in memory), but there is no direct high-level support to manipulate the reference as such. (Languages like C provide low-level address manipulation facilities.) Distinguishing between an object—an identity, state, and implementing class—and an object reference (just the identity) is important when considering persistence. As I'll describe later, almost all so-called persistence schemes just preserve an object's state and class, but not its absolute identity. An exception is CORBA, which defines Interoperable Object References (IORs) as stable entities (which are really objects). Storing an IOR makes the pure object identity persist.

Components and Objects

Typically, a component comes to life through objects and therefore would normally contain one or more classes or immutable prototype objects. In addition, it might contain a set of immutable objects that capture default initial state and other component resources. However, there is no need for a component to contain only classes or any classes at all. A component could contain traditional procedures; or it may be realized in its entirety using a functional programming approach, an assembly language, or any other approach. Objects created in a component, or references to such objects, can become visible to the component's clients, usually other components (or objects in other components). If only objects become visible to clients, there is no way to tell whether a component is pure object-oriented inside, or not.

A component may contain multiple classes, but a class is necessarily confined to a single component, since partial deployment of a class wouldn't normally make sense. Just as classes can depend on other classes (*inheritance*), components can depend on other components (*import*). The superclasses of a class do not necessarily need to reside in the same component as the class. Where a class has a superclass in another component, the inheritance relation crosses component boundaries. Whether or not inheritance across components is a good thing is the focus of heated debate (most likely it is not). The theoretical reasoning behind this clash is interesting and close to the essence of component orientation, but it's beyond the scope of this article.

Modules

Components are rather close to *modules*, as introduced by modular languages in the early 1980s. The most popular modular languages are Modula-2 and Ada. In Ada, modules are called packages, but the concepts are almost identical. An important hallmark of modular approaches is the support of separate compilation, including the ability to properly type-check across module boundaries.

With the introduction of the Eiffel language, the claim was that a class is a better module. This seemed justified based on the early ideas that modules would each implement one abstract data type (ADT). After all, you can look at a class as implementing an ADT, with the additional

properties of inheritance and polymorphism. However, modules can be used, and always have been used, to package multiple entities, such as ADTs or classes, into one unit. Also, modules do not have a concept of instantiation, while classes do. (In module-less languages, this leads to the construction of static classes that essentially serve as simple modules.)

Recent language designs, such as Oberon, Modula-3, Component Pascal, and now C#, keep modules and classes separate. (In Java, a package is somewhat weaker than a module and mostly serves namespace control purposes.) In these languages, a module can contain multiple classes and, where classes inherit from each other, they can do so across module boundaries. You can see modules as minimal components. Even modules that do not contain any classes can function as components.

Nevertheless, module concepts don't normally support one aspect of full-fledged components. For one, there are no persistent immutable resources that come with a module, beyond what has been hardwired as constants in the code. Resources parameterize a component. Replacing these resources lets you version a component without needing to recompile; for example, localization. Modification of resources may look like a form of a mutable component state. Since components are not supposed to modify their own resources (or their code), this distinction remains useful: resources fall into the same category as the compiled code that forms part of a component. A second aspect of components that is not usually associated with modules is the configurability of dependencies.

Component technology unavoidably leads to modular solutions. The software engineering benefits can thus justify initial investment into component technology, even if you don't foresee component markets.

It is possible to go beyond the technical level of reducing components to better modules. To do so, it is helpful to define components differently.

A Definition: Component

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Workshop on Component-Oriented Programming at ECOOP 1996.)

This definition covers the characteristic properties of components I've discussed. It covers technical aspects such as independence, contractual interfaces, and composition, and also market-related aspects such as third parties and deployment. It is the unique property of components, not only of software components, to combine technical and market aspects. A purely technical interpretation of this view maps this component concept back to that of modules, as illustrated in the following.

- A *component* is a set of simultaneously deployed atomic components. An atomic component is a module plus a set of resources.

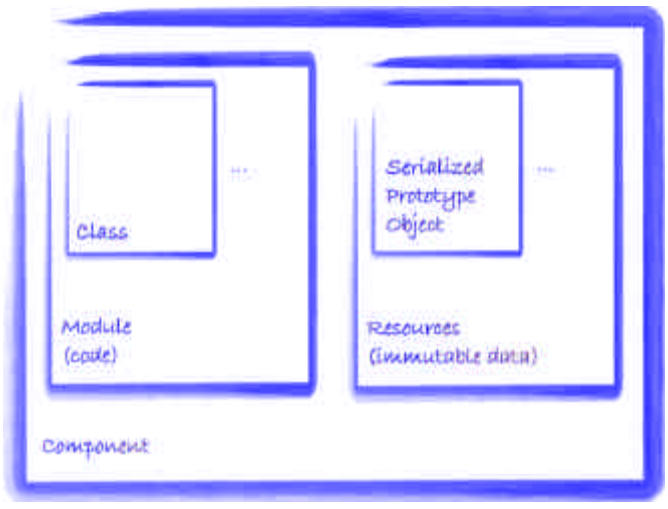


Figure 2. Components contain immutable code and data, typically called modules and resources. Classes can be found inside modules; serialized prototype objects inside resources. The entire structure of a component is immutable and thus suitable for deployment across physically separated systems (by means of replication).

This distinction of components and atomic components caters to the fact that most atomic components are not deployed individually, although they could be. Instead, atomic components normally belong to a set of components, and a typical deployment will cover the entire set.

Atomic components are the elementary units of deployment, versioning and replacement; although it's not usually done, individual deployment is possible. A module is thus an atomic component with no separate resources. (Java packages are not modules, but the atomic units of deployment in Java are class files. A single package is compiled into many class files—one per class. In Microsoft's .NET Framework, the unit of deployment is an *assembly*; also a package containing classes and resources, but based on multiple languages.)

- A *module* is a set of classes and possibly non-object-oriented constructs, such as procedures or functions.

Modules may statically require the presence of other modules in order to work. Hence, you can only deploy a module if all the modules it depends on are available. The dependency graph must be *acyclic* or else a group of modules in a cyclic dependency relation would always require simultaneous deployment, violating the defining property of modules.

- A *resource* is a frozen collection of typed items.

The resource concept could include code resources to subsume modules. The point is that there are resources besides the ones generated by a compiler compiling a module or package. In a pure objects approach, resources are serialized immutable objects. They're immutable because components have no persistent identity. Duplicates cannot be distinguished.

Interfaces

A component's interfaces define its access points. These points let a component's clients, usually components themselves, access the component's services. Normally, a component has multiple interfaces corresponding to different access points. Each access point may provide a different service, catering to different client needs. It's important to emphasize the interface specifications' contractual nature. Since the component and its clients are developed in mutual ignorance, the standardized contract must form a common ground for successful interaction. What nontechnical aspects do contractual interfaces need to obey to be successful?

First, keep the economy of scale in mind. Some of a component's services may be less popular than others, but if none are popular and the particular combination of offered services is not either, the component has no market. In such a case, the overhead cost of casting a particular solution into a component form may not be justified.

Notice, however, that individual adaptations of component systems can lead to developing components that have no market. In this situation, component system extensions should build on what the system provides, and the easiest way of achieving this may be to develop the extension in component form. In this case, the economic argument applies indirectly*:* while the extending component itself is not viable, the resulting combination with the extended component system is.

Second, you must avoid undue market fragmentation, as it threatens the viability of components. You must also minimize redundant introductions of similar interfaces. In a market economy, such a minimization is usually the result of either early standardization efforts in a market segment, or the result of fierce eliminating competition. In the former case, the danger is suboptimality due to committee design; in the latter case it is suboptimality due to the nontechnical nature of market forces.

Third, to maximize the reach of an interface specification, and of components implementing this interface, you need common media to publicize and advertise interfaces and components. If nothing else, this requires a small number of widely accepted unique naming schemes. Just as ISBN (International Standard Book Number) is a worldwide and unique naming scheme to identify any published book, developers need a similar scheme to refer abstractly to interfaces by name. Like an ISBN, a component identifier is not required to carry any meaning. An ISBN consists of a country code, a publisher code, a publisher-assigned serial number, and a checking digit. While it reveals the book's publisher, it does not code the book's contents. The book title may hint the meaning, but it's not guaranteed to be unique.

Explicit Context Dependencies

Besides specifying provided interfaces, the previous definition of components also requires components to specify their needs. That is, the definition requires specification of what the deployment environment will need to provide, such that the components can function. These

needs are called context dependencies, referring to the context of composition and deployment. If there were only one software component world, it would suffice to enumerate required interfaces of other components to specify all context dependencies. For example, a mail-merge component would specify that it needs a file system interface. Note that with today's components even this list of required interfaces is not normally available. The emphasis is usually just on provided interfaces.

In reality, there are several component worlds that coexist, compete, and conflict with each other. Currently there are at least three major worlds emerging, based on OMG's CORBA, Sun's Java, and Microsoft's COM and .NET. In addition, the various computing and networking platforms cause fragmentation of the component worlds. This is not likely to change soon. Just as the markets have so far tolerated a surprising multitude of operating systems, there will be room for multiple component worlds. Where multiple such worlds share markets, a component's context-dependencies specification must include its required interfaces and the component world (or worlds) it has been prepared for.

There will, of course, also be secondary markets for cross-component-world integration. In analogy, consider the thriving market for power-plug adapters for electrical devices. Thus, bridging solutions, such as the OMG COM-CORBA Interworking standard or SOAP (Standard Object Access Protocol) mitigate chasms.

Component Weight

Obviously, a component is most useful if it offers the right set of interfaces and has no restricting context dependencies; that is, if it can perform in all component worlds and requires no further interface. However, few components, if any, could perform under such weak environmental guarantees. Technically, a component could come with all required software bundled in, but that would defeat the purpose of using components in the first place. Note that part of the environmental requirements is the machine the component can execute on. In the case of a virtual machine, such as the Java Virtual Machine, this is a straightforward specification. More generally, this is true for portable intermediate formats, such as that of Microsoft .NET assemblies. On native code platforms, a mechanism such as Apple's fat binaries, which packs multiple binaries into one file, would still let a component run everywhere.

Instead of constructing a self-sufficient component with everything built in, a component designer may have opted for maximal reuse. Although maximizing reuse has many oft-cited advantages, it has one substantial disadvantage: the explosion of context dependencies. If component designs were frozen after release, and if all deployment environments were the same, this would not pose a problem. However, as components evolve and different environments provide different configurations and version mixes, it becomes a showstopper to have a large number of context dependencies. Maximizing reuse minimizes use. In practice, component designers have to strive for a balance.

Component-Based Programming vs. Component Assembly

Component technology is sometimes used as a synonym for visual assembly of pre-fabricated components. Indeed, for relatively simple applications, "wiring" components is surprisingly productive—for example, JavaSoft's BeanBox lets a user connect beans visually and displays such connections as pieces of pipework: plumbing instead of programming.

It is useful to take a look behind the scenes. When wiring or "plumbing" components, the visual assembly tool registers event listeners with event sources. For example, if the assembly of a button and a text field should clear the text field whenever the button is pressed, then the button is the event source of the event "button pressed" and the text field is listening for this event. While details are of no importance here, it is clear that this assembly process is not primarily about components. The button and the text field are instances, that is, objects not components. (When adding the first object of a kind, an assembly tool may need to locate an appropriate component.)

However, there is a problem with this analysis. If the assembled objects are saved and distributed as a new component, how can this be explained? The key here is to realize that it is not the graph of particular assembled objects that is saved. Instead, the saved information suffices to generate a new graph of objects that happens to have the same topology (and, to a degree, the same state) as the originally assembled graph of objects. However, the newly generated graph and the original graph will not share common objects: the object identities are all different.

You should then view the stored graph as persistent state but not as persistent objects. Therefore, what seems to be assembly at the instance rather than the class level—and thus fundamentally different—becomes a matter of convenience. In fact, there is no difference in outcome between this approach of assembling a component out of subcomponents and a traditional

programmatic implementation that "hard codes" the assembly. Indeed, visual assembly tools are free to not save object graphs, but to generate code that when executed creates the required objects and establishes their interconnections. The main difference is the degree of flexibility left in theory. You can easily modify the saved object graph at run time of the deployed component, while the generated code would be harder to modify. This line is much finer than it may seem—the real question is whether components with self-modifying code are desirable. Usually they are not, since the resulting management problems immediately outweigh the possible advantages of flexibility.

It is interesting that persistent objects, in the precise sense of the word, are only supported in two contexts: object-oriented databases, still restricted to a small niche of the database market, and CORBA-based objects. In these approaches, object identity is preserved when storing objects. However, for the same reason, you can not use these approaches when you intend to save state and topology but not identity. You would need an expensive deep copy of the saved graph to effectively undo the initial effort of saving the universal identities of the involved objects.

On the other hand, neither of the two primary component approaches, COM and JavaBeans, immediately supports persistent objects. Instead, they only emphasize saving the state and topology of a graph of objects. The Java terminology is *object serialization*. While object graph serialization would be more precise, this is better than the COM use of the term *persistence* in a context where object identity is not preserved. Indeed, saving and loading again an object graph using object serialization (or COM's persistence mechanisms) is equivalent to a deep copy of the object graph. (Many object-oriented systems use this equivalence to implement deep copying.)

While it might seem like a major disadvantage of these approaches compared against CORBA, note that persistent identity is a heavyweight concept that you can always add where needed. For example, COM supports a standard mechanism called *monikers*, objects that resolve to other objects. You can use a moniker to carry a stable unique id (a surrogate) and the information needed to locate that particular instance. The resulting construct is about as heavyweight as the standard CORBA Object References, but far more flexible, since new moniker classes can be added anytime. Java does not yet offer a standard like COM monikers, but you could add one easily.

Component Objects

Components carry instances that act at run-time as prescribed by their generating component. In the simplest case, a component is simply a class and the carried instances are objects of that class. However, most components (whether COM, .NET, or JavaBeans) will consist of many classes. A single class externally represents a Java bean; thus, a single kind of object represents all possible instantiations or uses of that component. A COM component (or a .NET assembly) is more flexible. It can present itself to clients as an arbitrary object collection, whose clients only see sets of interfaces that are unrelated. In JavaBeans or CORBA, multiple interfaces are ultimately merged into one implementing class. This prevents proper handling of important cases such as components that support multiple versions of an interface, where the exact implementation of a particular method shared by all these versions needs to depend on the version of the interface the client is using. The CORBA Components proposal promises to fix this problem.

Mobile Components vs. Mobile Objects

Surprisingly, mobile components and objects are just as orthogonal as regular components and objects. As demonstrated by the Java applet and ActiveX approaches, it is useful to merely ship a component to a site and then start from fresh state and context at the receiving end. Likewise, it is possible to have mobile objects in an environment that isn't component-based at all. For example, Modula-3 Network Objects can travel the network, but do not carry their implementation with them. Instead, Network Objects assumes that all required code is available already everywhere. It is also possible to support both mobile objects and mobile components. For example, a mobile agent (a mobile autonomous object) that travels the Internet to gather information should be accompanied by its supporting components. A recent example is Java Aglets (agent applets).

What's Up?

While components capture a software fragment's static nature, objects capture its dynamic nature. Simply treating everything as dynamic can eliminate this distinction. However, it is a time-proven principle of software engineering to try and strengthen the static description of systems as much as possible. You can always superimpose dynamics where needed. Modern facilities such as meta-programming and just-in-time compilation

simplify this soft treatment of the boundary between static and dynamic. Nevertheless, it's advisable to explicitly capture as many static properties of a design or architecture as possible. This is the role of components and architectures that assign components their place. The role of objects is to capture the dynamic nature of the arising systems built out of components. Therefore, components and objects together enable the construction of next-generation software.

Clemens Szyperski is the author of the Jolt-award winning book Component Software—Beyond Object-Oriented Programming (Addison-Wesley, 1988). After working both as entrepreneur and as academic, he is now a software architect with Microsoft Research in Redmond, Washington, USA.

Requirements Engineering Series

The RSI Approach to Use Cases

A Pattern for Structured Use Case Development

Use case analysis is a requirements capture technique that is most often used in the early stages of OO and component development projects. Mark Collins-Cope discusses an approach to categorising use cases based on their granularity and level of detail.

Introduction

When engineers first undertake use case analysis, a number of issues are raised for which easy answers can't be found in the text books. These include: What is the appropriate level of granularity for use cases? If large grained use cases are used, should they decomposed into 'lower level' use cases? If so, at what point should this decomposition stop, and how should these sub-use cases be used? Should user or external system interface functionality be described in use case text? Where do report layouts go? Should interchange file formats form part of the documentation? And, in particular, at the end of the use case analysis process, can you answer the question: *What exactly will this application do?*

In this article I look at the RSI approach to use case analysis. This approach provides a framework for analysing and understanding potential use case deliverables and their inter-relationships, with a view to answering the questions detailed above.

RSI Stuctures Use Cases In Three Types

RSI divides use cases three categories, shown by the UML stereotypes: «business requirement» (R), «application interface» (I) and «service» (S).

Business requirement use cases

Business requirement use cases document business processes for which automated support may be required by an application. They detail the business process that is driving the development of an application, are typically low on detail. In their book *Software Re-use*, Jacobson, Griss and Jonsson describe a business use case as follows:

> A **business use case** is a sequence of work steps performed in a business system that produces a result of perceived and measurable value to an individual actor of the business.

To this I would add:

> The **business use case model** (as a whole) defines the business context that is driving definition of the application to be developed.

The target audience for business use cases is very much end users, so the style of documentation should be targeted accordingly - so I recommend following Alistair Cockburn's (from his work on *goal oriented* use cases). Here's an example:

> **«business requirement» use case: get paid for car accident (insurance system):**
> Actor - Claimant
> Actor goal - to get paid for a car accident
>
> 1. Claimant submits claim with substantiating data;
> 2. Insurance company verifies claimant owns a valid policy
> 3. Insurance company assigns agent to examine case
> 4. Agent verifies all details are within policy guidelines
> 5. Insurance company pays claimant
>
> Extensions
>
> 1a. Submitted data is incomplete
> 1a1 Insurance company requests missing information
> 1a2 Claimant supplies missing information

Application interface use cases

Application interface use cases provide a detailed description of the interfaces presented to the application's actors and describe the functionality associated with it.

> An **application interface use case** describes a single interface (file format, report format or dialog) between the application and one or more of it's actors. The **application interface use case model** (as a whole) defines a functional contract between the outside world and the application.

Readers in a software house environment may find the application interface use case model particularly useful in tie-ing down detail for fixed

price development contracts – the application interface use case model provides one way of defining exactly what the application will do.

Application interface descriptions are targetted a two different different audiences: end users - for the user interface, and technical staff - for the external system interface. Detailed user interface descriptions are best 'documented' using a dynamic interface prototype - as this enables users to get a good feel for what the system will actually do, and to give feedback accordingly. In this case, the application interface use case model (the UML bubbles) can still be used to summarise the overall user interface structure, as follows:

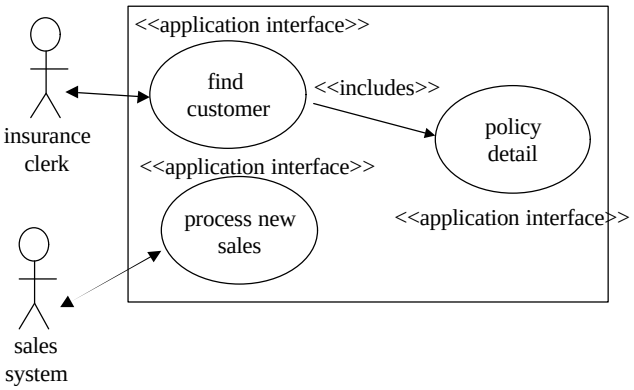


Figure 1. Application interface use case model

This summary diagram tells us that there is a "find customer" dialog (directly traceable from step two of the business use case described above), which uses an associated secondary dialog "policy detail" to show additional details on the customer's policy if requested by the insurance clerk. We'll come back to the "process new sales" use case in a moment.

In the case where it is not feasible to develop a dynamic interface prototype, UI sketches can be used to describe the interface in the following manner (the level of description here is cut down here due to space constraints!):

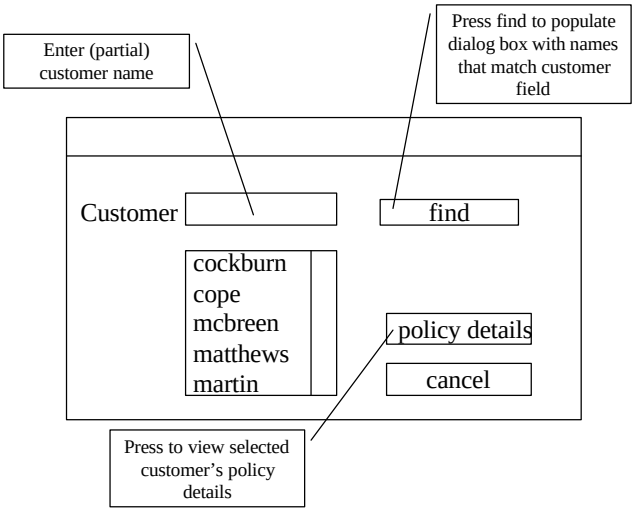


Figure 2. Cut down interface use case description

Application interfaces to external systems (such as the sales system shown in figure 1) indicate some form of external system interaction. In this case, the interface use case will detail the formats used by the use case, e.g.:

«application interface» process new sales - actor: sales system.

This use case is triggered when a file is placed into the **{transfer directory}**. The following file format is used to transfer new sales from the external system into the application.

[new policy holder] [new policy number]

…

Each new policy is processed and added to the application.

Service use cases

Service use cases provide a detailed description of the functionality provided by the *core* of a system in a manner independent of the needs of any particular interface. Take the application interface use case "process new sales." This interface will need to "add a new policy" to the system. Suppose the application also has a (user) interface use case "add new policy." This will equally need to "add a new policy" to the system. So underlying the differences between the two interface formats - one a user interface, the other an external system interface - is a common need. Indeed, it's not too difficult to imagine that the might be a multiplicity of user interface mechanisms that all use the *same* underlying service.

A service use case describes a function the application will provide in a manner independent of the interface used to collect the information it requires, and is *atomic* in that it is guaranteed to run to completion *without* further actor intervention.

The **service use case model** (as a whole) defines a functional contract between the outside world and the application that is independent of the interfaces used by the application.

The service use case model provides an alternative mechanism by which we can answer the question: *exactly what does this system do*? At an intuitive level, we're all quite happy to say things like: "we need to be able to add a customer" without worrying too much about the details of how the user interface is going to be implemented. Probing slightly more deeply, we can also see that regardless of how the information is obtained from the user, the underlying service will need to know the customer's "details." The service use case model provides us with a mechanism to enable us to work at this "intuitive" level.

Service use cases are intended to form the starting point for a component based development of a system - they give us a placeholder from which we can begin to assign services to components. Service use case descriptions are therefore targetted at system developers, as shown in the following fragment of a service use case (directly traceable to the interface to the sales system described above):

«service » "add new policy" (in: policyholder, policynumber)

pre-condition: *true*

post-condition: a new *policyholder* has beed added to *application.policyholders*; and a new *policy* has been added to *policyholder.policies*, such that *policy.number = policynumber*.

Note that the pre- and post-conditions of service use case *cross-reference* a specification level object model (essentially a type model of the system - with no operations) in a formal or semi-formal manner. The following model fragment shows the specification model corresponding to the decription above:

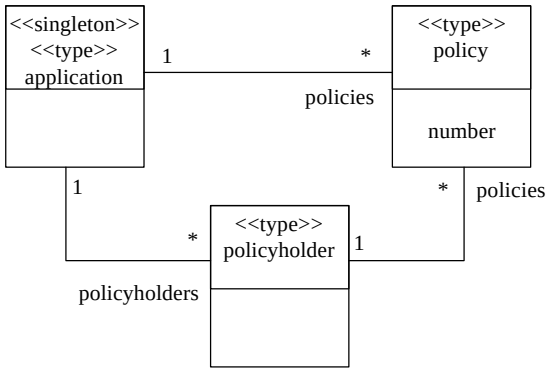


Figure 3. Specification model referenced by service use case description

This brings me to a final important point about the service model:

The service use case model together with its associated specification level object model form a complete **analysis model** of the application.

Inter-relationships

The formal relationship between business, application interface and service use cases is shown in figure #. The relationship can be briefly summarised as follows:

- any individual business process may have a number of application interfaces associated with it;
- any individual application interface may be used to support many business processes, and may require many services to implement its functionality;
- any individual service use case may support many application interfaces.

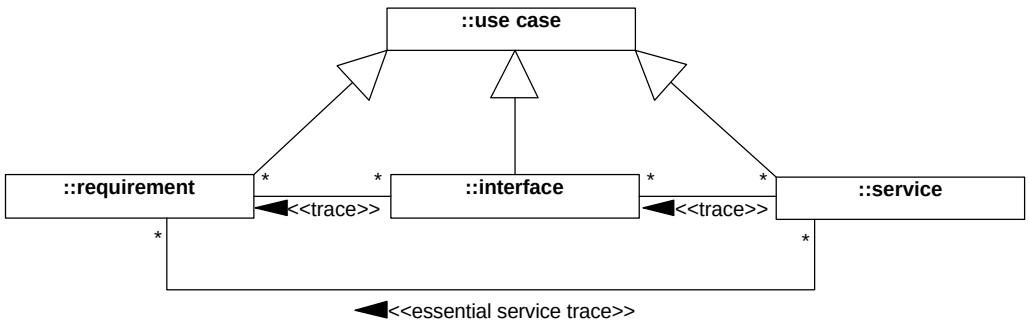


Figure 4. Relationship between BIS use cases

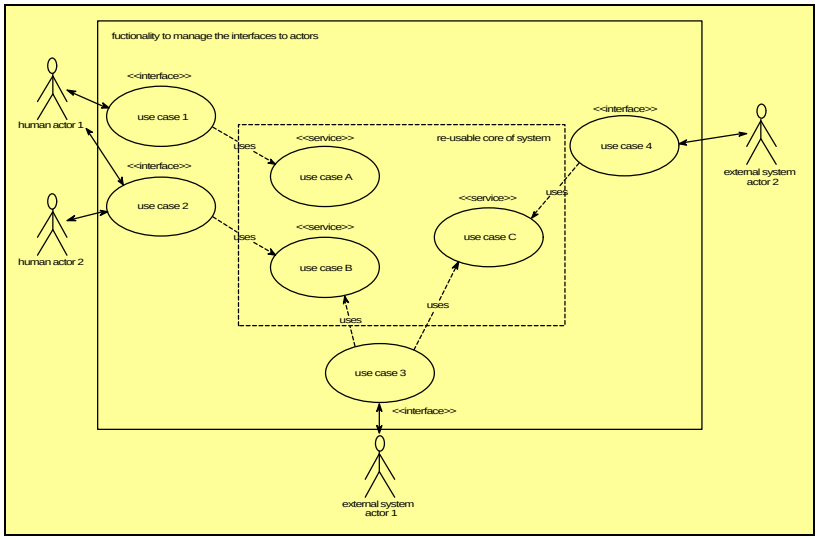


Figure 5. Alternative view of the relationship between service and application interface use cases

A conceptual process

We can see from the inter-relationships between RSI use cases that there are a number of logical dependencies between them:

1. Application interfaces depend of business use cases - if business requirements change, the interface the application presents to the world will need to change too.
2. Services are also dependent on business use cases - if business requirements change, the services provided by a system may need to change accordingly.

But what of the relationship between service and application interfaces. To understand these fully it is necessary break services into two groups subgroups - those that are directly mandated by the business use cases - the *essential* services, and those that are only required to support the application interface (the *non-essential* services). Interestingly, the former tend to be 'updates' to the application (i.e. they change the application's state), and the latter tend to be 'queries' (i.e. they return information about the system's state without changing it).

Given this, we can now describe the dependencies between application inteface and service use cases in the following manner:

3. Application interface use cases are dependent on the *essential* service model. If the essential service model changes, the application interface model may have to change accordingly.
4. *Non-essential* services are dependent on the application interface model. If the application interface model changes, the *non-essential services* may have to change accordingly.

Having understood the dependencies between the various use case types, we can describe the 'conceptual' process of generating the full use case set, which is as follows:

1. Develop the business use case model (for the current project increment)
2. Develop the essential service use case model and specification object model (for the current project increment)
3. Develop the application interface model (for the current project increment)
4. Develop the non-essential service use case model and update the specification object model if necessary (for the current project increment).

This process can be applied informally - in your head - or formally - on paper - as you see fit.

Summary

The RSI approach to use cases structures use cases into three categories, based on their granularity - the scope of the functionality they describe is - and their level of detail - how specific they are about what they say.

- Business requirement use cases describe a business process, and are generally wide in scope but low in detail. They provide a starting point for working out the functionality you might want from your application.
- Application interface use cases give a highly scoped description of a single interface (user or external system) the application presents to the outside world.

- Service use cases give a highly scoped and highly detailed description of the functionality provided by the application in a manner independent of the application's interface to the outside world, and together with its associated specification object model, provides a complete analysis of the application.

I have made suggestions as to how you might document the various types of use case, but more important than this is to understand that they each serve a different purpose during the analysis process - whether you identify them explicitly or not - and that there is a clear set of dependencies between them.

A more comprehensive article on the RSI approach to use case analysis can be found on the following web site:
www.ratio.co.uk/techlibrary.html

Mark Collins-Cope undertakes consultancy for Ratio Group, a training and consultancy company specialising in object and component based technology. Mark can be contacted at markcc@ratio.co.uk, Ratio group can be contacted by telephone on +44 (0)208 579 7900.

e-Business Development Series

Building e-Business Solutions

Enterprise JavaBeans & Intermediate Data

Keiron McCammon on e-Business system development

Abstract

Object-Oriented techniques have failed to deliver the wholesale re-use that they once promised. Few have managed to achieve significant success above the implementation of class libraries. Why? The answer generally cited:
"*Its too damn difficult*"

Delivering re-use across projects at the object level is hard. Re-use first requires the wholesale adoption of a common infrastructure; this infrastructure provides the bed upon which developers can lay down their own applications, assured that they will be able to inter-operate with others. With a common infrastructure in place it is then possible to define re-use at a much coarser level of granularity than the object, the *"component."*

This paper aims to take a look at the industry's move towards a common infrastructure and how this has lead to component-based development. In addition component development brings with it new challenges, one being the management of *"Intermediate data,"* this paper will define what Intermediate data is, its role and how it can efficiently be managed in the component world.

Background

Re-use, The Holy Grail

When object-oriented (O-O) approaches and technologies were first touted they promised to deliver a more natural way of modeling and solving real world problems, moving away from thinking in terms of how the machine (computer) works and thinking more in terms of the physical concepts apparent in the problem domain.

The pillars of the O-O paradigm:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Have provided the foundation for developing highly cohesive, loosely coupled software packages using data hiding (encapsulation) techniques to reduce interdependencies and isolate change. The ability to extend and re-use existing implementation (inheritance/polymorphism) facilitated development of generalized solutions and allowed a more layered and iterative approach to software development. This lead to the fabled belief that O-O would deliver on the promise of re-use. Re-use is seen as the *"Holy Grail,"* reducing development times, improving the quality of code, cutting project costs and generally make the world a better place to live in.

Whilst the adoption of the O-O paradigm has resulted in notable and valuable successes on the path to finding the grail:

- At the language level, standard and commercial libraries abound providing anything from re-useable collection classes, to simplification of complex areas like multi-threading and socket-based communication.
- At the analysis and design level, patterns have proved hugely successful, introducing a common mindset to solving common problems.

We have seen little beyond this *"fine grained"* re-use.

Where is the re-use of actual business processes? Where are the commercially available, *"off-the-shelf*" software components that can be bought and plugged together to provide a solution?

Well, re-use beyond isolated class libraries requires interoperability, to be able to re-use something, it must be able to inter-operate with what is already being used. In the past, to facilitate interoperability traditional 3rd party packages have supplied documented sets of APIs, but since every package is different its rare that they just plug together.

Achieving *"coarser grained"* re-use of actual business processing requires adoption of a common infrastructure. This commonality is the *"enabler"* that allows things to inter-operate *"out of the box"*.

But as many can testify, developing a common infrastructure can represent a significant project cost, and is only any good if adopted by all. But without a common infrastructure how are we ever going to progress beyond simple re-use of class libraries!

A Common Infrastructure

Communication is the basic requirement for any infrastructure, without communication there can be no *"coarse grained"* re-use. But a project specific (or company specific) communication infrastructure, whilst facilitating re-use locally, is not going to promote it to those outside or allow re-use of outside components inside. Therefore the communication infrastructure needs to be defined industry-wide.

This need was identified by the industry some years ago. CORBA and COM initiatives have provided an industry-wide (whether it be industry defined or de-facto) communication infrastructure. Any process using this infrastructure can inter-operate with any other, 3rd party or otherwise.

But communication is only one aspect of interoperability. Re-use of Enterprise business processes requires an infrastructure that encapsulates Enterprise services, like:

- Distributed Transaction Processing

  Co-ordination of processing, guaranteeing *"all or nothing"* semantics.

- Security

  Ensuring communication is secure and not open to abuse or misuse.

- Messaging

  Support for asynchronous, disconnected communication.

- System Management

  Ensure levels of service are maintained through load balancing, resource pooling and high availability/fail-over options.

- Persistence

  Guaranteed storage and recovery of business data beyond the lifetime of a given software process.

Component Based Development

The industry has come to address issues of communication with CORBA and COM and has been dealing with distributed transactions for many years. TPMs and the X/Open XA standard are well established. But in isolation these fail to address the Enterprise issue.

Hence the advent of Object Transaction Management products (OTM), which marry CORBA and TPMs; or MTS, which marries COM in a similar manner. The aim is to provide a higher-level framework for inter-operability offering security, system management (resource pooling, load balancing) and messaging. As a separate initiative, Enterprise JavaBeans (EJB)[1] aims to define the same for Java.

Herald the dawn of component-based development, perhaps the most significant revolutionary step in software development since client-server and a natural evolution of the n-tier/distributed computing paradigm.

A component is a cohesive unit of business processing that has been developed on top of a common infrastructure and hence can be re-used as-is by others. O-O is the foundation of the component, an O-O language is used in implementing the component and its common infrastructure (the building blocks) is defined in O-O terms.

Today, the *"Application Server"* (Weblogic from BEA or WebSphere from IBM, as examples) is the component-based environment. Its underpinnings are the various industry standards which it draws together to provide a cohesive, component framework or infrastructure.

At last we are moving towards an industry-wide component infrastructure.

CALL FOR PAPERS!

Get your most recent paper published in the next issue of ObjectiveView.

Suggested topics are component-based development, object/component architectures, use cases, requirements engineering, experiences in e-business development, experiences using different development processes, etc.

Submit to: objective.view.editorial@ratio.co.uk

Deadline for submission: 24 November 2000

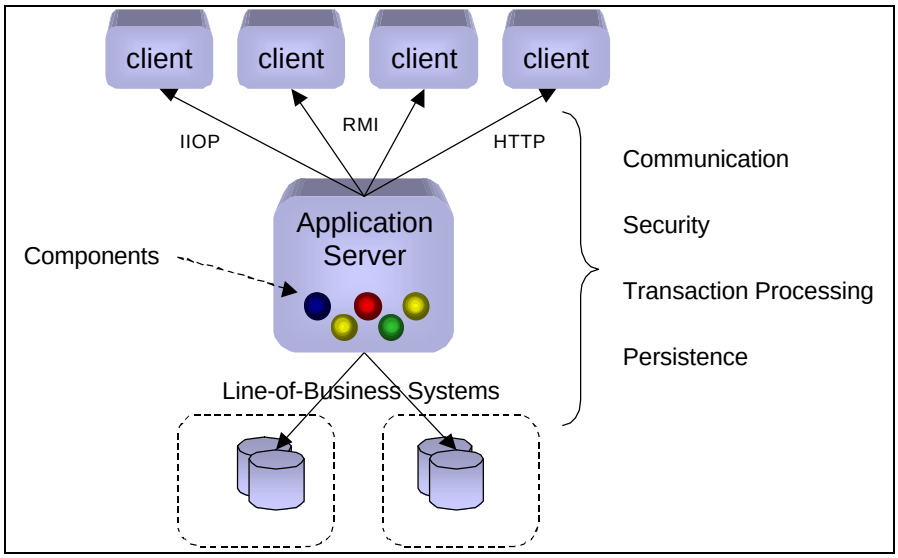


Figure 1. Application Server Architecture

Business Applications

Component-based development and use of Application Server technologies have many applications.

- There will be those who are beginning new projects (Greenfield Development) and want to take advantage of an industry-wide platform that not only provides core Enterprise services but also allows them to draw from a wide pool of skills and expertise.
- There will be those looking to capitalize on a new and growing business opportunity (Components for Resale), the *"component marketplace."* Developing components that can be resold and re-used by others.
- There will be those looking to leverage their investment in existing legacy systems (Enterprise Integration).

Greenfield Development

For those starting anew, looking to build a business solution, component-based development provides an industry-wide framework encapsulating key Enterprise services. There are no legacy dependencies and so the choice of technologies is open.

The key drivers will be time-to-market, ease of development and, if successful, ultimately scalability and performance. The ability to draw on an industry-wide pool of expertise is an additional benefit.

Components for Resale

For those looking to build stand-alone components or packages of components that can be resold as *"off the shelf"* software, component-based development provides the required industry-wide framework allowing components to be re-used by the widest possible audience.

The key driver is the platform and technology independence. End users must have the flexibility to deploy within their Enterprise, utilizing their existing technology and platforms.

Enterprise Integration

The majority of companies have investment in legacy systems, whether these are corporate databases or proprietary applications. For businesses to succeed in the Internet economy they are looking to leverage this investment into the e-business arena, no longer is it acceptable to throw out the old to build the new.

But the requirement is beyond just simply providing Internet access to these existing systems, building new e-business applications means developing significant additional application and business logic. Whether that be to provide a consolidated view of a customer across many *"stove pipe"* line-of-business systems or perhaps building a portal of aggregated information for customers and business partners.

The key drivers are time-to-market coupled with the need to leverage existing systems.

Intermediate Data

Any useful component will need to persist, whether it is to survive failure and aid recoverability, facilitate scalability beyond an in-memory model or just to ensure business transactions are captured (and processed) and information shared appropriately.

However, the type of data being persisted can be viewed in two ways, *"Business"* data and *"Intermediate"* data. Business data represents actual business transactions, whereas Intermediate data is everything else.

Imagine a simple example of an Internet shopping cart. The customer browses the on-line catalog, selecting products to add to a shopping cart. Once happy, the customer then proceeds to the checkout, fills in shipping details, provides credit card information and confirms the order.

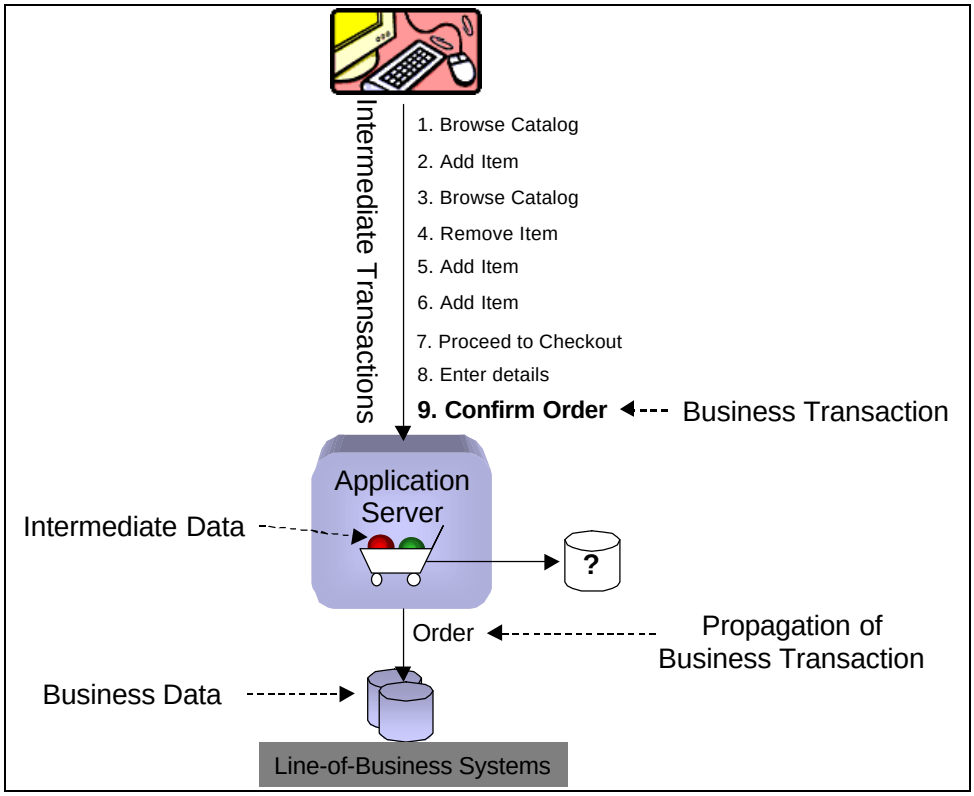


Figure 2. Intermediate Data

In this scenario browsing the on-line catalog, adding items to the shopping cart, filling in shipping and credit card information all happen in middle-tier. It is at the point of order confirmation by the customer that the transaction is of interest to the *"line of business"*, at this point the consolidated order information has to be propagated to the order processing system. Imagine the transactional load on the *"line of business"* systems if they had to handle each customer interaction.

From this example it is clear that Business data and Intermediate data have differing scope: Intermediate data only exists in the middle-tier, serving to support the application logic running there; Business data exists outside of the middle-tier, used to drive on-going business transactions (ultimate order fulfillment).

Additional examples of Intermediate data might include:

- *"Business Intelligence"* data, data captured in the middle-tier and used to provide personalized marketing.
- *"Meta"* data, data that describes how to interact with back-end systems and legacy databases, for business portals.
- *"Workflow"* data, data that describes business rules and the state of ongoing business processes.
- *"Session"* data, data that is relevant only to the on-going interaction with an e-business application (the shopping cart being a prime example).

Providing persistence for Intermediate data should be construed as a middle-tier issue and

management of Business data as an issue of business transaction propagation from the middle-tier to *"line of business"* systems. This approach alleviates the impact of the Internet on the business systems, isolating the transactional load in the middle-tier and gives cause to re-think the role of the database.

The role of the Database in the Middle-tier

Today, the dominant database technology is not best suited to working in the component world. A component represents and encapsulates complex business processing and computations. O-O modeling techniques are used to define a domain model. This might consist of simple-valued attributes (integers, strings), multi-valued attributes (dynamic arrays of values) and complex structures, along with inter-object relationships. And it's these relationships that are key - since they are not just one-to-one or one-to-many, but relationships that include semantics such as, sets (uniqueness), lists (ordering) and maps (associative lookup). These relationships may be complex objects in themselves, perhaps containing hashed values for efficient lookup and retrieval.

The focus of a component is on its business process, not its data. It is this that is fundamentally at odds with the use of relational technologies whose focus is on data, not business processes.

The relational model is based on a rigid, formally defined set of rules, defined by Codd[i] in the earlier 70's. Its aim was to provide flexible definition and storage of simple data based on simple predefined types; the manipulation of this being abstracted via a declarative, set based language. Objects and their inherent complexity were never envisioned in this *"two dimensional"* world.

An object model is *"multi-dimensional"* in nature and incompatible with the *"flat"* relational world. With no built-in ability to handle the complexities of objects the onus is on defining a mapping from one model to the other to overcome this *"impedance mismatch."* A direct impact, aside from the development, testing and maintenance headache, is the effect on performance. For anything other than a simple object model, reconstituting an object from relational tables will involve n-way joins and sorting.

What is needed is something that offers the benefits inherent in using a database:

- ACID Transactions
- Multi-user concurrency control
- Scalability
- Reliability
- Recoverability

Combined with native support for Java…an *"Intermediate Data Management System"* perhaps!

Intermediate Data Management

An Intermediate Data Management System is essentially a database that runs within the scope of the Application Server, in the middle-tier. It provides full database semantics and guarantees (unlike cache-only solutions) and can be shared between multiple instances of an Application Server.

[i] Elmasri, R., & Nacathe, S. (1994). *Fundamentals of Database Systems*. 2nd ed. Redwood City, CA: The Benjamin/Cummings Publishing Co.

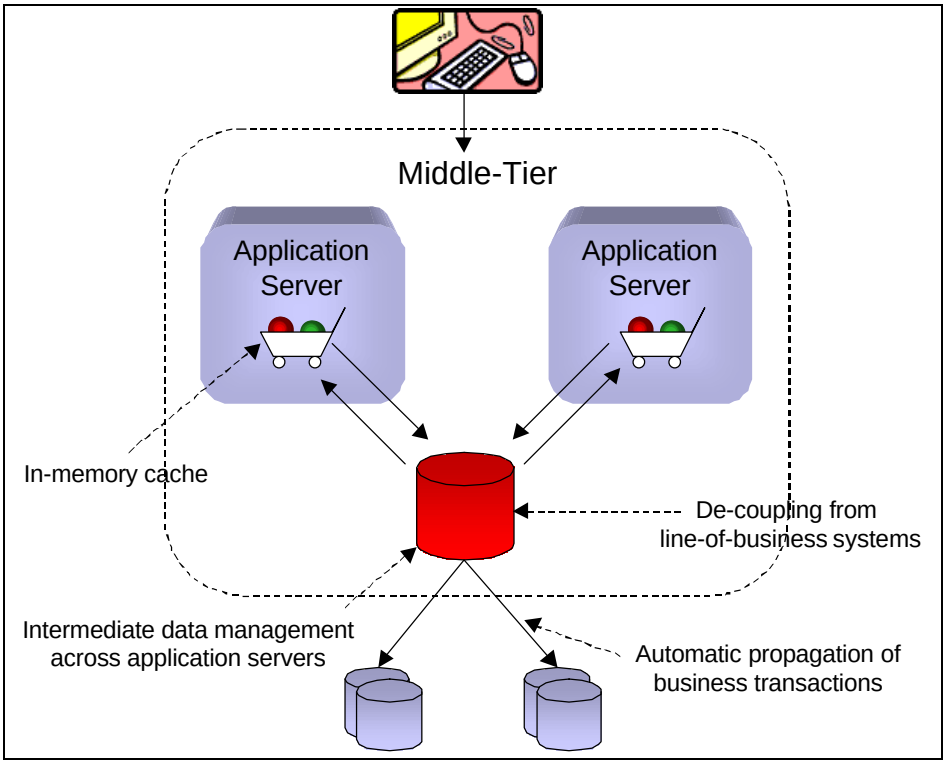


Figure 3. Intermediate Database

Because the *"Intermediate Database"* provides native support for Java (and objects) the developer is freed to focus on developing the business logic and solving the business problem. When it comes to interfacing to the business systems, this is a matter of determining what constitutes a business transaction and how it should be propagated. And of course components built utilizing Intermediate data are able to co-exist with components that directly access existing systems and co-ordinate activity through standard distributed transaction processing.
An *"Intermediate Database"* should have the following characteristics:

- Transparent persistence for Java objects

  Eliminates expensive overhead involved in mapping to/from the domain object model and eliminates the need for anything other then Java development skills.

- Full database guarantees

  ACID transaction semantics (not simply an in-memory cache).

- Shared

  Ability to share Intermediate data between multiple instances of an Application Server to accommodate load balancing and fail-over.

- Distributed transaction co-ordination

  Ability to co-ordinate updates to Intermediate data and "line of business" systems.

- Propagation of business data

  Ability to automatically manage updates to "line of business" databases and systems.

Business Transactions

Propagation of business transactions can be addressed in two ways, one synchronous and the other, asynchronous.

The synchronous approach utilizes the in-built distributed transaction management of the Application Server. The Intermediate data and Intermediate transactions are managed locally in the Application Server, upon completion of a business transaction application logic is used to update both the Intermediate representation and the *"line of business"* systems in parallel. The advantage of this approach is the transaction is propagated immediately, but this is also the disadvantage. For highly transactional systems it may be better to defer updates to some point in time, de-coupling the middle-tier from the *"line of business"* systems.

The asynchronous approach utilizes the database guarantees of the *"Intermediate Database"*,

allowing Business transactions to be cached without being lost. Updates are either propagated individually as they happen, or in batches, depending on throughput. The advantage being, because this is asynchronous, the middle-tier can return to the user prior to the update reaching the *"line of business"* system, safe in the knowledge that even in the event of failure the business transaction will still be propagated in the future. This de-coupling isolates the effect of system failures or downtime.

Of course for a particular solution a combination of both approaches may be appropriate.

Cache-only Solutions

Pure in-memory caching solutions simply provide an object layer on top of an existing relational database (RDB). Whilst they can be useful for mapping business data into the middle-tier for read-only access, they are limited in terms of scalability and ability to support new Intermediate data. Ultimately transactional throughput is limited by the underlying database and its inability to natively support complex object structures. Business transactions have to be propagated immediately since on failure, the in-memory representation is lost.

Versant enJin

Versant enJin™ is the world's first Intermediate Data Management System™. In conjunction with Application Servers like WebSphere™ from IBM and WebLogic™ from BEA, it provides a complete *"solution in a box"* for EJB and component-based development.

It leverages the proven abilities of the Versant database engine to handle Java objects, data complexity and transactional throughput in the middle-tier. Its O/R mapping solution can provide direct access to relational data where required and coupled with replication techniques can be used to propagate business transactions synchronously or asynchronously depending on need.

Summary

The industry-wide adoption of a standard infrastructure is the enabling initiative behind component-based development, which in turn looks set to deliver significant advantages for system development. However, the need for Intermediate Data Management is more compelling today than ever before. The prevalence of O-O approaches and technologies places a focus on developing business logic and solving business problems. Using an *"Intermediate Database"* ensures this focus is not skewed when it comes to considering issues of persistence.

Coupled with the pressures of the Internet to deliver now, to perform now and to scale when needed, new approaches have to be taken. The elimination of the relational mismatch simplifies development, saves time and money and provides a natural part of an Application Server architecture. Simply put:

"It's now much easier"

[1] Enterprise JavaBeans (EJB) specification: http://java.sun.com/products/ejb/docs.html

Keiron McCammon is the Director of Technology & Strategy for Versant. He has worked in the IT industry for over 8 years, principally applying object-oriented technologies and techniques to solving business problems as developer and manager. Since joining Versant in 1996 he has provided services to customers in the Financial and Communications arenas aiding in the development of e-Business solutions utilising Versant and associated technologies.

Visit Ratio's web site at http://www.ratio.co.uk for links on object technology, additional articles, and past issues of ObjectiveView.

Object Design Issues Series

Dynamic Object Model

Ralph Johnson, author of Design Patterns, on a dynamic approach to object structure

Introduction

Recently I have seen many examples of a type of architecture that was new to me. Half of the demonstrations at OOPSLA'97 were examples of this architecture. I have not found any descriptions of this architecture, yet the number of systems that I have seen indicates that it is widely used. This architecture leads to extremely extensible systems, often ones that can be extended by non-programmers. Like any architectural style, there are costs associated with this architecture. It is not efficient of CPU time, but is usually used where this doesn't matter. A bigger problem is that the architecture can be hard for new developers to understand. I hope this paper will help eliminate that problem.

The architecture has many names, sometimes called just a "reflective architecture" or a "metaarchitecture". It was called the "Type Instance pattern" in a tutorial at OOPSLA'95[Gamma, Helm, Vlissides]. This paper calls it the "Dynamic Object Model architecture". Most of the systems I have seen with a Dynamic Object Model are business systems that manage products of some sort, and are extended to add new products, so I have called it the "User Defined Product architecture" in the past[Johnson and Oakes]. I like the name "Dynamic Object Model" because it tends to be used as a modeling tool, and users define their own objects with it.

A Dynamic Object Model for products defines both a Product and a ProductType, and represents a new kind of product with a new instance of ProductType, not a new subclass of Product. Often the Product class has no subclasses, though sometimes the system uses inheritance for customization, as well. A Dynamic Object Model often denegrates inheritance, but it is object-oriented to the core. The purpose is to let people make new kinds of objects without programming.

The Dynamic Object Model has been used to represent insurance policies, to bill for telephone calls, and to check whether an equipment configuration is likely to work. It is used to model workflow, to model documents, and to model databases.

The Structure of the Dynamic Object Model

The Dynamic Object Model architecture is made up of several smaller patterns. The most important is Type Object, which separates an Entity from an EntityType. Entities have Attributes, which are implemented with the Property pattern, and the Type Object pattern is used a second time to separate Attributes from AttributeTypes. The Strategy pattern is often used to define the behavior of an Entity Type. As is common in Entity-Relationship modeling, a Dynamic Object Model usually separates attributes from relationships. Finally, there is usually an interface for non-programmers to define new EntityTypes.

Type Object

Most object-oriented languages have the notion of "class". A class defines the structure and behavior of objects. Most object-oriented systems use a separate class for each kind of object, so introducing a new kind of object requires making a new class, which requires programming.

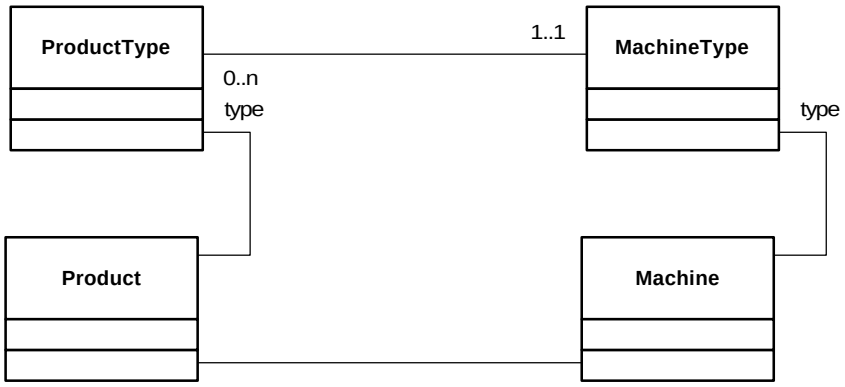
However, often there is little difference between new kinds of objects. If the difference is small enough, the objects can be generalized and the difference between them described by parameters.

For example, consider a factory scheduling system for a factory that makes many kinds of products. Each product has a different set of raw materials and requires a different set of machine tools. The factory has many kinds of machines, and has varying numbers of each. Each type of product would have a plan that indicates how to build it. The plan indicates the types of machines that are needed, but not the particular ones that are to be used. The factory scheduling system takes a set of orders and produces a schedule that ensures those orders are built on time. It assigns each order to a particular set of machines, checking that there are enough machines of a particular type to do all the

work needed in a day. When the factory builds a product, it might record its BuildHistory so that quality control inspectors will know the exact machines that were used to build it.

One way to associate plans with products is to introduce a subclass of Product for each type of product, and to define an operation in each subclass to return the plan. In the same way, there would be a subclass of Machine for each type of machine. However, the only difference between MachineTypes is the number of instances and their name. Further, a plan needs to refer to machine types, and some languages (like C++) make it hard to have an object point to a class or to create an object from a class with a particular name. There should be a MachineType object that knows all the machines in the factory of a particular type. A Plan will refer to a MachineType either by name or by direct reference. A system for designing Plans might require more information about a MachineType, but a system for scheduling will not. If MachineType is a separate class then Machines are general enough that there is no reason to subclass them. In the same way, the only difference between types of products is probably the plans used to make them. It is not necessary to make a subclass of Product for each type of product; make a class ProductType and create instances of ProductType instead of subclasses of Product.

Logical View



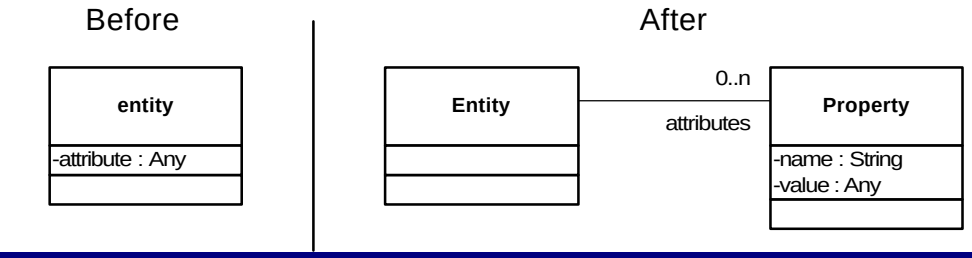
The Type Object pattern splits a class into two classes, Types and Instances, and replaces subclasses of the original with instances of the Type. It can be used in the factory scheduling system to replace subclasses of Product and Machine with instances of ProductType and MachineType. It can be used in an airline scheduling system to replace subclasses of Airplane with instances of AirplaneType (Coad 1992). It can be used in a telecommunications billing system to replace subclasses of NetworkEvent with instances of NetworkEventType. In all these cases, the difference between one type of object and another is primarily their data values, not their behavior, so the Type Object pattern works well.

Property

The attributes of an object are usually implemented by its instance variables. A class defines the instance variables of its instances. If objects of different types are all the same class, how can their attributes vary?

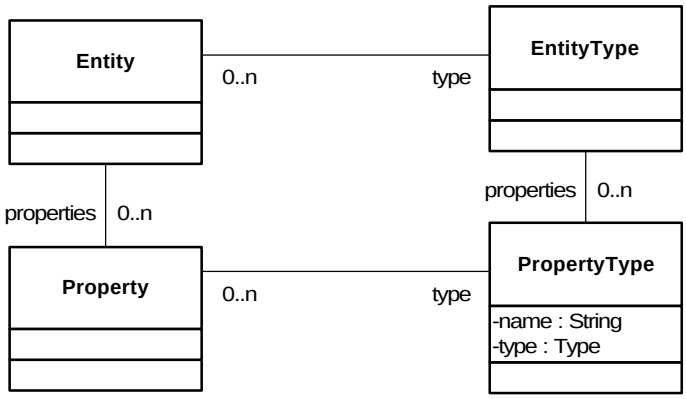
The solution is to implement attributes differently. Instead of each attribute being a different instance variable, make an instance variable that holds a collection of attributes.

Property Pattern



The core of a Dynamic Object Model is a combination of Type Object and Property. Type object divides the system into Entities and EntityTypes. Entities have properties. But usually each property has a type, too, and each EntityType then specifies the types of the properties of its entities. A PropertyType is usually more like a variable declaration than like an abstract data type. It often keeps track of the name of the property, and also whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following:

Dynamic Object Model



Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of Properties to represent a single record, and can use RecordType and PropertyType to represent a table.

But usually different kinds of objects have different kinds of behaviors. For example, maybe records need to be checked for consistency before being written to a database. Although many tables will have a simple consistency check, such as ensuring that numbers are within a certain range, a few will have a complex consistency checking algorithm. Thus, Property isn't enough to eliminate the need for subclasses. A Dynamic Object Model needs a way to change the behavior of objects.

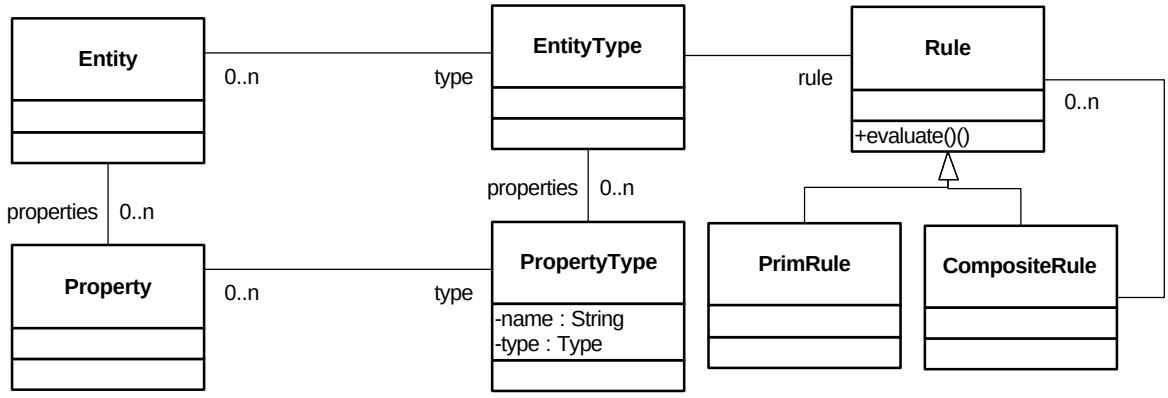
Strategy

A strategy is an object that represents an algorithm. The strategy pattern defines a standard interface for a family of algorithms so that clients can work with any of them. If an object's behavior is defined by one or more strategies then that behavior is easy to change.

Each application of the strategy pattern leads to a different interface, and thus to a different class hierarchy of strategies. In a database system, strategies might be associated with each property and used to validate

them. The strategies would then have one public operation, validate(). But strategies are more often associated with the fundamental entities being modeled, where they implement the operations on the methods.

Dynamic Object Model



Entity-Relationship

Attributes are properties that refer to immutable values like numbers, strings, or colors. Relationships are properties that refer to other entities. Relationships are usually two-way; if Gene is the father of Carol then Carol is the daughter of Gene. This distinction, which has long been a part of classic entity-relationship modeling and which has been carried over into modern object-oriented modeling notations, is usually a part of a dynamic object-model architecture. The distinction often leads to two subclasses of properties, one for attributes and one for relationships.

One way to separate attributes from associations is to use the Property pattern twice, once for attributes and once for associations. Another way is to make two subclasses of Property, Attribute and Association. An Association would know its cardinality.

Another way to separate attributes from associations is by the value of the property. Suppose there is a class Value whose subclasses are all immutable. Typical values would be numbers, strings, quantities (numbers with units), and colors. Properties whose value is an Entity are associations, while properties whose value is a Value are attributes.

Although this is a common pattern, I am not sure why it is used. Perhaps it is just a more accurate model. Or perhaps it is used by habit because designers have been trained in Entity-Relationship modeling. It is interesting that few language designers seem to feel the need to represent these relationships, but most designers of systems with Dynamic Object-Models do.

User Interface for Defining Types

One of the main reasons people design Dynamic Object-Models is so that the system can be extended by defining new types without programming. Sometimes the goal is to enable users to extend the system without programmers. But even when only developers define new types, it is common to build a specialized user interface for defining types. For example, the insurance framework at the Hartford has a user interface for defining new kinds of insurance, including the rules for calculating their price. Innoverse, a telephone billing system, has a user interface for defining geographical regions, monetary units, and billing rules for different geographical regions expressed in various monetary units. The Argos school administration system lets has a user interface for defining new document types and workflows.

Types are often stored in a centralized database. This means that when someone defines new types, applications can use them without having to be recompiled. Often applications are able to use the new types immediately, while other times they cache type information and must refresh their caches before they will be able to use the new types.

The alternative to having a user interface for creating and editing type information is write

programs to do it. In fact, if programmers are the only ones creating type information then it is often easier to let them do it by writing programs, since they can use their usual programming environment for this purpose. But the only way to get non-programmers to maintain the type information is give it a user interface.

Advantages of Dynamic Object Models

If a system is continually changing, or if you want users to be able to extend it, then the Dynamic Object Model architecture can be very useful. The alternative is to pick a simple programming language that is flexible and easy to learn. In fact, a Dynamic Object Model is a kind of programming language.

Systems based on Dynamic Object Models can be much smaller than alternatives. One architect told me that his 50,000 line system had more features than systems written without a dynamic object model that took over 3 million lines of code. I am working on replacing a system with several millions lines of code with a system based on a dynamic object model that I predict will require about 20,000 lines of code. This makes these systems easier to change by experts, and (in theory) should make them easier to understand and maintain.

Disadvantages of Dynamic Object Models

A Dynamic Object Model is hard for most programmers to understand and to use. The architects of systems that use a Dynamic Object Model often consider them the highlight of their careers, but programmers working on the systems often hate them. Part of the problem is that these systems are usually underdocumented, but another part is that the systems are abstract and so hard for most programmers to understand. This is by far the biggest disadvantage of this architecture, and architects should choose it cautiously and plan to spend more than usual on documentation and training.

A system based on a Dynamic Object Model is an interpreter, and can be slow. Most of the systems I've seen only required a little optimization to be fast enough. However, I've also seen a few in which some of the features were too slow.

A system based on a Dynamic Object Model is defining a new language. It is a domain-specific language that is often easier for users to understand than a general-purpose language, but it is still a language. When you define a new language, you have to define support tools like a debugger, version control, and documentation tools. This is extra work. If you let users define their own types, you have to teach them good software engineering practices like testing, configuration control, and documentation. Is it worth the effort? Some designers do not worry about this and their projects usually come to a bad end. Others avoid these problems by only allowing developers to define new types. Others train their users. There are many ways around this problem, but it is a problem that should be faced and not ignored.

Summary

A Dynamic Object Model provides an interesting alternative to traditional object-oriented design. Like any architecture, it has both advantages and disadvantages. The more examples we study, the better we will understand its strengths and weaknesses. Please contact me if you have used this architecture in the past and can provide more examples or if you know of any papers that describe this architecture or aspects of it.

Bibliography

[Foote98] Brian Foote and Joseph Yoder, *Metadata and Active Object-Models*, presented at PloP'98, Allerton Park, August 1998.
[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Johnson97] Ralph Johnson and Bobbie Woolf, Type Object, In *Patttern Languages of Program Design* 3, Robert Martin, Dirk Riehle, and Frank Buschmann ed., Addison-Wesley, 1997, pp. 47-66.
Also see http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata/UoI98MetadataWkshop.html

Ralph Johnson is one of the four authors of "Design Patterns." He teaches courses at the University of Illinois at Urbana-Champaign on object-oriented design, and specialises in helping companies design and document their frameworks. He has helped develop frameworks for accounting and billing, amongst other areas. He is not only an expert designer, he is also a master at explaining object-oriented design.

Software Process Series

Goldilocks and the Three Software Processes

Doug Rosenberg and Kendall Scott on the ICONIX Unified Process

Once Upon A Time there was a project manager named Goldilocks who went for a stroll in the bookstore. Goldilocks wasn't sure exactly what she was looking for, but she was thinking about how to manage her current software project, which was an international banking project being developed in London, Geneva, and New York. Goldilocks wandered into the Computer Books section, and was surrounded by shelves full of books on Enterprise Java Beans, COM+, and XML. Off in a corner of one shelf, she spotted some books on software development processes. "Aha!" Goldilocks exclaimed. "I think a development process might be just what we need."

Goldilocks saw three books from the same publisher that looked interesting. There was a Great Big Process, called the Rational Unified Process, or RUP; a medium-sized Process, called the ICONIX Process; and a little teensy-weensy process called Extreme Programming, or XP for short. When she saw the books for the Three Processes, she smiled and clapped her hands. "How pretty!" she cried. "I wonder where the writers are?" She stood on her toes and peeked over the information booth. There didn't seem to be anyone around, so Goldilocks started looking at the front and back covers of the books!

The first thing she noticed was that even though the books were all about the same size, they described porridge in bowls of three very different sizes: a great big bowl for the RUP, a medium-sized bowl for the ICONIX Process, and a tiny little bowl for XP. "Oh, what a joy to have three processes to choose from!" Goldilocks said. Then, as she was feeling really curious, she opened up the RUP book to taste the porridge.

"Ooooh!" she cried, dropping the book. "That porridge is much too thick and heavy!"

The RUP porridge was very thick and heavy indeed. Here's just a sampling of what Goldilocks saw:

- References to over a thousand pages of material describing all of the "artifacts" that a team has to produce to maintain adherence with the process. ("How will my team *ever* get all of this done?" Goldilocks wondered. "I know what'll happen: they'll run out of time producing tons of artifacts, and then jump straight to code without ever finishing the design, and *then* where will we be?")
- Activities broken down into "thinking" steps, "performing" steps, and "reviewing" steps. ("Don't people already do these things without being told how to do them?" Goldilocks mused.)
- Milestones called Life-Cycle Objective, Life-Cycle Architecture, and Initial Operational Capability. ("Whatever do these mean?" Goldilocks pondered. "Wouldn't it be better to establish milestones that are a little less lofty and easier for everyone to understand?".)

Next, she tasted the porridge in the tiny little bowl. But that porridge was much too thin.

The XP porridge was very thin indeed. To wit:

- "There are only four important things about software: Coding, Testing, Listening, and Design." ("Gosh, aren't requirements important?" Goldilocks asked. "I thought there were some pretty important regulations about international funds transfer that we had to comply with.")
- "The code *is* the design." ("Golly, I always thought design came *before* code!" Goldilocks exclaimed. "It's funny how code always seems to come first in this process.")
- "Do the simplest thing that could possibly work." ("So, if you think you might not need it, you don't need it," Goldilocks deduced. "But what happens when the *customers* decide *they* need it?")

Then she tasted the porridge in the medium-sized bowl. "Mmmmmm," she said. "This porridge is just right!" So she ate it all up!

The ICONIX porridge had just the right consistency. For instance:

- It offers a streamlined approach to software development that includes a minimal set of diagrams and techniques that a project team can use to get from use cases to code quickly and efficiently.
- It includes extensions to the UML that save time and money, and consistently yield good results.
- It focuses on helping projects avoid the dreaded analysis paralysis at those points at

which it's all too easy to get bogged down in nonproductive activities.

Then Goldilocks saw three chairs set before the bookshelf: a great big chair for RUP, a medium-sized chair for the ICONIX Process, and a tiny little chair for XP. "Oh, it would be nice to sit down for a while!" Goldilocks thought.

So she climbed into the Great Big chair that belonged to the RUP. "Oh, no!" she said. "That chair is much too hard."

Let's see what Goldilocks was trying to absorb about the RUP.

- The full RUP includes four phases, which are fairly easy to understand. But it layers six "engineering workflows" and three "management workflows" over the phases, and *then* it brings in "iteration workflows," which supposedly describe the process more from the perspective of what happens in a typical iteration. ("Isn't it too bad that they don't just go to the more practical stuff in the first place, and dispense with all of the heavy theory?" Goldilocks wondered.)
- Since the RUP describes phases and iterations, it's necessary to produce phase plans and iteration plans. But the RUP also talks about having to plan the plans, and that's where Goldilocks drew the line. ("So, who plans to plan the plan?" Goldilocks asked, not entirely in jest. “All this plan-planning makes my head hurt.”)
- The RUP specifies roles for eleventy-twelve different kinds of workers (27, actually, but still), including someone called a Use-Case Specifier, who "details the specification of a part of the system's functionality by describing the requirements aspect of one or several use cases." ("Doesn't this make use cases sound a lot scarier than they actually are?" Goldilocks said.)

Then she sat in XP’s tiny little chair. "Oh, no," she said, "That chair is much too soft!"

Why did Goldilocks decide that XP was too soft? Here are some ideas:

- XP disciples say there's simply no point in trying to do analysis since customers almost never know what they need at the beginning of a project, and even when they do start figuring it out, they change their minds weekly, even daily, sometimes hourly. Well, actually, XP programmers are supposed to do analysis on an ongoing basis as they're writing code, but since that code doesn't involve customers, saying that XPers do analysis is more than a little disingenous. ("And how come I can't find the word *analysis* in the index of Kent Beck's book?" Goldilocks wondered.)
- One XPer declared that use cases are just too complicated. "Use cases as I have seen them used are complex and formal enough that business doesn't want to touch them." This attitude is meant to further justify that you can more or less skip analysis because it's too hard to capture the results in a way that will please customers. ("But isn’t it easier to try to figure out what you’re building *before* you start building it?' Goldilocks asked.)
- XPers like to talk about how programmers should all be in one room, coding in pairs, how they use index cards to capture the things that their code can't, and how it’s a Very Brave Thing to rely on Oral Documentation." What happens when you have more than 10 or 12 developers, and they don't even live in the same area? Or when you need to capture things, like relationships among sets of classes and the larger context in which the system will operate, that don't lend themselves to small pieces of paper? Or when you realize that you need a full-time person to maintain the repository of project information, and there's no one around who can keep track of all of that "oral tradition"? ("How on earth am I going to coordinate what my programmers in London, Geneva, and New York are doing without anybody writing anything down?" Goldilocks pondered. “This oral documentation stuff sounds oxymoronic, but I’m afraid that for my project it would be just plain moronic.”)

Next, she sat in the ICONIX Process's medium-sized chair. "Ahh," she said with a smile. "This chair is just right!"

Here are some reasons that Goldilocks felt so comfortable in the ICONIX chair:

- The approach advocates starting with domain modeling, which involves identifying the objects in the real world that will serve as the vocabulary for the use cases. This gets much of the team meaningfully involved in the project right away, as opposed to having most of the players wait around for all the planning to get done, or putting everyone except the "star" programmers on hold while the latter huddle in a room (in pairs, of course) and build the system they feel like building.
- The ICONIX approach to use cases involves a healthy number of small pieces of straightforward text that captures functional requirements in a manner that's easy for everyone to understand. There's no place for

long and involved use case templates that simply clutter up the model, and the use cases also provide a firmer foundation for negotiation and exploration than index cards containing user "stories".

- Robustness analysis, which sits at the center of the ICONIX approach, is a deeply useful technique specifically designed to close the eternal gap between what the system being modeled is supposed to (the results of analysis) and how the system is going to function (the results of detailed design). This "missing link" provides a high degree of traceability that's simply not available in a Big Process (it gets buried under all that extra stuff) or in a Tiny Process (case in point: look up "requirements" in the index of the XP book, and see what you find—nothing).

Goldilocks, though, was very curious indeed, so she decided to give the Tiny Process chair another chance, because she liked the sound of XP's "core values," Communication, Simplicity, Feedback, and (especially) Courage. Just then, though, there was a loud crack! and the little chair broke right through!

Goldilocks stood up and dusted herself off. (It turns out that "courage," which in XP terms basically means "feel free to start coding right away and spend lots of time ripping up and rewriting code you've already written because you didn't understand what it was supposed to do when you coded it," wasn't the most suitable principle for Goldilocks to focus on for her software development project.) So she climbed upstairs to the lounge of the bookstore. There she saw three beds all in a row.

"Oh," she said, yawning. "I am feeling sleepy."

So she pulled down the covers and climbed into the RUP's Great Big bed. But she quickly jumped down. "That bed is much too hard.," she said.

Why couldn't Goldilocks sleep in the Great Big RUP bed?

- Rational says the RUP is highly customizable. For instance, you can add, change, or remove activities; add checkpoints for review activities; add guidelines; and tailor templates. But in doing so, you have to plan the process implementation, execute the process implementation, and evaluate the project implementation—and then start over if the process didn't take. ("Isn't it less work to start with a smaller process and add what you need to it instead of starting with everything in the whole wide world and taking out what you don't need?" Goldilocks wondered.)
- In some ways, the RUP is really less about process than it is about Rational's tools. Trying to capture requirements? Requisite Pro is just the thing for you. Managing analysis and design models? You simply have to have Rose. Configuration management and change control? You really can't do those without ClearCase. The amigos even admit that the tools and the process were developed together. ("Goodness, I'm not sure if I want to tie my entire project to a single vendor" Goldilocks pondered.)
- The RUP has a number of dubious constructs and also some rather gaping holes. It advocates the use of large and unwieldy use case templates, which we alluded to earlier, as opposed to compact text that's less likely to intimidate customers (and developers, for that matter). It also says that you should write "flows of events—design," as opposed to just putting the text of the use case on your sequence diagrams. But the RUP somehow manages to shortchange domain modeling, just like Objectory did, and its use of the UML falls short in some other areas as well.

Then she tried XP's tiny little bed. But it was too soft.

Why did Goldilocks realize so quickly that XP is just too soft for most projects?

- XPers like to talk about something we might call the "ready, fire, aim" sequence, which works in conjunction with the one-day-or-less "nanoincrement." Unfortunately, this basically boils down to the combination of (a) a refusal to spend any meaningful amount of time figuring out what to do before starting to "produce," and (b) the insistence that doing great work on an ongoing basis "in the small" will somehow magically result in the whole system being correct when the smoke clears. ("Gee, the book says that XP is different from 'Cowboy Coding', but what's the difference, really, if you jump to code before understanding your requirements?" Goldilocks queried.)
- In the eyes of true XP believers, documentation is basically useless. Of course they don't need to document the code, because "merciless" refactoring results in perfect (and perfectly readable) code. Of course they don't need to draw pictures, because the code is the design, and analysis doesn't come into play. And of course they don't need user guides, because "all documentation is to be distrusted as out of date and subjectively biased." Of course, oral tradition falls apart when a few of the traditionalists leave, but that doesn't seem

to bother XPers, who insist that the customer has the right to request useless documentation from the coders, if they're foolhardy enough to do so. ("You mean my customers would have to make special requests to get material that explains how their system works?" Goldilocks wondered.)

- Let's not forget the mystical aspects of XP. Our favorites have to do with "asking the code." Actual statements by disciples of this less-than-lightweight "process" include the likes of "Restructur[e] the system based on what it is telling you about how it wants to be structured," "The system is riding me much more than I am riding the system," and our all-time favorite, "A 'code smell' is a hint that something has gone wrong somewhere in your code." Smell the code, indeed.("I thought my friend Alice had a strange time when she fell down that rabbit hole, but that was tame compared to some of this." Goldilocks mused. "I wonder if Alice's mushrooms would solve the scalability problems").

So she climbed into the ICONIX Process's medium-sized bed. It was just right. Soon Goldilocks was fast asleep!

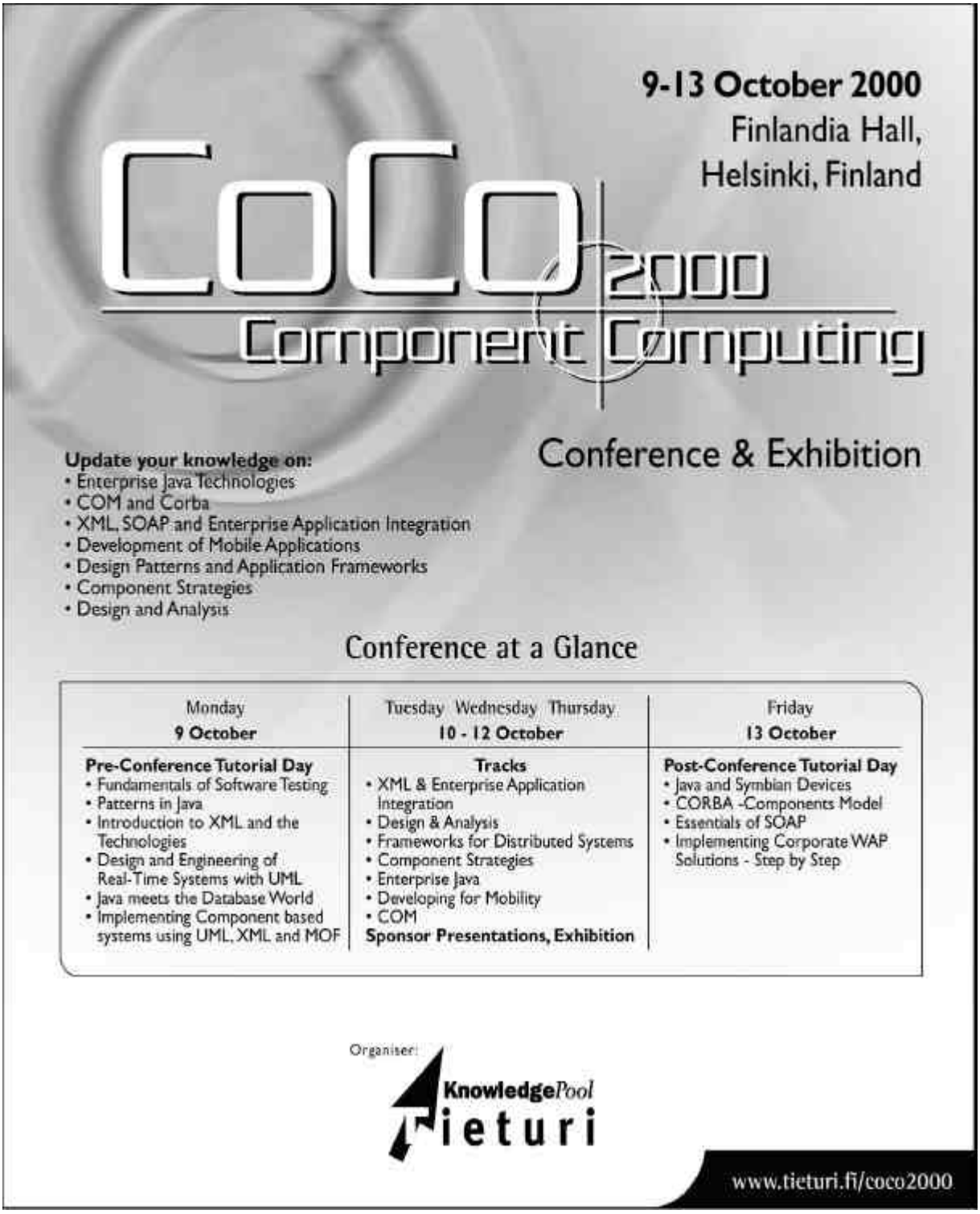
Shall we speculate on how Goldilocks got to sleep so quickly in the ICONIX bed?

- The ICONIX process is 98 percent fat-free. It focuses on what you need, and ignores what you probably don't need. The key is that it helps you stay relentlessly focused on answering the fundamentally important questions about the system you are building while also helping you refuse to get caught up in superfluous modeling issues. If it turns out that you really need to do activity diagrams, or model several levels of worth of substates, then you can simply add those kinds of tasks to the ones that the process prescribes; you just don't have to make the effort to remove the extraneous stuff.
- On the other hand, the ICONIX process is still a real OOA&D process, with analysis and design playing appropriately important roles. It advocates use case modeling, a technique that has worked on countless projects, to capture requirements. It describes how to build sequence diagrams that will serve as the foundation of detailed design. And it uses robustness analysis to close that infernal gap between "what" and "how," which helps a project team build the right system *and* build the system right.
- The creators and popularizers of the ICONIX process don't offer lofty claims or catchy slogans. They don't proclaim that the process captures many of the best practices in modern software development (even though the practices it addresses are, indeed, very strong practices indeed). They don't spout nonsense about how the exponential cost of change curve is no longer valid. Instead, they offer plain talk about an elegant yet rigorous process that's customizable and scalable without being overwhelming.

While she slept, Goldilocks had a curious dream in which the words "minimal yet sufficient" kept repeating over and over. When she awoke, she realized that these words were the key to implementing a successful software process. Goldilocks knew that she had to avoid too much process or her project would fall into Analysis Paralysis, and yet she needed a process that was scalable and sufficient to keep her project from degenerating into chaos. "RUP looks more than sufficient, but it sure isn't minimal" Goldilocks said to herself, "and XP is most certainly minimal, but there just doesn't seem to be enough there for my project".

So Goldilocks put the RUP book and the XP book back on the shelf, and took her copy of *Use Case Driven Object Modeling with UML* to the cash register. After reading it carefully (which only took her a single evening), she ordered a copy for each of her programmers in London, Geneva, and New York. The software was delivered on schedule, bug-free, and met all the customer's requirements. Goldilocks got a big raise, her customers were all thrilled, and everyone lived happily ever after.

Doug Rosenberg is the author of "Use Case Driven Object Modeling with UML -- A Practical Approach" with Kendall Scott. Founder and President of ICONIX Software Engineering, Inc., he has been teaching OOAD since 1992, several years before the advent of UML. Kendall Scott, the supporting author with Martin Fowler of the award-winning "UML Distilled", is a UML consultant and mentor with particular expertise in domain modeling and requirements capture via use case modeling. He is also the principal of Software Documentation Wizards. Kendall and Doug are currently writing "Applied Use Case Driven Object Modeling -- An Annotated E-Commerce Example", and "Understanding Distributed Components: Cutting Through the COM, CORBA and EJB Hype", both scheduled for release in 2001.



RATIO GROUP AT COCO 2000!

Mark Collins-Cope, Technical Director at Ratio Group will be delivering two presentations at this year's Component Computing conference, on Tuesday 10 October:

- A Reference Architecture for Object and Component Based Systems
- Use Case Analysis for Component Based Systems

Also stop by the Ratio stand in the Exhibit Hall for information on our services including expert training, consultancy, mentoring, and development services.

W E K N O W T H E O B J E C T O F

TRAINING

Excellence in Object and Component Training

The following courses are offered both in-house and on a regular public schedule basis.

Object-Oriented Analysis & Design using UML

This course gives you a practical understanding of the major techniques of the UML (Unified Modelling Language) object-oriented analysis and design notation, and how these techniques can be applied to improve quality and productivity during the analysis and design of computer systems.

What they thought…

"Thanks for this! Everybody is buzzing after the course. Thanks to you and your team for all of your efforts, particularly the lecturer, who has an excellent manner and just knows his stuff inside out."
Chris McDermott, Polk Ltd.

Component-Based Design using UML

This course gives you a firm understanding how to analyse and design extensible and customisable re-usable business (domain) oriented components, and how to assemble such components to create bespoke applications. The course has a clear focus on the architectural aspects of component-based design.

What they thought…

*"Patterns were particularly useful as were the hints & tips & tricks that were sprinkled throughout. It was also very useful to be shown *why* some of the techniques we use are good; up until now we've been choosing the techniques based on instinct."*
Phil Harris, Silicon Dreams

Object-Oriented Programming in C++

This course will leave students with a firm understanding of the C++ language and its underlying object-oriented principles. Attendance on the course will enable participants to make an immediate and productive contribution to project work.

What they thought…

"This has been a worthwhile exercise. The course was concise ... well focused via examples and practical sessions"
Course delegate, MTI Trading Systems

"Things were explained clearly, in simple terms and with relevant examples."
Course delegate, Primark

Object-Oriented Programming in Java

This course will give you a practical understanding of the major features of the Java development environment and language, both in the context of web applets, and in the context of stand-alone applications. Students will leave the course able to start productive work immediately.

What they thought…

"I particularly liked the hands-on implementation of the Java language theory in an extendable example."
Graham Hoyle, Tetra Ltd.

"Really good course, well presented, well informed, lots of leads to wider ideas, etc."
Roy Turner, Silver Platter Information Ltd.

Email **info@ratio.co.uk** or call Ratio Sales on +44 (0)20 8579 7900
for more information