Why IT veterans are sceptical about MDA

Graham Berrisford
Atos Origin (UK)
graham.berrisford@atosorigin.com

Abstract: This paper identifies problems with the MDA approach to specifying transformations, and barriers to MDA adoption such as the scale of the enterprise problem domain, the skills and knowledge required, and the distributed system problem.

The paper addresses question such as: What kind of detail is best suppressed from a model? What level of granularity is best for modelling business rules? How to capture all the essential business rules in a model? How to make UML more helpful to analysts looking to build a PIM or CIM? How to build enterprise-scale models?

The paper promotes the importance of understanding where and how persistent data is divided between discrete data stores and where units of work can be rolled back. It suggests models that are to be completable by business analysts yet also transformable by forward engineering must be event-oriented as well as object-oriented.

Introduction

An enterprise may own and maintain millions of lines of software code. We all know how difficult it is to read code and understand its purpose. Even the most extreme of extreme programmers agree that we need to maintain abstract specifications that are more concise than the code.

A never-ending fascination of our business lies in the immense variety of answers to three questions: What kind of abstract specification is best? How many levels of abstract specification do we need? How tightly do we maintain the abstract specifications in alignment with the code?

Many believe an abstract specification should take the form of a model. Readers of this paper will already have heard of the OMG's MDA (Model-Driven Architecture) and its three levels of model: CIM (Computation-Independent Model), PIM (Platform-Independent Model) and PSM (Platform-Specific Model). What these terms might mean is explored in the paper.

There is something to be said for an MDA scheme that is vaguely defined. Interest groups and vendors can use such a scheme as a springboard to invent new ways to be more efficient and effective in the analysis, design and construction of software systems. Even if they only reshape their existing ideas to fit the scheme, they are likely to clarify and elaborate those ideas.

There is value in the promoting and discussing MDA as a device to bring together interest groups and vendors from different realms. This encourages cross-fertilisation and new ways of thinking and working. It may help to improve UML and to reinvigorate efforts to define higher level or more universal programming languages. This last appears to be what many are focused on.

Nevertheless, the pedants amongst us hanker for more clarity in and wider agreement about the definitions of CIM and PIM. And the veterans amongst us are wary of people using MDA to recycle ideas that have not proved successful in the past. This paper sets out some reasons why some pedants and veterans are sceptical about MDA.

Two challenges facing MDA

Taking UML as a starting point, MDA is expected to meet the challenges of those wanting more abstraction and those wanting more detail. Let me quote the practical experience of a software engineer who religiously maintained an abstract specification of his software in the form UML diagrams.

> "The [UML] models I produced were useful to me, but far too detailed to be helpful for talking about the design with someone else. I had to produce more simplified views for this, but I was proud of the fact that my models always represented the state of the code. Unfortunately, maintaining a very detailed model is really no easier than maintaining the code. My models tended to become rigid, even though I was working with UML diagrams. Finally, I have decided that working with too detailed a model is a trap. Basically this is the same trap that we were trying to avoid by working with UML in the first place. It's very important to work at the correct level of abstraction. I won't be going back to using forward code generation from a model."

This most earnest of software engineers concluded that higher-level or more abstract specifications are needed. So, MDA has to meet the challenge of analysts and designers who want languages that are abstract enough for their specification purposes.

At the same time MDA has to meet the challenge of analysts and designers who want languages that are semantically rich enough to specify all the necessary business rules. To code any process we need to know not just its inputs and outputs, but its preconditions and post conditions as well.

How can we build models that meet these apparently conflicting challenges (abstraction and comprehensive business rule specification) at once? Will MDA help?

On models

A model is an abstraction of the real world. A model can represent only tiny part of the real world, usually a part we want to monitor if not control. A comprehensive model has both structural and behavioural aspects; it features both persistent entities and transient events; it defines the business rules that are supposed to apply in the relevant part of the real world.

A business rule may be invariant (guaranteed to hold true at any time) or transient (guaranteed only immediately before or immediately after a discrete event). An invariant business rule may be declared as a property of a persistent entity. A transient business rule may be declared as a property of a transient event.

A running data processing system is itself model, it is an abstraction of the real world. So, a logical model of the real world model can be abstracted from the platform-specific definition of a data processing system. You may abstract logically discrete entities from physical database tables. You may abstract logically discrete events from physical database transactions or units of work.

On abstraction

Three abstraction tools can be used to raise the abstraction level of a model.

- Generalisation means creating a super type. This can save us from having to know about several subtypes. One might look to the OMG for a super type programming language, or a super type platform or operating system. (By the way, does a super-type platform help us to build a PIM? Or does it remove the PIM-PSM distinction?)
- Suppression of detail means delegating elaboration to another person or a machine. This can save analysts, designers and programmers from much tedious effort. E.g. we have long worked with platforms that automate the functions of data storage, indexing, sorting and transaction management. Suppressing the detail of transaction roll back is important in the conclusions of this paper.
- Composition means grouping details and hiding them behind the interface of a larger component. This enables people to manage large and complex systems. This has been a tenet of most if not all analysis and design methods since the 1970s.

These three devices are often combined and entangled in practical software engineering. Consider a class library whose classes offer general infrastructure operations that form or extend the platform and enable us to suppress detail from our applications. But it is useful to recognise them as three separable ideas.

On business rule specification

Business rules embrace business terms, facts, constraints and derivations. Terms are the names of things; they appear in software as the name of entities and their attributes, of events and their parameters. Facts are relationships between things; they appear in software as pointers, foreign keys and message-passing interactions. Constraints appear in software as data types, domain value ranges, relationship multiplicity constraints and validation tests. Derivations appear in software as the calculation of a data item value, the sub division of one data item, the concatenation of several data items.

Business rules of all these kinds appear in data structures and processes at every level of software engineering, from user interfaces to databases. They might be specified as preconditions and post conditions of processes at any level of software engineering, in conditions governing a step-to-step transition in a business process, and in conditions governing the commit/rollback of a database transaction.

Forward and reverse engineering transformations

A good way to explore the meaning and usefulness of the three levels of model in MDA is to consider possible CIM<->PIM<->PSM transformations, while recognising that these transformations may never be fully automated.

Automated transformations sell tools. Transformations that require human intervention are no less interesting, and they sell training courses. Of course we look for automated support wherever it is possible. But most of the transformations that I am interested in require some human intervention.

When people transform a model at one level into a model at a lower level or higher level, they often call this forward engineering or reverse engineering. Reverse engineering is a process of abstraction; it abstracts by suppression of detail, generalisation, or composition, or a combination of those three devices. Forward engineering is a process of elaboration; it elaborates by adding detail, specialising or decomposing, or a combination of those three.

For human beings, reverse engineering is infinitely easier than forward engineering. It is easier to remove detail than to add it. It is easier to generalise from than to create specialisations. It easier to group details into a composition than to detail the members of a group.

MDA brings the possibility of two reverse engineering transformations (PSM-to-PIM and PIM-to-CIM), and two forward engineering transformations (CIM-to-PIM and PIM-to-PSM). I will discuss all four, though intending to focus on transformation from the highest level of model, the CIM, to a PIM.

PSM-to-PIM reverse engineering

PSM-to-PIM transformations have been around for decades. Take a database schema, erase some of the DBMS or platform-specific details and you can express the database design as Bachman diagram. Erase some more detail and you have an entity-attribute-relationship model (aka data model).

This illustration shows that there are degrees of platform independence. Abstracting upwards from a PSM, there is not one level of PIM but many. And even at the first level of abstraction, we may choose to abstract in different directions, so there are potentially many branches as well as many levels of PIM.

Most of the people interested in MDA are interested in process structures more than data structures. (Indeed, some in the data management community are yet to be convinced the OMG or MDA are relevant to the issues of data management.)

Some are interested in abstraction by generalisation of coding languages. Where the PSM is a model of Java or C++ code, then the PIM could employ a more generic OOPL. An even higher level PIM could abstract between a generic OOPL and a procedural language like COBOL. But what MDA may deliver by way of a programming language is likely to be more complete rather than more abstract. Stephen Mellor has said:

> "What UML calls a computationally complete "action language" will have at least the following features:

- complete separation of object memory access from functional computation. This allows you to re-organise data and control structure without restating the algorithms--critical for MDA
- data and control flow, as opposed to purely sequential logic. This [enables you to] distribute logic across multiple processors on a small scale (e.g. Between client and server, or into software and hardware)
- map functions and expansion regions that let you apply operations across all the elements of a collection in parallel. This … maximizes potential parallelism, again important for distribution, pipelining, and hardware/software co-design.

> While not huge linguistic advances, these properties enable translation of complete executable models into any target. In my view, that is the key reason we build models of the more-than-a-picture kind."

I am less interested in turning UML into a more complete programming language than in building models that abstract by suppression of infrastructure detail. E.g. where the PSM defines all the details of transaction start, commit and rollback processes, then the PIM can be a model that is very much simpler because it includes only hooks for these platform functions.

Hmm … that last so-called PIM contained hooks for the transformation to a PSM. So it is not purely platform-independent, it posits the existence of a platform with transaction start, commit and roll back functions. I think this particular postulation is vital to the making of a CIM that can in practice be related to a PIM. I will return to this later.

Aside on true platform-independence

On the one hand, we want to build platform-independent models. On the other, we want those models to be transformable with minimal effort into software systems. The trouble is that software systems can be implemented in many ways and using many technologies. So, to ease forward engineering, people do in practice model with their chosen technology in mind.

A model that somebody claims to be a PIM may be more tied to a specific platform than the claimant recognises. When building a PIM for coding in C++, does the modeller ask: Would I draw this model the same way if we were to code in Java? And when building a PIM for coding in either C++ or Java, does the modeller ask: Would I draw this model the same way if we were to code in PL/SQL? Or VB.Net?

The meta model underlying UML looks, at its heart, to be a model of an object-oriented programming languages. If we want a truly universal modelling language, designed to model a truly platform-independent model, then the meta model might need some revision. I will come back to this later.

PIM-to-PSM forward engineering

It is possible to reverse the abstraction examples discussed above, and to employ a tool that will automate some of the forward engineering elaboration. This kind of PIM-to-PSM transformation dominates many people's view of MDA. So much so that one wag round here renamed MDA as MDCG (Model-Driven Code Generation). Allan Kennedy has, in an OMG discussion, defined MDA thus

> "In a world where MDA is the dominant development paradigm, all that most developers will work with is an MDA development environment supporting executable UML as the new whizzy programming language supplemented by a number of commercially available model compilers for popular platforms.
>
> Platform specialists and software architects will work with tools for building custom model compilers which might even be based on whatever emerges from the current QVT process. The need for the majority of developers to fill the 'gaps in their IT knowledge' will have been eliminated by the move to "platform-independent" UML as the abstraction level for specifying system behaviour."

Generation of lower-level code from a higher-level language is often presented as being an unquestionably good thing. The presenter may put down a challenge from the audience by reminding us of the luddites who wanted to continue coding in Assembler after COBOL was introduced.

But the resistance of those luddites faded in the 197Os. Several attempts were made in the 1980s and 1990s to move up from the level of COBOL. Several 'application generators' were sold on the basis that you could write in a general 'business rules' language, and from that generate either COBOL or C. Veterans bear the scars of these attempts.

(I ought to exclude here data model driven tools like Visual Data Flex that, when used with a data dictionary that captures the business rules, can be good for generating business data maintenance applications with relatively simple graphical user interfaces.)

Yes, we can forward engineer by specialisation. We could build a PIM using a generic programming language and/or assuming generic platform infrastructure, then transform this into Java or C++ or C or COBOL using platform-specific infrastructure. But it is far from obvious that the benefits outweigh the costs and risks.

Portability benefit? In practice, portability between programming languages or platforms is rarely a requirement you can clearly establish up front. Then, trying to anticipate the requirement can cost more than meeting the requirement if and when it arises. Transformation between closely related languages (e.g. dialects of SQL) costs little – probably less than efforts to anticipate the requirement. Transformation between very different languages (e.g. COBOL and Java) is, as far as I know, either impossible to anticipate or counter-productive because it denies the programmer opportunities to use the very features the language was designed to offer.

Productivity benefit? I recall an application generator salesman making a sale on the basis that the tool's 'business rules' language was up to 50 times more concise than COBOL. But actually, he compared the source code with the tool's *generated* COBOL. His business rule language was no more concise than COBOL; his generator simply wrote clumsy, long-winded code (be it COBOL or C).

It has previously turned out that the benefits promised by code generators (with the possible exception of data model driven tools when used for appropriate applications) were outweighed by the costs and risks listed below:

- forward engineering tools generate clumsy long-winded code that is less efficient, sometimes too inefficient to meet non-functional requirements.
- you become dependent on the vendor of a niche-market tool
- you become dependent on relatively scarce programmer resources, people who know the code generator's specific language, and other product-specific features
- when things go wrong, you have to refer to the generated code anyway, meaning you remain dependent on programmers who understand that level also
- the code generator inserts invocations to infrastructure services when and where you don't want them

A colleague, while enthusiastically proposing we try a specific MDA tool/product, added the rider that "you need to be a Java, J2EE, struts, UML, MDA and product expert to properly leverage the product." How do we find or train these people? Veterans will need a lot of convincing that mainstream projects should use a tool that requires designers to understand all that, the MOF (Meta Object Facility), and tool-specific patterns and transformations.

Veterans, listening to a presentation on MDA tools, are likely to worry about the potential costs and risks above.

A kind of forward engineering that yields real productivity benefits is based on suppression of detail. We don't want programmers having to model or write code that a general-purpose machine can do for them. So we look to automate forward engineering by getting a machine to elaborate, add detail, add generic infrastructure.

e.g. PIM-to-PSM transformers that add in the detail of platform-specific transaction management or database management functions enable us to limit our modelling effort to more business domain-specific concerns.

Having said that, IT veterans have been modelling and coding in ways that assume the support of transaction management and database management functions since about 1980. So marketing PIM-to-PSM transformation tools on the grounds that they add functions of this kind can raise something of a wry smile.

PIM-to-CIM reverse engineering

Two kinds of CIM have surfaced in OMG discussions. The first kind of CIM is a model of a business enterprise, a stand-alone CIM, independent of data processing and of potential software systems. A purely conceptual or domain model of this kind is interesting per se. It can be used to define some business rules. But forward engineering transformation is problematic. In my experience, the majority of software engineers do not find such models helpful when it comes to practical software projects. We do sometimes need to steer systems analysts away from paralysis by analysis and towards defining a CIM that is useful to software system designers.

The second kind of CIM is definitively related to one or more data processing systems. It can be transformed into software systems that consume input data and produce output data. Such a CIM may be thought of as a very abstract PIM. And given there are degrees of PIMness, there must surely be degrees of CIMness. As one person's PIM is another person's PSM, so one person's CIM may be another person's PIM.

It is always possible to abstract upwards or backwards. And again, reverse engineering is infinitely easier than forward engineering. Some people focus on abstraction by generalisation of variant forms into a common or shared form. But I propose abstraction by composition and suppression of detail will prove more profitable.

We can envisage abstracting one CIM from one PIM. This focuses the CIM on the domain and requirements of a single data processing system. We can then realistically ponder the forward engineering transformation from CIM to PIM. It might be interesting to explore this, but my concern here is to focus on the problem of the large enterprise with hundreds of distributed and loosely-coupled data processing systems.

We can construct one CIM by abstraction from many application-specific PIMs. We can use abstraction not only to reduce the number of business rule variations (by generalisation), but also to reduce the number of rules (by composition and suppression of detail). For example:

- If one application's PIM includes an EmailAddress (must include an @ sign) and another application's PIM includes TelephoneNumber (must be numeric), then we might define in a CIM a more generic ContactDetails item with a more generic data type.
- If one application's PIM includes an orderValue formula that calculates sales tax one way and another application's PIM includes orderValue formula that calculates sales tax another way, then we might define in a CIM a simpler orderValue calculation that suppresses the detail of tax calculation altogether.
- If a CustomerAddress has 3 lines in a regional application, has 5 lines in a global application, and has a structured set of attributes in another application (that uses name, town and postcode for other purposes), then we might define in a CIM a single composite CustomerAddress data item.

Still, we have to face two awkward questions about the enterprise-scale CIM.

First: How to resolve the million-rule problem? The large enterprise has hundreds of applications and a million business rules. We cannot maintain an enterprise CIM with that many rules. This has classically led enterprise ‘data architects’ (really, ‘data abstracters’) to define an abstract data structure containing a few generalised entities such as party, contract, place and event. They define for each entity a few attributes that appear in several applications. They may perhaps define a few business rules associated with those few attributes. Similarly, enterprise process architects have defined abstract business processes, each with a generalised sequence of business process steps such as register, authorise, process, deliver and close.

In practice, I haven’t found people making good use of such a highly abstract enterprise-scale CIM. How to separate the business rules that are somehow most essential or important from the impossibly vast multitude of necessary business rules? I don't see people successfully grappling with specifying business rules in an enterprise architecture (in the sense I mean enterprise, that is 'enterprise-scale', rather than simply 'business level') other by being highly selective, by focusing on only a tiny part of the enterprise problem domain.

Second: How to resolve the loose-coupling problem? The large enterprise works with many distributed and loosely-coupled systems in which different, perhaps conflicting, business rules apply. The enterprise's business processes have to work despite the fact that data in discrete data stores *will* be inconsistent. Surely an enterprise CIM (if it is to be useful for forward engineering into more than one PIM) must acknowledge that consistency cannot be guaranteed

across all the discrete business data stores maintained by the enterprise?

Wherever the infrastructure does not exist to roll back a mistaken process across discrete data stores, then we have to design all manner of error handling and undo processing, and our models of the code have to incorporate all this design.

Where the infrastructure does exist, where we know a transaction can be automatically rolled back, we certainly don't want to model the error handling and roll back processes by hand. We can/should/must suppress the roll back details from our abstract models. Surely we can do this only by employing the corresponding abstract concept of a "unit of work" or "discrete event" in our models?

CIM-to-PIM forward engineering

I propose we cannot realistically envisage forward engineering from a purely conceptual CIM. We can however envisage forward engineering from a CIM that abstracts from data processing systems, and we can recognise this kind of CIM because it will:

- acknowledge the divisions between data in discrete loosely-coupled data stores
- define what units of work clients invoke or require on each distinct data store, with the preconditions and post conditions of each unit of work
- define what data must persist in each discrete data store for those units of work to be completable.

To put it another way: whatever paradigm you follow or platform you use, to build model that can be transformed into a data processing system, you must answer two requirements-oriented questions:

Q1) what units of work do clients invoke or require? A "unit of work" is a service. It is a process that acts on persistent data, or, if the necessary conditions are not met, it does nothing but return/output a failure message. A "client" could be a user, or a user interface, or an I/O program, or an actuator or sensor device.

Q2) what data must persist in a coherent data structure for those units of work to be completable? Every software system of note maintains some persistent data. The data structure could be anything from a handful of state variables representing the state of a few devices in a process control system, to millions of business database records representing the orders by customers for products.

In specifying the business rules of software systems, the persistent data structures and the units of work on them are fundamental. Whether your coding language is Java or PL/SQL, you will have to specify them.

Conclusions and remarks

How can MDA meet our two apparently contradictory challenges – to abstract from detail and to comprehensively specify business rules?

What kind of detail is best suppressed from a model?
A good way to keep a model simple is to postulate that a process can be rolled back automatically. This means we can ignore the design and specification of the backtracking needed when it is discovered that a processes' precondition has been violated. Indeed, it would be futile to model undo processing where we know our platform can automate the roll back of a process.

What level of granularity is best for modelling business rules?
A high-level abstract business process model can be recursively decomposed many times before we get to executable code. At which level of granularity should business analysts specify business rules, given we want these rules to be coded more or less directly from the specification?

The components and processes of a PSM must be defined down to the level of granularity dictated by our target

programming language and platform.

The best level of granularity for a PIM or CIM is more open to debate. I propose we have to model the components and processes of a PIM with some minimal knowledge of the target platform's transaction management capability. We should know and declare two kinds of platform-related information in a PIM:

- the units of system composition - the discrete systems across which the chosen platform can automate roll back of a process - a discrete system has a discrete structural model and often maintains a discrete data store
- the units of work on each discrete system - the roll-backable services offered by each discrete system

People sometimes try to capture business rules by documenting the preconditions and post conditions of a use case. Often they get this wrong, because they specify for the whole use case what are rightly the preconditions and post conditions of one or more discrete back-end services. See FOOTNOTES below.

More controversially, I propose we may have to model the components and processes of a CIM with the same things in mind. We have to recognise discrete system boundaries. We have to model discrete events as well as discrete entities. We have to work on the assumption that discrete events can be automatically rolled back. At least, we have to do these things if we want the CIM to be readily transformable into a PIM. We cannot hope to do forward engineering from CIM to PIM if we cannot envisage the former as a reverse engineered abstraction of the latter.

How to capture all the essential business rules in a model?
An entity-oriented approach, defining a structural model, seems the natural way to analyse and specify invariant business rules. Some have proposed that a CIM should be defined using *only* a structural model. But if we want to model *all* the business rules, then this way lies the madness of defining every unit of work as an entity type and elaborating the model to include history of every attribute value over time.

So, an event-oriented approach, defining a behavioural model, seems the natural way to analyse and specify transient business rules. And to specify these business rules, we should define the preconditions and post conditions of processes at a specific level of granularity - the unit of work - the level where we assume roll back can and will be automated by the given platform.

(We can specify preconditions and post conditions for processes (say use cases) at a higher level of granularity than the unit of work. But most business rules belong at the unit of work level, since units of work act directly on stored business data, and one unit of work can be shared by several use cases. We can specify preconditions and post conditions for processes (say operations) at a lower level of granularity than the unit of work. In fact we can recast every condition and action within a unit of work as a precondition or post condition of a lower-level process. But let us not confuse the work of programmers, who have to work at the lowest level, with the work of analysts who must specify the business rules at the level users are conscious of.)

How to make UML more helpful to analysts looking to build a PIM or CIM in the ways indicated above?
Do we want a truly universal modelling language or method? Are we serious about building truly platform-independent models? Do we want to capture requirements in models? Do we want to help systems analysts document what matters?

I propose that units of system composition (discrete systems) and units of work (discrete events) should be first-class concepts in the UML meta model, not merely stereotypes of 'class' and 'operation'.

To define a forward engineerable CIM, we have to define the persistent data structures, the units of work on those data structures and the transient rules (preconditions and post conditions) of those units of work. Why?

- we have to capture what domain experts understand of how persistent data constrains processes and is changed by processes
- an enterprise's data is distributed, and it is vital to define which data stores the business requires to be consistent and which data stores need not be consistent
- business people should understand the effects of the units of work that they invoke from a system's user interface
- if we define invariant rules in a structural model, and postpone defining transient business rules to a lower level of design, then we are simply overlooking an important set of the business rules

- if we don't build a model on the assumption that unit of works can be automatically rolled back, then we are forced to model all manner of complexities, error handling and undo processing.

Its is difficult to teach event-oriented analysis and design techniques within the context of an OO methodology. UML does not include the unit of work. OO models contain operations at every level of granularity, and roll-backable operations are not marked out. People teach instead fuzzy concepts like the “responsibilities” of a class or component. For me, this is a limitation of the OO paradigm as currently taught. I want people to take both object and event-oriented views equally seriously - at least during the building of a CIM or a PIM.

I propose we teach people that event-oriented unit-of-work-level services are fundamental analysis and design artefacts, as important in an OO design as the entity-oriented components. We should teach that it is a good idea to identify the roll-backable units of work that clients invoke or require, and consider the effects of each unit of work on the entities in the persistent data structure. This analysis should reveal to OO designers the responsibilities of entity classes and the business rules that operations must apply to objects.

If the OMG truly wants UML to be a truly universal modelling language, then making units of work explicit in the UML meta model might help. This will make for a more complex meta model, since one unit of work may trigger operations on many entities, and the effect of one unit of work on one entity can involve more than one operation, but it will also make for a more universal meta model, one that embraces event orientation and object orientation as equals.

How to build enterprise-scale models?
We should not pretend an enterprise is a single coherent system, or is supported by a single coherent data processing system. We have to model a large enterprise as a set of discrete systems, with potentially conflicting business rules. Then we have to model each system model with much abstraction.

If we are to work at the highest possible level of abstraction, reduce the number of things to be modelled, produce the most concise specifications, then we must maximise the scope of the discrete systems we regard as units of composition, and maximise the size of the discrete events that we regard as units of work on those systems. How to maximise the size of discrete systems and discrete events is beyond the scope of this paper.

If we are to reduce the number of business rules to be modelled, then we have to further suppress detail somehow. Specifying rules using one or more derived data items is one way to hide the elementary input and/or stored data items. Some detail of a derivation rule can be suppressed by defining it using a derived data item calculated from lower-level data items. Some detail of a constraint rule can be suppressed by defining it in terms of a derived data item (e.g. and most fatuously, a "PreconditionsMet" boolean) that sums up the result of lower-level processes.

It is hard to think what abstraction devices to use beyond this, other than arbitrarily omitting business rules that we intuitively regard as unimportant, and using informal rather than formal syntax.

Finally: Is CIM-to-PIM-to-PSM a sensible basis for a software development methodology?
I fear MDA has confused in one scheme modelling the real world per se with modelling a data processing system (which is itself a model of the real world). The two are related, but nobody I know in the IT industry looks to define a PIM from a purely conceptual CIM. They define a PIM from a statement of data processing system requirements. And these requirements are better expressed in terms of use cases and input and output data structures, rather than in the form of a CIM. Inputs and outputs are an aspect of system theory that MDA seems, curiously, to have overlooked.

FOOTNOTES

On design by contract

I have no space here to discuss Bertrand Meyer's “Design by Contract”, but let me suggest that the opposite strategy of “Defensive Design” is better for multi-user database systems.

On why use cases are not enough

Extremist disciples of the distributed object paradigm and the relational database paradigm take a surprisingly similar view of their task. They both envisage building general-purpose components (be they distributed objects, web services, or databases) that sit there waiting. Waiting for clients to find them and make use of them. Waiting for requirements they can service. Waiting for their general services to be extended with additional features or specific variations.

It is a good thing in system design to anticipate future requirements a little and to generalise for future clients a little. But since we have to deliver systems to time and budget, our analysis and design method has to emphasise and prioritise the known requirements of current users. For this reason, most systems analysts nowadays define something akin to use cases during requirements analysis.

Use cases are far from object-oriented. They are procedural. They are also outward facing, user-task or HCI oriented. They define the flows of control that govern what users do at the user-system interface. Conventional use case definitions contain very little of what is needed to develop or generate code. We are likely to need also:

- the UI design, its appearance, fields and commands
- the I/O data structures (an XML schema-like grammar or regular expression notation might be useful for serial data flows)
- the state date of a user session (which may be stored on the client machine)
- any state transition constraints on the user session (state machine notations may be used here)
- the preconditions and post conditions of each *unit of work* that is invokable.

Use cases involve, or better, invoke, units of work. This isn't functional decomposition so much as client-server design. You can think of use cases and units of work as being arranged out-to-in rather than top-down.

A use case is a usage of a software system to facilitate a task in a business process - typically a one-person-one-place-one-time user-system dialogue or function – or a process to consume or produce a major data flow.

A unit of work is a process that is a success/commit unit, usually acting on a coherent data store. You might call this a "service use case". My company calls it a "business service". But I call it a "unit of work" here because this term implies roll-backableness, and that is essential to the concept I am promoting.

Other challenges facing MDA

There are many practical obstacles to successful forward engineering from one level of MDA to the next. Some are mentioned above. Other barriers to MDA (mostly suggested to me by Chris Britton) include:

- Existing systems: current tools generate UML from code - not much help really. Are there tools to reverse engineer PIMs and CIMs from legacy systems?
- System integration: how to build CIMs and PIMs for message brokers and adapters?
- Verification: how to verify models than have not yet been implemented?
- Abstraction from the distributed object paradigm: aren't user requirements essentially event-oriented rather than object-oriented? Aren't outline solution components (subsystems) really rather different from programming level components (DCOM objects, whatever)?
- Non-functional requirements: these constrain the results of a PIM-to-PSM transformation
- Primitive data types: how to define basic or generic constraints on data item values in a CIM or PIM?

On reuse

Use cases and units of work are not products of object-oriented analysis; they are products of event-oriented analysis. And after identifying what use cases and units of work are needed, an important task in detailed design is to optimise reuse.

Every discrete software system can be defined as a process hierarchy (though hierarchy here means a network rather than a strict hierarchy, since a lower-level process can be part of several higher ones). e.g. A use case may involve zero, one or more units of work, and a unit of work may be reused in several use cases.

You can also factor out common processes at one level. You might find two use cases share a common process, or two units of work share a common process. Sometimes, that common process is wanted on its own in another context, so it can be defined as discrete use case or unit of work. The lowest level common processes in units of work are operations on the lowest level encapsulated entities.

There is a formal event-oriented technique for defining reuse between units of work. In this technique, the unit of work is called a discrete "event" which has an “effect” on each of one or more “entities”. Two discrete events can share a common process, known as a "super event". The OO concept of a responsibility is akin to an effect, or more interestingly, to a super event.

In short: You identify events. You identify where two or more events have same preconditions and postconditions wrt an entity (that is, the several events appear at the same point in the entity's state machine and have the same effect). You name the shared effect as a super event. You analyse to see if the super event goes on from that entity (where the events’ access paths come together) to have a shared effect on one or more other entities, and if so, you adopt the super event name in specifying those other entities’ state machines.

I don't mean to promote this specific “super event” analysis and design technique. I am only wanting to indicate that event-oriented analysis and design has a respectable and successful history, since many OO designers are unaware of this history.

By the way, you may be able to generalise two similar units of work into one, but you have to define the two distinct requirements before you can know this is possible.

On practical experience of using UML as the programming level

“On some previous projects, I went full-scale for using Rose to produce detailed models and generate code from them. When I needed to extend the design, I would always return to the Rose model and make the change there, forward generate the code, and then fill in the details. I worked for several years this way and advocated for it strongly among my peers, although few took me up on the approach, at least in part because of the steepness of the learning curve for Rose with code generation in C++.

The models I produced were useful to me, but far too detailed to be helpful for talking about the design with someone else. I had to produce more simplified views for this, but I was proud of the fact that my models always represented the state of the code.

Unfortunately, maintaining a very detailed model is really no easier than maintaining the code. My models tended to become rigid, even though I was working with UML diagrams. Finally, I have decided that working with too detailed a model is a trap. Basically this is the same trap that we were trying to avoid by working with UML in the first place.

It's very important to work at the correct level of abstraction.

In my current project, we have taken the simple design and refactoring approach. We develop in three week iterations. We draw the designs we need for each iteration on a whiteboard. When we all understand them well enough, we code them. We refactor continuously, driven by code smells and design considerations. At the end of an iteration, we reverse engineer our code (using Rose) and produce simplified views that we use to inform our designs on the whiteboard in the next iteration.

This seems to have worked very well, and I feel our design is quite good. Partly, this represents that I have more experience as a designer.

However, I also see that this way of working keeps us focused on the right level of detail at the right time. In other words, a white board can be a more effective tool than Rational Rose for working out a high level design, and refactoring using the code can be a very effective tool for improving a design. The reverse engineering works well enough and the model is always up to date. We have a team of three that has been working for 15 months in this way, and the design and code have not become rigid. I won't be going back to using forward code generation from a model.”