Sims Architectures

Enterprise MDA®

or

How Enterprise Systems Will Be Built

Author: Oliver Sims

Version: v01-02

Issue Date: July 27, 2004

File Name: Enterprise MDA v01-02.doc

Word Count: 6008

Success for MDA in the longer term depends on its ability to bring compelling value to enterprise IT. Neither UML® nor executable UML are currently major players in enterprise IT. They risk becoming also-rans if their potential fails to be realized in the enterprise context, both in developing new applications and, arguably more important, in the integration and interoperability of existing systems. In addition, MDA must inter-work with other present and emerging standards, such as web services, business process management (BPM), and JCP (Java Community Process) initiatives.

This paper first analyses the OMG's stated goal for MDA, a goal that is well beyond the oft-heard talk of model transformations and code generation. It then describes how a small number of disparate but important advances in the industry can be brought together in a truly synergistic way to achieve that goal. Lastly, a roadmap for reaching the MDA goal is suggested.

1 MDA - What's it all about?

MDA often appears on the surface to be all about transformations between four different kinds of model—a CIM, a PIM, a PSM,[1] and code (code being a model of the run-time). Often the code generated is thought of as being restricted to skeleton code, plus some glue code. And if this is all it is, then it would certainly not live up to the OMG's claims. What claims? Well, the home page of the OMG's MDA website (http://www.omg.org/mda/), under the heading "*How systems will be built*", presents what might be called the MDA vision:

> *MDA provides an open, vendor-neutral approach to the challenge of business and technology change. Based firmly on OMG's established standards, MDA aims to separate business or application logic from underlying platform technology. Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA, J2EE, .NET, and Web Services or other Web-based platforms. Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution, and further enable interoperability. In addition, business applications, freed from technology specifics, will be more able to evolve at the different pace of business evolution.*"

Why will systems be built this way? Because achieving MDA's aims can radically reduce development, maintenance, and evolution time, provide for enhanced flexibility, and can bring the solution design much closer to the problem definition.

But how can MDA do this?

The clues lie in these key parts of the vision statement: "separate business from technology", "enable IP to move away from technology-specific code", and "fully-specified PIMs (including behavior)" on "a range of platforms". This means that a PIM that includes behavior specified using an action language could either be the basis for generation of 100% of code, or could be directly executed (or more accurately, interpreted).[2] In the enterprise system context, "platform-independent" means independent of platforms such as J2EE, CORBA, .NET, various databases,

1.1

[1] CIM: Computation-Independent Model (often referred to as a business or requirements model); PIM: Platform-Independent Model; PSM: Platform-Specific Model.
[2] There is a valid argument that a PIM that is executable is actually a PSM that is specific to whatever engine handles the execution. However, in the context of enterprise systems, where "platform" generally means the commercially-available middleware, we prefer retaining the term "PIM" for a model that can be executed on two or more of those platforms.

various GUIs, and so forth. Clearly achieving this aim would be of extreme value! Even a partial attainment of the goal could be highly attractive.

But what factors can enable the aims to be attained? MDA alone cannot do it all. MDA's scope is certainly larger than simply defining kinds of model and the relationships among them: it is based on the sound foundation of MOF™, such that languages other than UML can be used, and important inter-language capabilities can be managed automatically. However, MDA certainly does not include all the various aspects needed for scalable, flexible, service-oriented, and interoperable enterprise systems. So MDA must assume other factors that support its central vision. What are these factors?

To answer this, let's re-state the key parts of the MDA vision in terms of questions:

1. How do we separate the business logic from technology logic?
2. What is a good structure for generated code, and therefore of a PIM, such that the necessary 'ilities (scalability, flexibility, service-orientation, etc.) are delivered?
3. How do we ensure that the "business" in the business logic actually relates well to the business in the business?

Happily, there are good answers to these questions. Briefly, they are:

1. A **product line approach** can separate business logic from technology logic.
2. Effective **architecture**, preferably component-based, can be delivered to application developers through UML profiles and Domain Specific Languages (DSLs)[3].
3. An approach to **bridging the "Business/IT Divide**" that provides a good CIM-to-PIM transformation.

And even more happily, these three answers tend to be very synergistic. Indeed, some have argued that to be successful in addressing enterprise IT challenges with one, you have to address the other two as well. Of course, other factors such as process and organization must also be addressed; however, implementing the three approaches just listed tends to lead inexorably to others, so they're not forgotten. Meanwhile, let's consider the three approaches and how they contribute to MDA goals.

2 Implementing the Goal – MDA's Enterprise Companions

2.1 Separation – a Product Line Approach

MDA "aims to separate business or application logic from underlying platform technology." Business logic can be defined as any development artifact, or part of an artifact, that is unique to the business. For example, code that calculates a price, or a screen definition that defines how a sales order record should appear, or a data schema for customers, are all "business logic". On the other hand, code that sets up the necessary conditions for a web service to be invoked, or handles the window creation mechanism at the GUI, or manages an ACID transaction, is "technology logic."

1.1

[3] DSL stands for Domain-Specific Language. DSLs have been addresses in previous MDA Journals (see [5][10])

Ideally, the business application development team—whether working on new applications or integration projects, whether outsourced or in house—should be concerned only with business logic, everything else being consigned to a COTS (Commercial Off The Shelf) “platform,” as shown at the top part of Figure 1. However, such a COTS platform does not exist today. Although much technology logic is, of course, handled by COTS products (middleware, operating systems, DBMSs, EAI managers, and so on), there is always a gap between the platform provided by a given set of COTS products and the business logic within any given development project (“platform gap” in Figure 1).

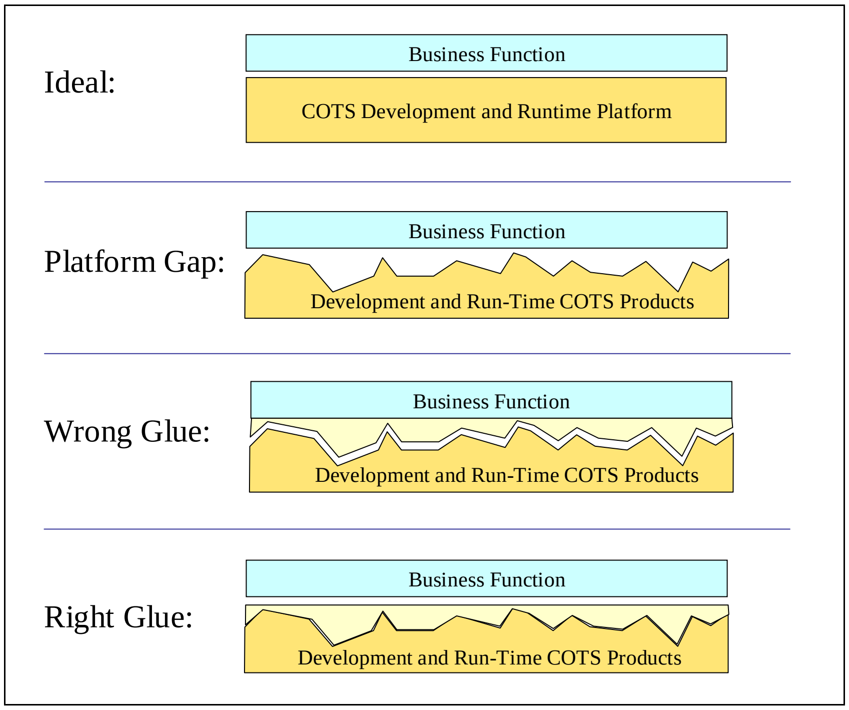

Ideal:
Business Function
COTS Development and Runtime Platform
Platform Gap:
Business Function
Development and Run-Time COTS Products
Wrong Glue:
Business Function
Development and Run-Time COTS Products
Right Glue:
Business Function
Development and Run-Time COTS Products

Figure 1

The gap comprises both answers to “how to” questions, and specific artifacts that deliver the answers pre-packaged for business developers. Examples include how to handle concurrency, transactions, transparency across different communications stacks, and pooling of various sorts (threads, DB connections)—as well as how to integrate the various COTS products. There are also how-to questions about the development environment (itself a fairly complex IT system) such as how to share models effectively, transform models, structure models and code, and define and collect useful metrics. Finally of course, there are the big issues: how to provide for flexibility, scalability, re-usability, and so forth.

A specific example of an artifact that helps “fill the gap” is a client-side proxy that enables a business developer to invoke another component either synchronously or asynchronously with respect to his or her thread of control, providing a call-back method/operation for an async reply.[4] This isolates the developer entirely from the nature of the underlying communications stack. Another example is a script that enables a model to be transformed into another (for example, from CIM to PIM).

1.1

[4] Experience suggests that most of the time invocations are made synchronously. But occasionally async invocation is needed—and when it’s needed, it’s really needed!

The gap is often informally filled either by all business application developers learning a great deal about software technology, or by a few expert software technologists within a project team or shared across project teams. However, the filling—often called "glue"—is seldom captured and re-used by other projects. Hence the technology efforts are often duplicated, and often projects run late because the technical skills required to fill the gap are underestimated. In summary, filling the gap on a by-project basis is the wrong way to provide glue ("wrong glue" in Figure 1).

Platform vendors may argue that the gap is necessary, since they must provide for a very wide range of customer requirements. This is true as far as it goes; however, such platform vendors do not seem to have taken on board the fact many applications—or integration projects—are often technically incredibly similar to each other. This similarity is what makes product lines feasible. Technically similar systems are often said to conform to the same "architectural style" [1] or "approach" [2]. Sometimes the similarity derives from similarities in the business area being addressed, and sometimes from similarities in the technical approach to solving quite different business problems. In any case, systems built to the same architectural style have very similar glue requirements. The observation that many systems are technically similar is at the heart of the product line initiative. The product line concept [3] has been summarized in previous MDA Journals [4] [5], and there have been other proponents of the same approach, albeit under different names, including Herzog [2] and Hubert [1].[5]

Indeed, platform vendors could find it profitable to provide for the common architectural styles of enterprise systems ("right glue" in Figure 1). Currently, however, this glue must be provided by the IT organization; and providing it informally within each project is a huge waste of corporate resource. A product line approach enables the glue to be captured and used (or re-used if you prefer) for multiple projects.

Separation of business from technology logic can be done in two ways:

- Provide the glue as an addition to the COTS runtime, already deployed, and treated as an in-house addition to the run-time—for example, a logging service.
- Generate the glue code each time it is needed by a project, and deploy it with the application—for example, code to log to a common logging database.

In general, one should generate as little code as possible within a given project. The reason for this is that the more is generated, the more has to be tested. Imagine that much of the technology logic in a logging service were to be generated for each application. A change to that generated logic would require that all the applications that embed it will have to be re-tested and re-deployed. Some might say that the supposedly unchanged business logic does not have to be re-tested; others would be more cautious, and say that re-compilation or even re-link must be re-tested even though a large part of the source code has not changed (as far as anyone knows!).

In reality, both approaches will be needed. However, it is clearly much better, wherever possible, to produce glue once, and to deploy it once, thereby effectively creating a higher-level platform. Since "platform" normally refers to COTS products, I apply the term *virtual platform* to the combination of COTS products (middleware, GUI frameworks, DBMSs, application servers, etc.) and glue.

1.1

[5] The product line concept is not new: a good argument can be made that the billions of lines of mainframe COBOL code that ran the world's businesses in the last third of the last century were often produced in a similarly-structured environment.

Separating application development tasks and artifacts from platform tasks and artifacts, however, can take a significant management effort. Standardizing on a given set of COTS products is something that many enterprises already do. However, it's much better also to rigorously separate as much of the "glue" as possible. This leads to an organizational structure whereby a "platform" (or "infrastructure") group has as its mission to provide as many transparencies as possible for a separate application development group. Making that organizational change is often not so easy. The higher-level virtual platform is created and maintained by a platform group, whose mission is to "delight" (as one of my clients put it) the application development group. Application development projects are run within the latter group. In this way, business change and evolution is separated from that of technology.

Now that we've got the business logic as fully separated as possible from technology "stuff," we can now consider how MDA applies to business logic by itself. But a "fully-specified PIM", with action language providing behavior, should in principle be executable. And executability requires more than just the business logic: it also requires a specification of the structure being executed—especially so if the target platform is a distributed system. So: how should the model be structured—and how should the generated code be structured?

2.2 Enterprise PIMs

It is relatively easy to answer the question of how code generated from an enterprise PIM[6] should be structured. It should, of course, be structured following best-practice modularization (e.g. high cohesion low coupling), where modules interact so as to deliver scalability and to minimize dependencies for flexibility. It must also conform to the virtual platform so that viable code can be generated. Best-practice modularization means taking a mature computer-based software engineering (CBSE) approach. By "mature", I mean the hard-headed software engineering approach as opposed to the "let's buy everything off the shelf and mix-n-match to magic a solution" approach.

2.2.1 Structure and Architectural Style

But how should the PIM as a whole be structured? Arguably the best way is to structure the PIM according to mature CBSE as well. Since each component encapsulates and realizes a specific business concept, it becomes fairly clear where the business logic fits. The UML2 component provides an ideal model element for this approach, since it "addresses the area of component-based development and component-based system structuring, where a component is modeled throughout the development life cycle and successively refined into deployment and run-time." [6]

In a product line approach, the question of technical structure is handled by the platform group—or by a separate architecture group (possibly a third IT organizational element). Such structural design is usually called an "architecture", and is the expression of an architectural style. Of course, there are other forms of architecture; for example, of particular use is a "business" architecture that provides business patterns—for example, the pattern for design of a Contract, perhaps as suggested in [7].

Structure—including allowable interactions between components—should be defined by the platform group, and delivered to the application development group in the form of a tested UML

1.1

[6] I say "PIM" singular: but of course a large system may be designed within several models, each at the same level of PIM-ness.

profile. The profile also defines how a PIM is packaged, so that, for example, a design-time can be taken as a whole from a repository and plugged into (re-used in) a PIM. Indeed, there is no reason why, as soon as the component is identified, it should not become "executable" within the development environment, so that it can be queried, for example, as to its development status, and also be able to run whatever simple test cases may be available: for example, create an instance which has a unique key. This approach to "living components" is well described in [1]. It is particularly useful in its ability to generate appropriate test data on an ongoing basis as function is added to the component.

2.2.2 The PIM's Profile

In defining a UML profile for PIMs, the architecture group first has to understand the particular architectural style being addressed. Then a model of this architectural style is created. Such a model (for example, [8]) might define:

- Distribution tiers
- The kinds of component in each tier and allowable interactions across and within tiers
- The kinds of classes that realize components (such as a "focus" class—a UML-defined stereotype)
- Patterns for such things as component granularity, scalability, and dependency management
- The kinds of binding (tight, loose, etc.) used for different model elements
- The required structure of the PIM itself, including namespaces (for example, a component could be required to be a separate namespace)

Second, and always assuming that the business developers are using a modeling tool that can make use of UML profiles to guide developers when they build their PIMs, the architects create a UML profile from the architectural style model. (Actually, with the better profiling tools, building the model also builds the profile). They then test the profile, before shipping it to what might be called the "development-time platform". In this way, the developers have architecture delivered to them through their everyday tools (just as detailed procedures defined by a development process might be similarly delivered).

A sample fragment of a profile for distributed enterprise component-based systems is shown in Figure 2, where relationships in red would be specified in OCL and are shown here for convenience only. The profile makes use of the new Component concept in UML2, which obviates the need for the more complex profiles that were required to model enterprise components with UML 1. The figure is addressing component granularity, and defines four levels, each characterized by a different kind of component. An Application Component is an application delivered as a component (although seldom as a single artifact). It is realized by a collaboration of "Business Components", which is the core concept presented by Herzum [2]. A Business component encapsulates a business concept such as "Customer" or "Order Manager" across the distribution tiers, and is realized by a collaboration of Distributed Components. A Distributed Component is an abstraction of the kind of component provided by EJB, COM+, and CCM. Two subtypes of Distributed Component (not shown in the figure) allow for implementation using either a programming language (e.g. EJB, COM+)—the Algorithmic DC, or using only a declarative script of some sort (e.g. BPM definitions)—"the Declarative DC". Finally, a Service Component is a collaboration of distributed components that realizes a closely-related set of services provided by a given distribution domain, for example, the "logical server" domain, or the "user interaction" domain. A full discussion of discussion of distribution domains and tiers is beyond the scope of this paper, but can be found in Sims [8].

Figure 3 shows a fragment of a model using this profile. The figure shows a Service Component in the “logical server” domain. The model uses standard UML stereotypes (focus and auxiliary) for the classes that constitute the realization of the Order component. The implementation of the profile enforces scalability through such things as restricting an ACID transaction to occur within a single invocation of the logical server domain from other domains such as user interaction or business process management (BPM) domains.

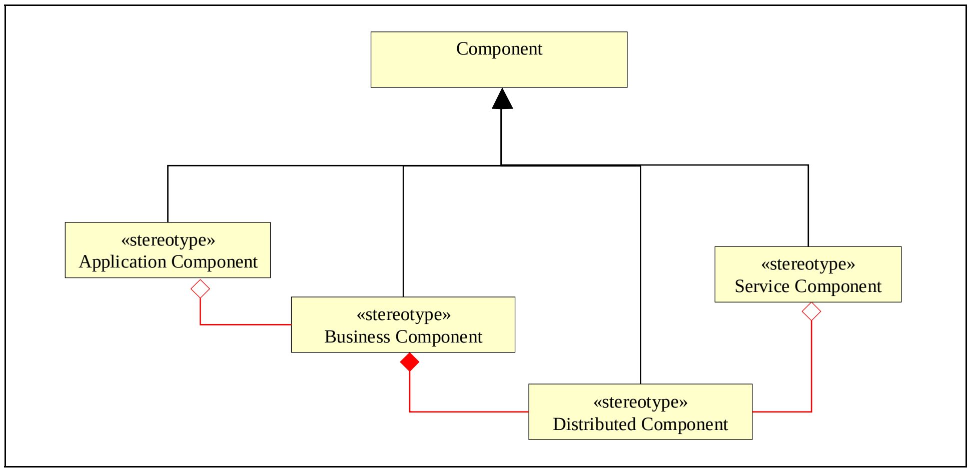


Figure 2

A modeling tool that enforces a profile can ensure that business logic developers conform to the defined architectural style. The “ilities” (scalability, maintainability, re-usability, accessibility as services, flexibility, configurability, manageability. etc.) can be, to a large extent, enforced through constraints specified within the profile, by the virtual platform, and by code generated from models that are created via the profile.

A profile of this kind can also provide for “wrapper” components that provide a service through their interfaces, and internally access legacy applications. Where there is an enterprise interoperability bus that defines a specific “real time” (as opposed to batch) interface technology and design (such as a particular usage of WSDL), the profile can be used to define wrappers from which the appropriate WSDL interfaces can be generated, perhaps wrapping some EAI adapters or BPM scripts.

2.2.3 Behavior – the Action Language

To be computationally complete, a PIM must include algorithmic behavior. This can be achieved through use of the UML Action Semantics (see [9]). Using an existing 3GL would require that the language be subsetted. In practice, this means that the subset must be documented, taught, and maintained—probably a much larger job than using an existing action language. Having said that, it must be pointed out that today, although profile building and action languages are available on the market in modeling tools, I am not aware of any tool that combines both.

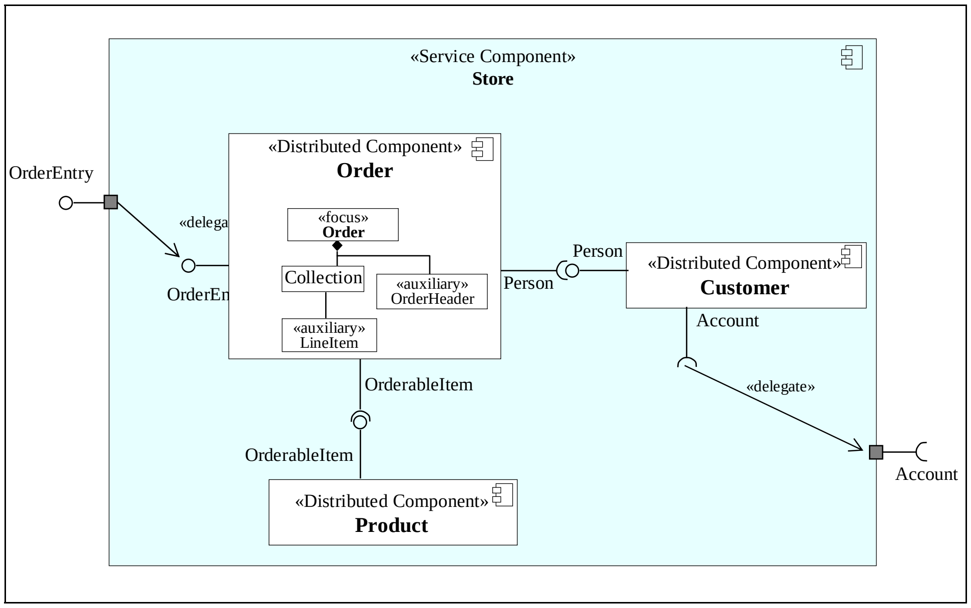
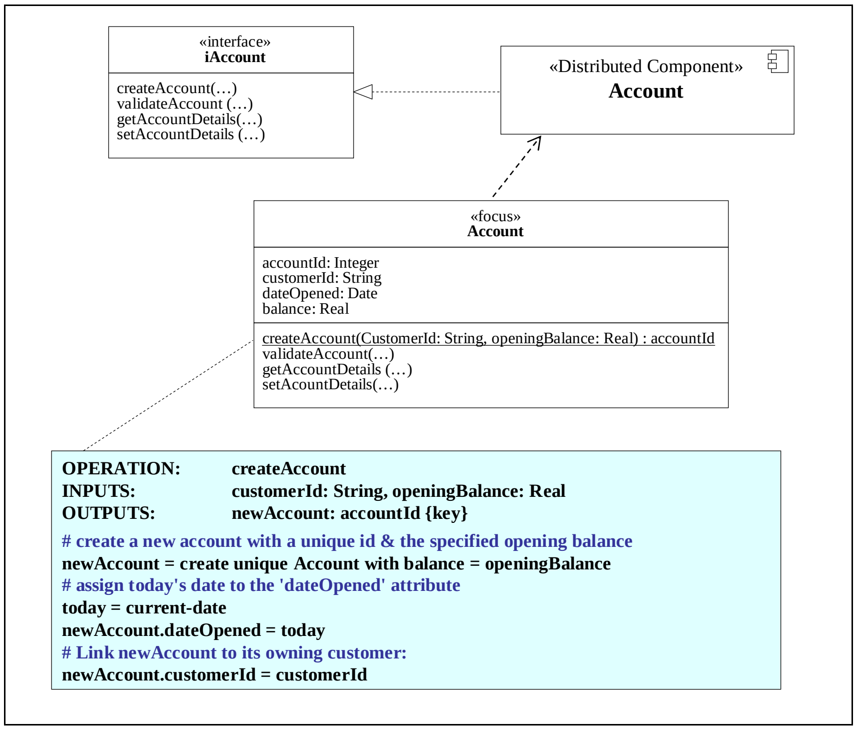


Figure 3

Figure 4 illustrates use of the action language.[7] The figure shows an Account component that provides an interface, and that has a realization—the focus class “Account”. Some design detail that would be present in a real PIM has been omitted or compressed, so please do not take this as a fragment of a real working PIM. The behavior of one operation—createAccount()—is shown, although a tool would not normally present the action language as a UML comment as the figure suggests.[8]

1.1

[7] UML defines the abstract syntax for Action Semantics, but does not define a specific notation (that is, a concrete syntax). The concrete syntax shown in the figure is Kennedy Carter’s implementation of Action Semantics. The content is a modification of a sample taken, with their kind permission, from Kennedy Carter’s tutorial on Executable UML (xUML—see http://www.kc.com/MDA/xuml.html). However, any errors or omissions in the figure I claim as my own.
[8] See [9] for an example of action language used to fully define a state machine.



```
OPERATION:     createAccount
INPUTS:        customerId: String, openingBalance: Real
OUTPUTS:       newAccount: accountId {key}
# create a new account with a unique id & the specified opening balance
newAccount = create unique Account with balance = openingBalance
# assign today's date to the 'dateOpened' attribute
today = current-date
newAccount.dateOpened = today
# Link newAccount to its owning customer:
newAccount.customerId = customerId
```

Figure 4

2.3 Business/IT Bridge

“Perhaps the greatest difficulty associated with software development is the enormous *semantic gap* that exists between domain-specific concepts … and standard programming technologies used to implement them.” [10] This has often been termed the “Business/IT Divide”, and has often seemed particularly intractable.

However, the CIM-PIM-PSM trichotomy strongly suggests that MDA provides for CIM-to-PIM generation as well as PIM-to-PSM-to-code. This means removing the business/IT divide.

A CIM is often known as a business model, or as a requirements model. There are many ways of interpreting what such a model is. To some, it is a model of the enterprise. To others, it is an idealized model of an application. Since the context for MDA is that of IT systems, I interpret a CIM to be a computation-independent model of that part of the business that is to be addressed by an IT system. While parts of a CIM might be simulated (using, for example, the “naked objects” [17] approach), it cannot, even in principle, be automatically transformed or interpreted such that it can be directly deployed into an operational system.

I also believe that the work of building a CIM stops when there are no more questions to be asked about the business in order to build the IT system. This includes low-level business processes (procedures or algorithms) about, for example, exactly how a price is calculated, or how stock is to be allocated against a sales order across perhaps several warehouses with different delivery schedules and shipping routes.

Of course, this does not imply that a CIM must be complete before work on the PIM starts: they can be gainfully and quite happily be overlapped.

So the problem is this: how can we develop a valid CIM that is also structured such that it can be straightforwardly transformed into a skeleton PIM, where the skeleton provides the structure for, and some of the content of, a valid IT system?

An answer is provided by the recent EU Combine project [13], which suggested that a business model can be seen in terms of four categories. The first two are:

- Processes that require human intervention (these are candidates for implementation as workflow using declarative distributed components that are typically implemented using a COTS workflow product)
- Processes that do not require human intervention, but where the business requires that a record be made, for future consultation, of intermediate states. (these are candidates for BPM approaches using declarative distributed components that are typically implemented using a BPM or EAI COTS product)

But what about the core business systems that BPM, EAI, and Workflow processes call upon? This is where the third and fourth categories come into play. The Combine project developed an approach, based mainly on Taylor's concept of business engineering [11] and also on my own early experience with the GUI end of CBSE [12], called "business element analysis".[9] Full exposition of business element analysis is beyond the scope of this paper; however, the basic idea is to produce process and information models as usual, then, using a set of defined heuristics, separate the model into two categories of "business element" as follows:[10]

- Processes that do not require human intervention and where the business is not interested in keeping (for future consultation) a record of any intermediate states. These processes—which can often go down to the procedure level—can often be grouped by the resource they primarily operate upon—for example, create order, amend order, delete order, and query order(s). Each such group is a "process business element",[11] and is often the responsibility of a single organizational unit. (A process business element a candidate for implementation as a process business component.)
- The "important" entity resources[12] (for example, Sales Order, Customer, Addresses) that the business needs to record for use by processes. Each such important resource is typically a group of the resources in an information model (for example, Customer has several kinds of address, various codings such as "major customer", customer number, and so forth.) Each

1.1

[9] A paper on the Combine approach to business modeling, which included a section on business elements, was presented at the EDOC 2003 conference. [14]
[10] For those familiar with the approach, it is often useful to start with the obvious business elements and derive process and information models, and further business element models, as you go.)
[11] I appreciate the term "element" being used for a group of things is oxymoronic; however, that's what the term is at present anyway.
[12] The word "important" is being used in a special sense here. A description of the identification process for "important" resources is presented in [14] and is beyond the scope of this paper. Briefly, however, in an ERP system (for example) Customer and Order are "important" whereas "Address Line" or "Quantity Ordered" are not. Suffice to say that business people have no problem with the concept. They will say for example, "Our business deals with customers, suppliers, orders, pricing engines, etc." They realize of course that such resources have attributes such as address line or actually composed people, But they do not say: "Our business handles address lines, quantities ordered, etc." A more formal version of this reality is outlined in [14]. Finally, it should be mentioned that Combine defined two quite different kinds of resource: "artifact" (information or entity) resources, and "actor" resources. For the purposes of this paper I have ignored the latter.

such group is an entity business element (and is a candidate for implementation as an entity business component).

Business element analysis provides a view of the business that is much less cluttered than many others, since low-level but essential business detail is hidden within each business element. But it does something more important from the MDA point of view. Given a component-oriented architectural style of the kind mentioned previously, it becomes clear that business elements can be nicely—and automatically—mapped into business components, where the type of the business element becomes the type of the UML2 component. Indeed, if not outlawed as just too heretical, the same model element can flow from CIM right through to code! Now that's traceability!

Figure 5 illustrates part of a CIM, with business elements represented by stereotypes of the UML2 component (the "BE" in the stereotypes signifies "business element"). Figure 6 shows a PIM that could have been generated from the CIM fragment in Figure 5. The "BC" in the stereotypes refers to the business component concept mentioned previously.

Of course, the PIM is initially very skeletal, and must be greatly refined to approach the kind of fully-specified PIM discussed previously. For example, each business component is refined into however many of the architecturally-defined distribution tiers are necessary to properly express the business concept in the system, each tier being realized by one or a few distributed components. Service components are also defined for each distribution domain.

The key point is, however, that it is quite possible to generate a PIM from a CIM such that there is isomorphic traceability. Thus there is a straight-line process from CIM through to code, which Hubert [1] calls "component metamorphosis." Note, for example, the similarity between Figure 5 and Figure 3!

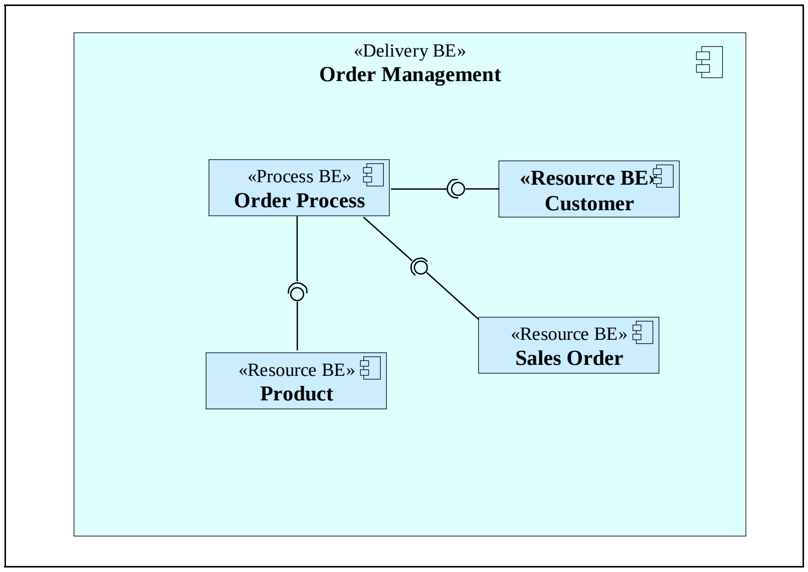


Figure 5

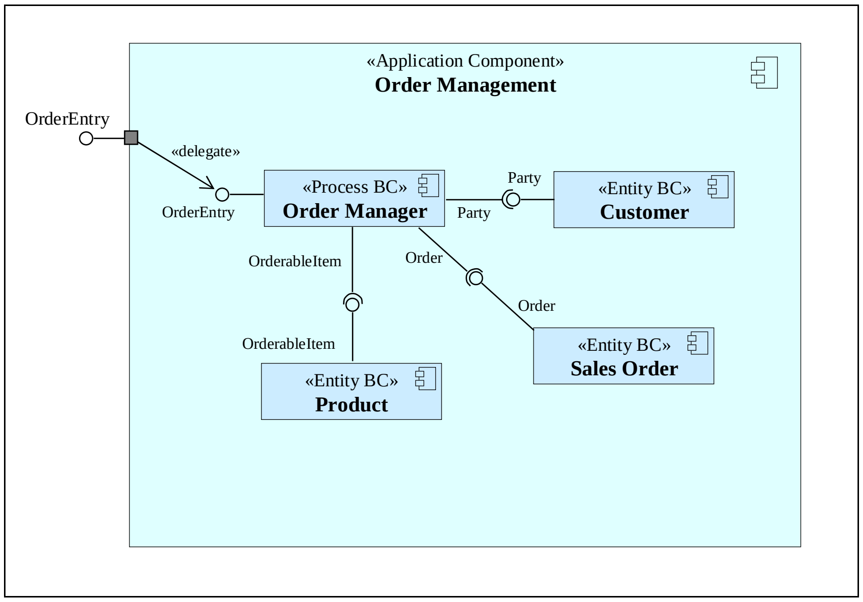


Figure 6

2.4 Domain Specific Languages (DSLs)

DSLs are an old idea currently being given a whole new set of rather attractive clothes. Described in previous MDA Journals [10] [15], they are currently being pursued by both IBM and Microsoft. There is a real sense in which a UML profile is a DSL. However, the kind of DSLs being foreseen will have a look and feel quite different than today's typical UML tools, even when profiles are applied. This is partly because an enterprise system needs other design tools than UML alone, and partly because a DSL is likely to be presented to the user as one of a family of tools, all hosted in a more general tool such as Eclipse.

I have recently been working with a client on DSLs for a particular domain within the finance industry. The domain is a small part of an enterprise system. Very surprisingly, it seems that it may be possible to start work now on a product that, in two year's time, may provide for end users to define their own applications with only a little help from their IT people. This is an extremely attractive proposition. It is made (we think) possible first through the constrained nature of the domain, second by the ideas presented in this paper, and third by the imminent emergence of mainstream DSL tools.

MDA and MDA-like approaches + Product Line + Architecture + DSLs → The Future!

3 Getting there – a roadmap

In this paper, I have tried to show that the MDA vision, in enterprise systems, can only be fully achieved through a combination of product line, architectural style thinking, and solving the business/IT divide. This doesn't mean that things like process engineering, software engineering, testing, deployment, project management, and so on are not also affected. However, the driving forces are the organizational and technical directions of the product line approach, and the structural and simplifying directions of applied architectural styles and CIM-PIM linkage, with the MDA approach tying them together synergistically.

Although Microsoft is progressing along a slightly technical different road than MDA, its essential goals for enterprise systems seem to be similar.

So how do we get there? In effect, we're looking at a transition from where we are now to a much-improved development environment. And although all the capabilities needed are not yet integrated into commercially available tools, there is certainly sufficient support available now for early adopters to start the journey. In particular, we can construct UML profiles for enterprise systems today, and generate at least skeleton code.

The question is, how does an IT organization make the transition to MDA?

Luckily, transition processes are not new. Guttman and Matthews [16] describe a particularly good one. The key is non-intrusion into current and planned development projects, and certainly to avoid big bang approaches. Thus the idea is to start small, developing initial capability in the context of a few (one to three) real projects. These projects might be termed "pilot projects" because, although real development projects, they are the vehicle to pilot the transition.

But who does the extra work (for extra work there will inevitably be)?

Another key part of the transition is to fund a group that has, or is in the process of gaining, technical knowledge of the MDA approach, and who share the MDA vision. People in this group, skilled architects, designers, modelers, and software technology engineers, will devote somewhat more than 50% of their time to working in the pilot projects as project members, helping to produce project deliverables. The other part of their time is spent growing the virtual platform, based on their project experience and on project priorities. This will include capturing the architectural style in a UML profile, applying that profile through tools, and so forth.

In effect, this group is the genesis of a separate Infrastructure/Architecture/Process unit (what we called the "platform group" previously) within the IT organization. It is funded to provide and evolve a high-productivity environment for business application developers. Working within projects, and developing re-usable "glue" based on project priorities, should prevent this group becoming an ivory tower; especially when the results of their efforts are deemed null-and-void unless they are delivered through tools, and unless business application developers—their customers—like the results. In other words, the IT organization must evolve to one where the platform group provides high-quality and immediately useful services and "products" to their customers, who are the business application developers and their managers.

The transition process moves forward from the initial pilot projects through several defined stages (with go/no-go points built in), ending with phased roll-out of MDA capability to the whole of the IT development organization. On the way, the platform group will probably evolve into two groups: an "infrastructure" group responsible for provisioning and maintaining the virtual platform, and an "architecture" group responsible for designing the virtual platform. Process may be handled by the architecture group or by a separate group within the platform area.

Experience suggests that the major impediments to success in such a transition are funding problems and difficulties in making the required organizational changes. Creating a product line environment that majors on the MDA vision requires a fixed focus on making life easier for the business application developer. After all, dealing with the awesome intricacies of the reality of business is challenging enough. Dealing at the same time with forty-eleven services thoughtfully provided by middleware vendors is a truly Herculean task. We must change to an environment that does not require each business application developer to be a Hercules.

4 Summary

We asked how the claims made for MDA can be substantiated in the context of enterprise distributed systems. One answer lies in the synergistic combination of three major approaches: product line, component-based enterprise architecture, and the business element approach to resolving the business/IT divide. Product line organization separates technology logic from business logic; clean business logic is structured, and key 'ilities provided for, by a component-based enterprise architecture; the enterprise architecture defines a structure into which the CIM business elements can be isomorphically mapped. MDA provides the catalyst. The result is the prospect of fully specified enterprise PIMs.

Figure 7 illustrates this vision. Within the platform group, the infrastructure group delivers the development and run-time environments, complete with generators and transformation tools, as well as the various COTS products. In the figure, the major tools used by business application developers are represented by the "Modeling and Development Tools" box. Profiles that the architecture group provides are used to define PIMs and to configure the various tools. Solid arrows show the main data flow through the development lifecycle where the "data" consists of artifacts such as the CIM, the PIM, and so on. Dotted arrows show dependencies; for example the modeling and development tools depend on the enterprise SOA profile.

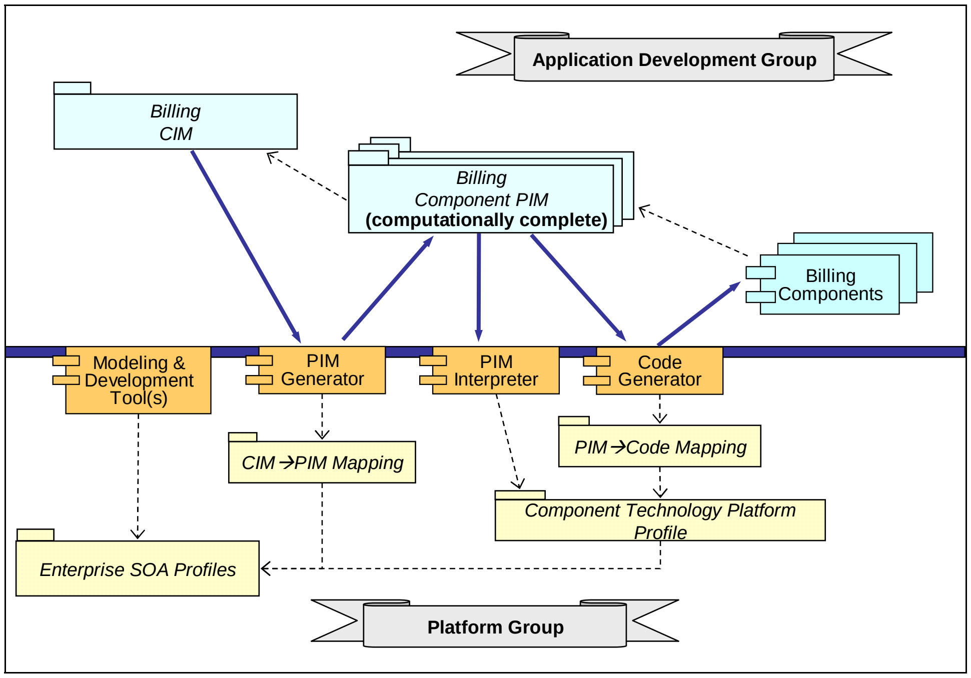

Application Development Group
Billing CIM
Billing Component PIM (computationally complete)
Billing Components
Modeling & Development Tool(s)
PIM Generator
PIM Interpreter
Code Generator
CIM→PIM Mapping
PIM→Code Mapping
Component Technology Platform Profile
Enterprise SOA Profiles
Platform Group

Figure 7

Achieving such synergy is not easy. However, a viable way ahead is available for those who choose to take it. Already today there are many organizations taking the first steps in MDA, and there are many vendors who provide various aspects of the MDA vision. Indeed, everything needed to achieve the vision is available today—just not in the same place! This isn't about being an early adopter of MDA—it's too late for that. It's about being an early adopter of the longer-term aims of MDA—with which we started this paper.

The end point of this vision is that, perhaps late in this decade, there will be a generation of application developers, whether in-house or outsourced, who never write programs in today's languages. Instead, they will design fully specified PIMs of their business logic and structure, PIMs that can either be directly executed by an architecture-aware UML virtual machine, or used as the basis for automatically generated code.

MDA has put a stake in the ground. The stake is a signpost to a most desirable future. Both IBM and Microsoft have announced their intention of getting there, albeit by different technical routes. Their chosen tools centre on IBM's Eclipse with EMF (Eclipse Modeling Framework), and

Microsoft's Visual Studio with the extensions mentioned by Steve Cook [15]. Steve says that Microsoft will not be using precisely the same MDA technologies (MOF, UML, etc), but will use variations of them. Both include the concept of DSLs, whether UML-based or not.

However the future unrolls, it is the MDA vision, as opposed to its current technologies, that points a way out of the current morass of cottage industry approaches to IT, to the broad sunlit uplands of truly effective, productive, agile, and flexible system development. Today code is king. Tomorrow design will be king. And the process of bringing innovative solutions to business challenges will become progressively simpler as more and more of the underlying software technology that today's business developers battle with becomes increasingly buried in the platform—which is the true domain of software technology experts.

5 References

[1] R. Hubert, *Convergent Architecture*, Wiley 2002.

[2] P. Herzum & O. Sims, *Business Component Factory*, Wiley 2000.

[3] P. Clements & L. Northrop, *Software Product Lines*, Addison-Wesley 2002.

[4] D. Frankel, *The MDA Marketing Message and the MDA Reality*, MDA Journal, March 2004.

[5] J. Bettin, *Model-Driven Software Development*, MDA Journal April 2004.

[6] OMG Document ptc/04-05-02 *UML 2-0 Superstructure Specification* (www.omg.org).

[7] H. Kilov, *Business Specifications*, Prentice Hall 1999.

[8] O. Sims, *A Component Model*, Cutter Executive Report Vol 5 No. 5, May 2002.

[9] S. Mellor, *Agile MDA*, MDA Journal June 2004'

[10] G. Booch et al., *An MDA Manifesto*, MDA Journal May 2004.

[11] D. Taylor, *Business Engineering with Object Technology*, Wiley 1995.

[12] O. Sims, *Business Objects*, Wiley 1994 (now out of print but available in pdf form at http://www.simsassociates.co.uk/books.htm).

[13] The COMBINE Project – See http://www.opengroup.org/combine/overview.htm. Details of the project results are not yet publicly available.

[14] S. Tyndale-Biscoe, et al., *Business Modelling for Component Systems with UML*, paper presented at the EDOC 2002 Conference, Lausanne.

[15] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal, January 2004.

[16] M. Guttman & J. Matthews, *Migrating to Enterprise Component Computing*, Cutter Executive Reports, 1998/9.

[17] R. Pawson & R. Matthews, *Naked Objects*, Wiley 2002.

End of Paper