MDA Journal

December 2005

David S. Frankel
Lead Standards Architect - Model Driven Systems
SAP Labs

David.Frankel@SAP.com

MDA, UML, XMI, and CORBA are Registered Trademarks of the Object Management Group. The logo at the top of the second page is a Trademark of the OMG.

www.bptrends.com

Contents

Introduction 1
Business Process Platforms: A Quick Recap 1
Some of the Principal Challenges 2
  Finding the Right Components 2
  Building the Right Components 3
  Configuration Management 3
    Design-TIme and Deployment-TIme Configuration 3
    Versioning 3
Addressing the Challenges 3
  A Metadata-Rich Environment 4
    Semantically Rich Service Contracts 4
    The Benefits of Service Contracts 6
    Quality-of-Service Constraints 7
    Configuration Invariants 7
    Version Metadata 7
    Metadata Management 8
  Building the Right Components via Product Line Practices 8
Conclusion 10
References 10

Introduction

In July's MDA Journal, I described the rise of business process platforms and the opportunities that they portend. I also mentioned that the transition to business process platforms presents some challenges, and thus will be gradual. In this edition of MDA Journal, I examine the principal difficulties, and strategies for overcoming them.

Business Process Platforms: Quick Recap

A business process platform is a new type of platform that has two basic parts (Figure 1):

1. *Technical Software Platform*: A technical software platform is a set of technical capabilities that are packaged so that application developers can use them to build applications more easily than would otherwise be possible. It includes database and network management systems. It also includes middleware that manages transactions, componentization, security, persistence, data transformation, web services, and so on. The steady rise in the abstraction level of technical software platforms has enabled a similar rise in the abstraction level of programming languages and model-driven development tools.
2. *Application Platform:* An application platform consists of frameworks of reusable, executable business-oriented services – such as a post-to-ledger service – and executable business processes composed of multiple services, such as an order entry process. Application platforms

[1] http://www.openapplications.org/wg/PaymentHarmonization/200311107-Gartner/Background.htm

Copyright © 2005 David Frankel

MDA Journal

December 2005

enable a further rise in the abstraction level of model-driven tools that compose services into more complex services, processes, and applications. An application platform sits on top of a technical software platform.

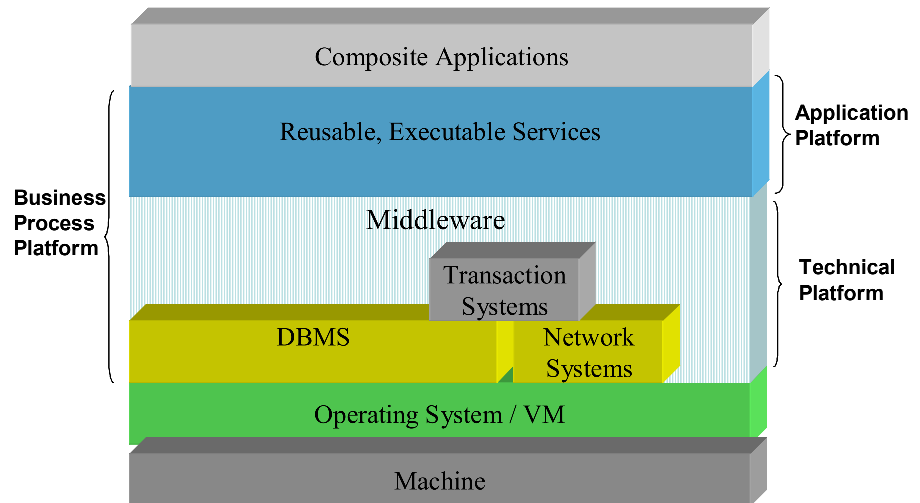

Figure 1. A Business Process Platform

Some of the Principal Challenges

As we scale business process platforms (BPPs) up, a number of issues arise. We focus on three specific challenges having to do with the application platform, the most novel part of the BPP. The challenges are these: Finding the right components, building the right components, and configuration management.

Finding the Right Components

Over time, business process platforms will offer increasingly large portfolios of reusable services, as well as larger numbers of processes that are composed from those services. With a vast array of reusable components from which to choose, a basic problem confronts those who build new applications or business processes from the components: *How do you find suitable services and processes to reuse?* That is, how do you identify components that may be suitable for building your new service, process, or application? And, once you have identified candidate components, how do you vet the list of candidates? How do you determine whether the multiple components that you wish to reuse to build your new offering are compatible with each other?

Copyright © 2005 David Frankel

MDA Journal

December 2005

Building the Right Components

An application platform is the latest evolution of service-oriented architecture, and service-oriented architecture is the latest incarnation of component-based development. The idea of building software from reusable components has often not worked in practice as well as hoped. It is not easy to anticipate the needs of those who want to assemble things from components. It is particularly difficult in large platforms intended to support a multitude of business needs including enterprise resource planning, customer relationship management, supplier relationship management, and so on.

Configuration Management

Just because you can compose applications from reusable components relatively quickly does not mean that all traditional challenges across the software lifecycle go away. In fact, some of them can get worse. Configuration management is one area that can be particularly complicated.

Design-Time and Deployment-Time Configuration

Each component may have a set of properties that the application modeler or developer sets at application design time. For example, a design-time configuration property of an accounts payable component might indicate whether to use cash or accrual accounting; the application developer sets the property's value when reusing the component to build an application.

Furthermore, the component may have a set of properties to be set at deployment time – that is, at the time when the application that contains the component is deployed. An example of a deployment-time configuration property for our accounts payable service is a reference to a relational database source that the component uses for persistence. There may also be technically oriented deployment-time properties, such as one that determines whether the service communicates via JAVA/RMI/IIOP, WSDL/SOAP, and so forth.

The more that we fulfill the dream of assembling applications from components, the more complex managing these configuration properties becomes, because each component is likely to have its own set of configuration parameters. It can be daunting to debug errors that arise from subtle incompatibilities among configuration property settings for multiple components.

Versioning

The fact that each component that makes up an application has its own lifecycle and version history complicates matters further. You not only have to worry about whether you've deployed the right version of an application at a particular site. You also have to worry about whether you've deployed the right versions of all of the components upon which the application depends.

Addressing the Challenges

There are several things that we can do to address these challenges to scaling business process platforms.

Copyright © 2005 David Frankel

MDA Journal

December 2005

A Metadata-Rich Environment

In order to make finding and managing large numbers of components tractable, a business process platform has to be a metadata-rich environment. Metadata provides machine-readable information about the components of the business process platform.

This section outlines the most crucial, foundational kinds of metadata that business process platforms need.

Semantically Rich Service Contracts

The contract of a service is composed of four fundamental elements:

- *Signature*: The signature of a service defines the types of information that the service requires as input and the types that it produces as output.
- *Preconditions*: A precondition is a constraint that must be satisfied in order for the service to execute in a well-defined manner. An example of a precondition for a service that transfers money from one bank account to another might be that the source and destination accounts have to be owned by the same customer. Some services might not have this restriction, but, for those that do, this precondition is an important part of the service's contract.
- *Postconditions (Effects)*: A postcondition is a constraint that must be satisfied when the service has completed execution. Postconditions describe the effects that result from executing the service. For example, a postcondition of our example money transfer service is that the balance of the source account is decremented by the amount of the transfer, and the balance of the destination account is incremented by the same amount.
- *Information Invariants*: Information invariants are constraints that apply to the information structures that services use as their input and output parameters. For example, an ordinary checking account may have an invariant that says that the balance of the account may never dip below zero. That invariant would invalidate an attempt to transfer an amount from an account that is greater than the account's balance. However, a different kind of checking account with overdraft protection might have an invariant that states the balance may dip as low as negative one thousand dollars, but no more. Still another kind might require that the balance never go below some minimum amount.

Typically in the industry today, the only part of the service contract that is expressed in a machine-readable fashion is the signature. For example, WSDL allows you to define the inputs and outputs of services. The use of preconditions, postconditions, and information invariants to define a service's contract is the heart of an approach called *Design-by-Contract*™, which defines the semantics (the meaning) of the contract. When we express these semantics using formal constraint languages, they become machine-readable too.[1] More often than not, the semantics of a service contract are buried in code, and are not even captured informally in English or some other human language. In a metadata-rich

[1]For the technically inclined, Design by Contract requires that we express each precondition, postcondition, and invariant as a declarative, Boolean assertion.

Copyright © 2005 David Frankel

MDA Journal

December 2005

environment, the full contract in machine-readable form – including the signature and the semantics – is an integral part of the component's manifest.

Figure 2 and Figure 3 illustrate how to declare a machine-readable contract for a money transfer service using UML®'s Object Constraint Language (OCL). The constraints are in the blue boxes (shaded boxes, if viewing in black and white).[2] The UN/CEFACT Modeling Methodology (UMM) is one example of a methodology that uses UML this way.[3]

There are other avenues besides UML, including various initiatives under the banner *Semantic Web Services* aiming to use constraint languages in concert with the Web Services Description Language (WSDL) and in conjunction with the Semantic Web languages RDF and OWL. Sometimes these initiatives refer to constraint languages as *rules languages*, but the idea is the same. There are also emerging technologies for representing machine-readable constraints in a human language – such as carefully structured English – so that business analysts who are not technically oriented can read them.[4]

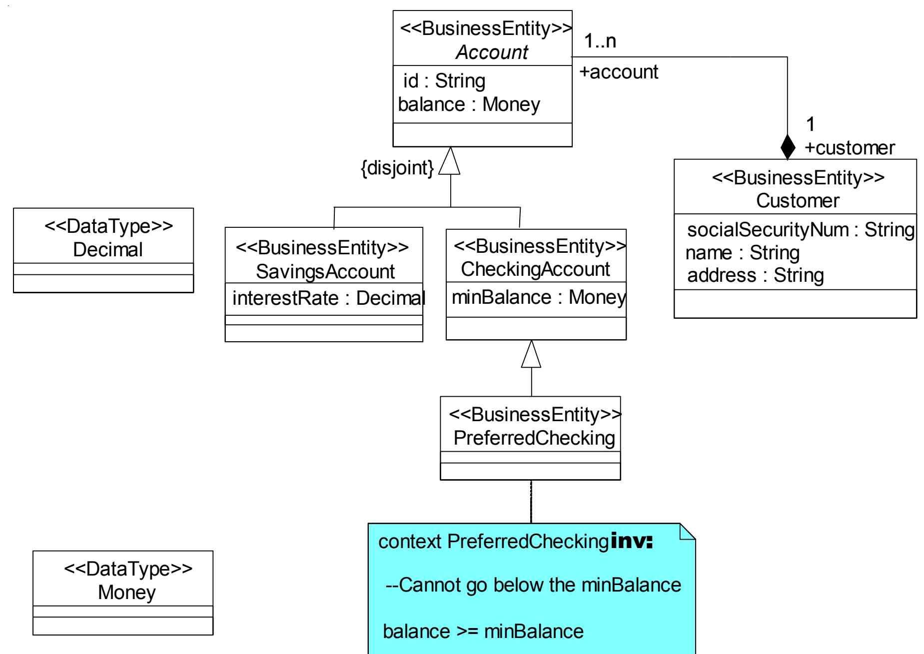

Figure 2. Business Information Model

There are also approaches to structuring business information that reduce – but do not eliminate – the need to express information invariants in general constraint languages. UN/CEFACT has an approach called *Core Components (CCTS)*[5] for organizing the definitions of *global data types*, such that the very structure of the definition of a data element provides a map of the element's semantics. CCTS leverages ISO 11179-5's approach to using semantic structure to drive the naming of data elements.[6] ISO 11179-5 is also popular in the Semantic Web community. These semantic maps are powerful metadata that enrich the semantic content of the service contract when the data elements

[2] In OCL, each invariant is labeled "inv:" and In OCL, each precondition is labeled "pre:" and each postcondition is labeled "post:". Comments are preceded by a double hyphen; I use comments to express the constraints in English.
[3] [UMM]
[4] [SBVR]
[5] [CCTS]
[6] [ISO 11179-5]

Copyright © 2005 David Frankel

MDA Journal

December 2005

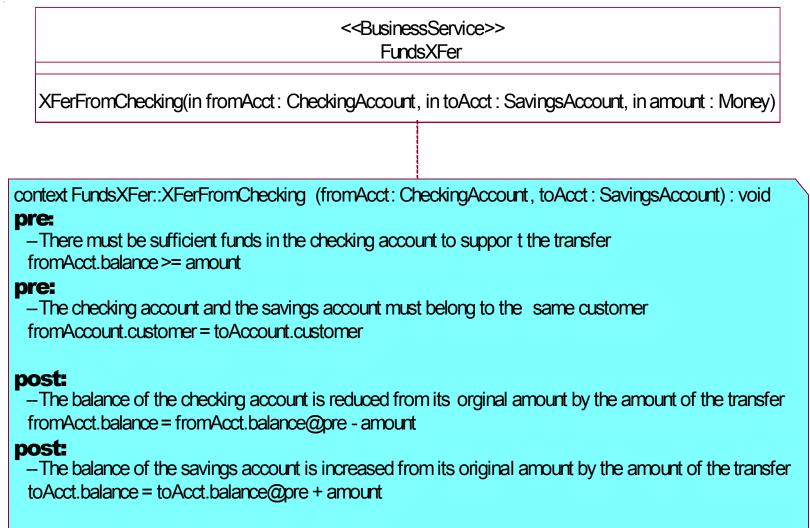

Figure 3. Business Service Model

appear as parameters to the services. We are just beginning to learn how to exploit this kind of metadata.

The Benefits of Service Contracts

Including service contracts in the metadata for the business process platform provides some basic benefits that are a foundation for addressing some of the scalability issues that I outlined earlier. Advantages include:

- *A precise, unambiguous contract*: Most basically, a precise specification of the contract makes it more likely that the person who wishes to reuse the service component will understand whether the component is likely to satisfy the requirements. Mathematically specified constraints are unambiguous. This is particularly important in an age where value-chain driven business means that components have to interact with each other across organizational boundaries and across human language barriers. Whether the service user wishes to create a more powerful service, create an executable business process that invokes the service, or build some kind of specialized application with the service, knowing the contract helps.
- *Constraint enforcement*: With some limitations, tools can enforce constraints. For example, it is possible to generate code that checks whether a precondition is satisfied before invoking a service.
- *Collision detection*: Breakthroughs in the field of knowledge representation make it possible to detect certain kinds of conflicts among contracts. Tools can prevent subtle bugs before they happen by warning a developer that the components she wishes to combine have mutually contradictory constraints. For example, a constraint on one component may require that the customer's age be 18 or less, whereas a constraint on another component may require that the customer's age be 21 or more. In order to take full advantage of these emerging tools, the languages used to define the information structures must be grounded

Copyright © 2005 David Frankel

MDA Journal

December 2005

via mathematical formalisms, as is the case with the Semantic Web's RDF and OWL. Even with such formal grounding, certain kinds of collisions cannot be detected with certainty; but in such cases the tool can raise a warning flag, allow the human to decide how to proceed, and store the human's decision. The next time a human encounters the same warning, the tool can display how other humans have decided to proceed in the same circumstance, and the tool can eventually learn from those decisions to suggest a default decision.

- *Semi-automating service composition*: Tools are emerging that partially automate the identification of candidate services that may satisfy a requirement. The tools require the searcher to express the requirement in a machine-readable fashion, and they scan available services by matching the requirements against the service contracts. For the foreseeable future, the human will have to be the ultimate arbiter of whether to use the candidate services, which is why we label this approach as semi-automated. Once again, these tools require the use of formally grounded languages.

Quality-of-Service Constraints

Note that the service contracts I've talked about so far specify the functional behavior of the service. They do not specify non-functional, quality-of-service constraints. Yet quality-of-service constraints are important metadata too. Finding a suitable service is not just about finding a service that implements the functional behavior that you need. It is also about finding a service that satisfies your quality-of-service requirements.

It is useful to separate the functional aspects of the contract from the non-functional, quality-of-service aspects. For example, standards bodies might want to codify the functional behavior contract for a common service, such as a post to ledger service, while leaving the specification of quality-of-service constraints to negotiations among the providers and clients of the service. Implementers who provide the service can advertise the quality-of-service that the implementation offers.

Configuration Invariants

A configuration invariant is a special kind of information invariant that specifies a constraint having to do with a component's design-time or deployment-time configuration parameters. The value of one configuration parameter may constrain the values of others. Tools can enforce these kinds of constraints, with some limitations.[7] They can also detect collisions among configuration constraints that would result from specific combinations of components; again, sometimes detection is certain and sometimes only suspected.

Applying these principles of Design by Contract to configuration management does not solve all problems, but can help manage them.[8]

Version Metadata

You need quite a bit of metadata to get the versioning problem under control. Simply tracking the versions of the application is not enough. You have to track

[7] For the technical reader: The limitations I've referred to several times have to do with the complexity of the constraints. It's possible for tools to detect – with certainty – collisions among the relatively simple kinds of constraints that make up description logics. However, certainty is generally unattainable when trying to detect collisions among constraints that use first order logic

[8] See [CZAR-KIM] – particularly sections 5 and 6 – for an interesting discussion of how tools can use invariants to manage configuration. The lead author is currently an academic, but spent many years in industry.

Copyright © 2005 David Frankel

MDA Journal

December 2005

the versions of all of the application's components, and you have to do that for each version of the application.

In order to be able to anticipate the potential ramifications of a change to a component, you also need a cross reference that keeps track of which applications are using which components, down to the version level. This, of course, impacts quality assurance procedures as well.

Metadata Management

Business process platforms need all sorts of metadata to reach their potential. Because of the wide variety of metadata, there is a danger that we won't be able to use it in an integrated fashion. Too often, enterprises have quite a bit of metadata, but most of it is locked up in silos because each kind of metadata is managed by different tools that use completely different metadata management mechanisms.

That's why I've been such a strong proponent of integrated metadata management technologies such as the Meta Object Facility (MOF™), one of the core MDA standards. The Eclipse Modeling Framework (EMF) is essentially a flavor of MOF that underpins Eclipse's metadata management facilities. Eclipse's tremendous traction holds out hope that the silos will start to break down. EMF is the second most popular Eclipse download after the Eclipse SDK itself.

Building the Right Components via Product Line Practices

The Carnegie Mellon Software Engineering Institute (SEI) – the same organization that gave us the Capability Maturity Model (CMM) – has defined an approach to organizing for reuse called *Software Product Lines (SPL)*. SPL addresses one of the main problems that has bedeviled component-based development – the problem of scope. Thoughtfully constraining the scope to which frameworks of components apply makes the problem of making the components truly reusable more tractable.

SEI defines a software product line (SPL) as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[9] An example of an SPL is a set of products that manages risk for portfolios of tradable financial derivatives. Another example of an SPL is a set of role-based access security products. Note that the scope of these sample product lines is fairly narrow. It is easier to assure reusability over a well-defined, restricted scope than it is to do so over an ill-defined and broad scope.

The SPL approach (see Figure 4) divides the software development process into two distinct but related processes – core asset development and product development. Core asset development produces a framework of reusable assets for the product line, and defines an architecture for the framework. Product development uses the framework to produce individual products. A production plan provides instructions on how to use the framework in accordance with the architecture in order to produce products. It is interesting to note that SPLs are a cornerstone of Microsoft's *Software Factories* approach to model-driven systems.

[9] [SPL]

Copyright © 2005 David Frankel

MDA Journal

December 2005

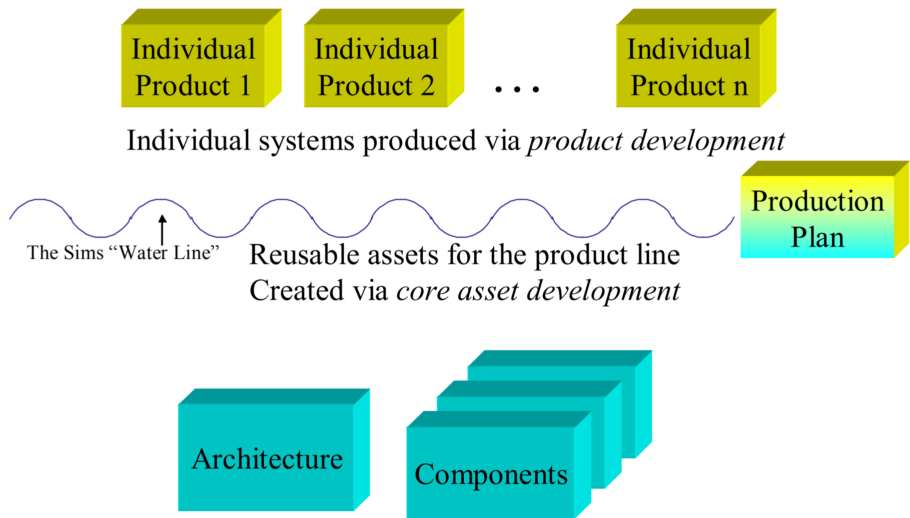

Figure 4. Software Product Lines[10]

Organizing a business process platform as a set of distinct but integrated product lines (see Figure 5), each with a well-defined scope, can help to make the task of building reusable service and process components more manageable.

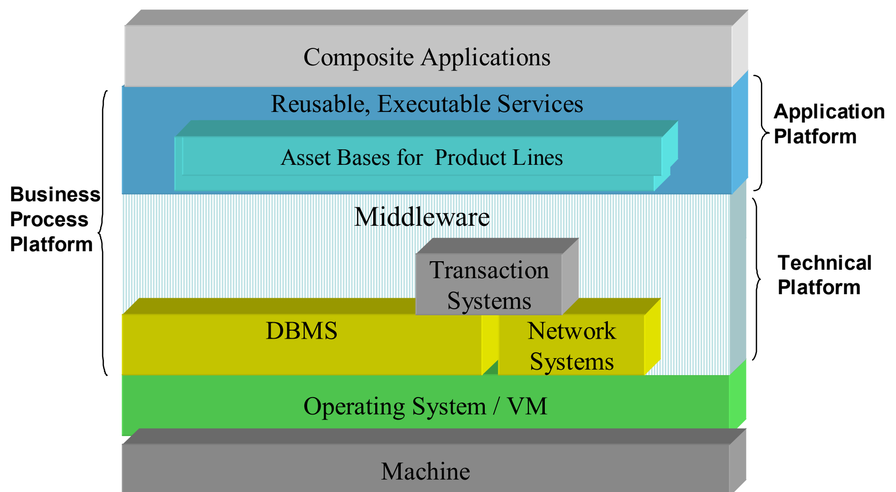

Figure 5. Applying Software Product Lines to Business Process Platforms

[10] The "Sims Water Line," depicted in Figure 4 and invented by Oliver Sims, uses the analogy of a water line to describe the separation of concerns between aspects of a system that surface to the application developer's viewpoint, as opposed to aspects that the infrastructure handles below the surface.

Copyright © 2005 David Frankel

MDA Journal

December 2005

Conclusion

Business Process Platforms will change the face of IT and business. For many reasons, the change will be gradual. Along the way we have to address some challenges that arise as we increase the scale of this new kind of platform. Now is the time to recognize these challenges and face them square on. An approach that combines this kind of pragmatism without losing sight of the goal can help ensure that business process platforms provide real business value at every stage of the evolution.

References

[CCTS] UN/CEFACT, Core Components, http://www.unece.org/cefact/ebxml/CCTS_V2-01_Final.pdf , also published as ISO 15000-5 under the auspices of ISO TC 154

[CZAR-KIM] Krzysztof Czarnecki and Chang Hwan Peter Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," Proceedings of the First International Conference on Software Factories, OOPSLA 2005.

[ISO 11179-5] ISO 11179 Metadata Registries, Part 5 Naming and Identification Principles, http://metadata-standards.org/11179/#11179-5

[SPL] Carnegie-Mellon Software Engineering Institute, *Software Product Lines*, http://www.sei.cmu.edu/productlines/index.html

[SBVR] *Semantics of Business Vocabulary and Business Rules*, OMG document bei/2005-08-01, 22 August 2005.

[UMM] UN/CEFACT Modeling Methodology, "Design by Contract" is a trademark of Interactive Software Engineering.

Copyright © 2005 David Frankel