Model-Driven Architecture and Integration

Opportunities and Challenges

Version 1.1

Desmond DSouza, Kinetium (desmond@kinetium.com)

Model-Driven Architecture (MDA) is an approach to the full lifecycle integration and interoperability of enterprise systems comprised of software, hardware, humans, and business practices. It provides a systematic framework to understand, design, operate, and evolve all aspects of such enterprise systems, using engineering methods and tools. The framework is based on modeling different aspects and levels of abstractions of such systems, and exploiting interrelationships between these models.

In this paper we drill down into the MDA vision, define it's broader objectives from a modeling and integration perspective, identify key opportunities and challenges, and show how the ***"M****odel"* *and* ***"A****rchitecture"* *in "MDA" could be used to effectively enable large-scale model-driven integration. We also highlight how the MDA will impose significant requirements on both the UML and MOF.*

For a current version of this paper, more information about solutions for model-driven architecture and integration, or to discuss how these issues might be addressed in UML 2.0, visit www.kinetium.com or email info@kinetium.com

About Kinetium

Kinetium is a start-up focused on leveraging shareable architectures for model-driven development and integration of systems, based on precise architectural styles and inter-model relationships across models of domains, subject areas, levels or abstraction, and technology platforms. Stripped off its buzzwords, that simply means that we offer advanced solutions to those who are ready to exploit a model-driven approach to architecture, integration, and interoperability, through services, methods, and products. For more information visit www.kinetium.com or email to info@kinetium.com

1 Introduction

Model-Driven Architecture (MDA) is an approach to the full lifecycle integration of enterprise systems comprised of software, hardware, humans, and business practices. It provides a systematic framework to understand, design, operate, and evolve all aspects of such enterprise systems, using engineering methods and tools. MDA is based on modeling different aspects and levels of abstraction of a system, and exploiting interrelationships between these models.

MDA is motivated by integration and interoperability at the enterprise scale. It utilizes models and a generalized idea of architecture standards to address integration of enterprise systems in the face of heterogeneous and evolving technology and business domains. From [Soley00]:

> *"It's about integration. It's about interoperability….The need for integration remains strong – we need to build on the success of UML and CORBA … to provide the model-based standards that are necessary to extend integration beyond the middleware approach."*

In this paper we drill down into the MDA vision, identify key opportunities and modeling challenges, and show how the **"M**odel**"** and **"A**rchitecture**"** in "MDA" could be used to effectively enable large-scale model-driven integration. This paper make several telling points in a fairly dense form, and some may be lost in a casual reading. The main points are listed here:

1. Software systems exist to support a business. Hence modeling of software systems is strongly linked to modeling the environment of those systems, their business processes and domains.

2. A system can be described from different viewpoints, each focused on particular concerns. A viewpoint, applied to a system, gives a view of that systems. That view, expressed in one of the languages suited to that viewpoint, is a model of that system from that viewpoint, and in that language. The same viewpoint can be used on different systems. This is true from the level of business strategy and policy to engineering and technology details.

3. A system can be modeled at different levels of abstraction, even within certain viewpoints: the outside (the environment into which the system will be deployed to play its part in some larger goal), the boundary (the interfaces of the system), and the inside (its internal design). Further, any of these descriptions can be at varying granularities of objects and actions. This is true from the level of the enterprise to engineering and technology components.

4. Separated viewpoints are interrelated; integrating them requires that relationships, including correspondences, between the models be clearly defined, separately from models of each viewpoint. Similarly, each interface of a component is modeled separately; when the entire component is specified, these models have to be integrated.

5. Separated levels of abstraction are related in non-trivial ways so the realization faithfully meets the guarantees made by the abstraction. This relationship, called refinement, must be explicitly modeled, separately from both abstraction and realization, and regardless of the order in which abstraction and realization were created or the degree of automatic generation or transformation involved. Refinement enables a fractal zoom-in/zoom-out approach, and lets us treat a multi-tiered business component as either a single object or as a composition.

6. The modeling core must be very small and precise e.g. object and action applies uniformly at

all levels (Section 6.3). Once we allow refinement of objects (a single abstract object is refined to some community of finer-grained objects) and of actions (a single n-way abstract action is refined to some protocol of finer-grained actions) we have a core that works across levels, and can build appropriate higher-level constructs and notations on this base. Note that every needless distinction at the core level causes multiplicative overhead as we layer on that core.

7. Architecture guides design. For any abstraction there are many possible realizations, and the architecture style you use admits some of those realizations but not others. A good architecture provide shared ways of devising suitable realizations for different abstractions.

8. The approach of connecting building-blocks into larger assemblies applies at all levels of development, including code, interface specs, domain models, and architectures. To compose Java Beans, you connect event, property and method ports to create larger assemblies; to composing Corba components, you connect facets and receptacles ports; to compose models of design patterns, you connect parts by substituting your own model elements for places in the design pattern; to connect processes you identify outputs with inputs and connect using flows.

9. Interoperability is about ensuring that parts which embody different realization decisions can still be integrated so they work together as expected at a more abstract level. *"… as expected at a more abstract level"* implies that we have to be quite careful about how we interpret the models[1] at both levels, and about precisely what is normative and guaranteed vs. what is auxiliary and the relation between these two.

10. Interoperability specifications need conformance tests. However, what happens to those tests if the specs deliberately permit refinement of observable behavior at the concrete level? For example, a platform-independent specification will translate into different observable platform-specific interfaces. Platform-independent tests require a precise definition of the architectural styles that transform platform-independent to platform-specific observable behaviors.

11. To define component interoperability, we must distinguish specification of a replaceable component (a 'closed' black-box spec including all it guarantees to, and requires from, its environment, via all interfaces), from specification of a single interface (an 'open' black-box spec where all other interfaces are not described) e.g. consider a legacy wrapper component.

12. Hence a common core of concepts – abstraction, realization, refinement relation; viewpoint, view, integration of views, interfaces; component, port, connector, assembly; and architectural style – exists at all enterprise levels; and objects and actions work at all levels.

13. Certain categories of systems – such as those referred to as "product-lines" – are highly configurable to fit into different environments and processes. These bring additional challenges of characterizing domain variability, choosing a flexible set of components and composition machinery, specifying requirements for a given configuration, and mapping to a particular configuration of components that will realize that requirement.

14. Models and systems at this scale will always evolve. The architecture for the models should support transparent re-factoring of some parts without needlessly impacting others.

15. Models will be structured in packages, but well-written concise prose documents are a

[1] This general risk of over-specification is particularly true of non-declarative models.

necessary wrapper around the models, adding rationale, examples, tables, analogies, metaphors, and the like. So packages include documentation. Beyond documents and traditional models, packages contain source code, compiled code, patches, requirements (including change requests and bugs reports), tests, test results, and structured documents. Packages even define inference rules – what you can infer from any model.

16. Models of software components provide important run-time descriptions. If a Interface Repository linked Corba or .NET registry information to more abstract models, it could offer run-time access to models of the platform-independent behavior of interfaces (their guarantees and assumptions, beyond just interface signatures), connected to models of their domains. This enables services such as traders offering intelligent matching services, run-time negotiation and creation of cross-platform adaptors across different data and interaction protocols, and smart agents that discover and reason about each others world views.

17. Models should include, or at least be integrated with, distribution and deployment information. A facility which linked models to directory services could offer comprehensive descriptions about all entities listed in those directories, and vice-versa. This could form the basis of keeping models and the actual entities being modeled in sync as needed, through both model deployment and model discovery, and for intelligent run-time services that understood some parts of the specifications and mappings in the models.

18. The MDA initiative, if successful, will define a large space populated by a huge number of models. There has to be some coherent structure on this space of models, providing a rational and consistent way to locate, partition, and relate these models. This structure is the essence of the "**A**" in "MDA".

2 The Integrated Enterprise

Since MDA is about integration, here is our vision of the integrated enterprise.

> An integrated enterprise is one whose multiple business areas, strategies, architectures, organizational units, people, processes, automated systems, technology platforms, data and information — and the organization's understanding of all of these — form a consistent, cohesive, and adaptive whole.

2.1 Dimensions of Variation

Although this definition covers a broad range of interrelated aspects, we can largely reduce it to the three main conceptual dimensions of variations in Figure 1: *vertical* – different levels of abstraction of the same subject; *horizontal* – different subject areas or views that are not, of themselves, more or less abstract than others; and *variants* – different systems, actual or imagined, as-is, as-was, or as-could-be, including variants that arise within a family of related systems configured to different needs.

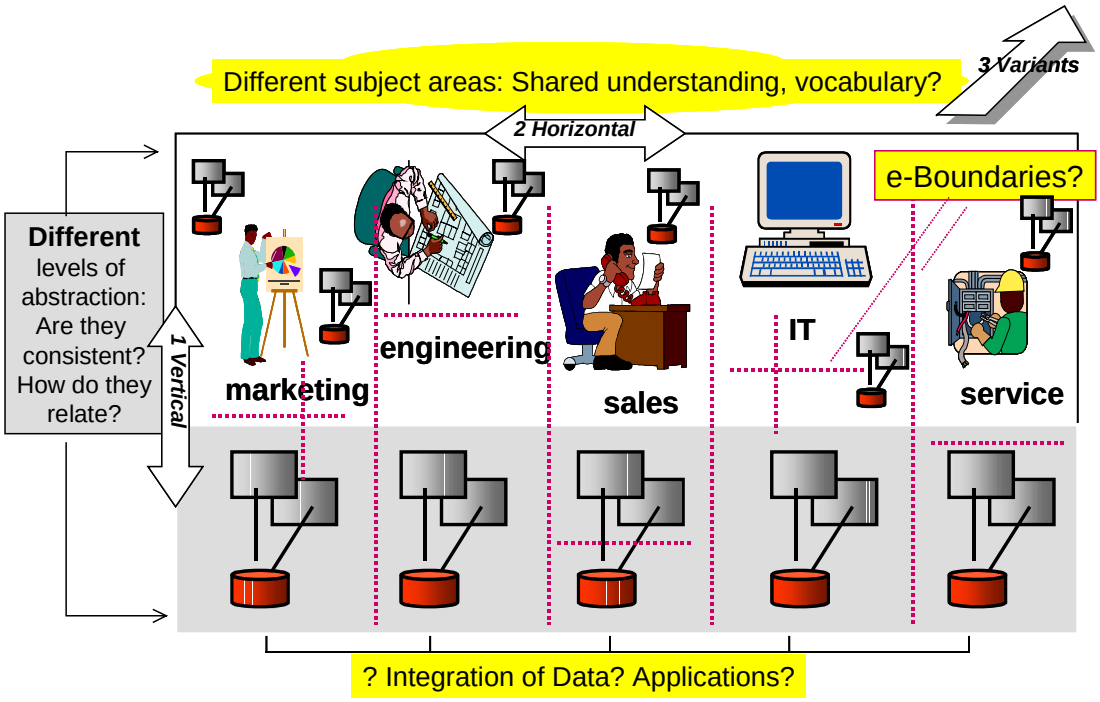


Figure 1 Vertical and horizontal dimensions for integration

Vertical dimension: this comes from different levels of abstraction of the same system[2], from physical data models, logical data models, networks, applications specifications, component assemblies, business process models, and business goals and strategies.

An interface specifies observable behavior, a class provides a particular implementation of that interface. A large-grained component might abstract multiple tiers and specify a single entity with

[2] *Vertical* variation often (not always) corresponds to introducing a new domain into the description of one subject area: *"A asks B to do X"* abstracts away communication. *"A sends a message with request X addressed to B via the post office; the post office delivers it to B"* describes the same action but includes the communication domain.

a user-interface and some persistent state; it is realized by a composition of a thin-client, a stateless business-tier component, and a database. A high-level business action such as *fulfill order*, has its abstract specification; it is realized by a particular detailed process involving collaboration between multiple software systems and people. Summary or aggregate information from a data warehouse can be "drilled down" to see detailed constituents.

These "vertical" points must relate correctly to each other ("integrated") if the detailed levels are to faithfully provide what their more abstract levels promise, and for platform-independent verification or compliance test suites can be well defined.

Horizontal dimension: these are different views, one no more detailed than the other; e.g. different subject areas or domains, whether business (like marketing, engineering, and sales) or technology (like performance monitoring and security)[3]. Marketing teams track product trends and define product requirements, while engineering teams develop new products. Each area has its own narrow view of the enterprise. But marketing and engineering teams both deal with overlapping views of product specifications and release dates, which raises an integration issue: do the different areas, people, terminology, business processes, and software systems, actually work together?

More concretely, a data warehouse performs "horizontal integration", combining data from the marketing and engineering databases, reconciling differences in syntax, semantics, and data values. Bridges between two technology platforms, each offering its own guarantees about clients and servers, play the same role. So horizontal integration has analogs at all levels of the enterprise, including applications, business processes, domains, and even business strategy.

It is worth noting that horizontal integration is needed mainly to provide vertical integration: you build a data warehouse because the integration realizes a higher-level information model which abstracted away the split across the data sources; vertical integration relates solutions to problems.

Variant dimension: the parts of an enterprise and its systems have many variants. Thus, models of system variants across time (as-is and to-be), branch locations, product-family member, or technology platform are about this dimension. This is technically similar to "horizontal" variation, but is more concerned with configurations, variants, evolution, and evolution-focused architectural rules and modeling standards.

2.2 Integration

Any point in the space on Figure 1 could involve both humans and machines. The dashed lines indicate that e-business boundaries might fall at arbitrary places: any combination of subject areas (marketing, engineering, infrastructure management) at any level of abstraction (from high-level policy through detailed operations), all with variants and changing with time.

To integrate means to join parts while properly reconciling overlaps and differences between them. Any two distinct points in this space represent an integration problem in this broader sense, across different subject areas, different levels of abstraction, different technology platforms, or different points in time. This is a complex picture, and well structured models can help people comprehend, manage, and evolve it more effectively.

[3] Such subject areas include typical business domains, like finance, engineering and marketing, as well as technology domains such as distributed systems, systems management, or user interfaces. Subject areas share common parts.

3 *Modeling the Enterprise*

A model is a formal description of some key aspect of a system, from some viewpoint. As such, it always presents an abstraction of the "real" thing, by ignoring or deliberately suppressing those aspects that would not interest a user of that model. Different modeling constructs focus attention by ignoring different things. For example, an architectural model of a complex software system might focus on its concurrency aspects, while a financial model of a business might be concerned with its projected revenue. Model syntax includes graphical, tabular, and formal text.

Models should be explicitly represented and managed, precise enough to at least enable unambiguous communication analysis, and abstract enough to focus attention and provide insight. A model is simpler to comprehend than the thing it represents; well-structured models can make complex systems comprehensible. Modeling helps users achieve consensus about what exists or can be built, since it provides a concrete focus to agree and disagree about. A good model does not have to be executable, but it must be readily validated against concrete examples[4].

Model ≠ UML. We can use models at all levels, from business strategy and process through software applications, databases, and networks. More familiar models cover application specifications, software architecture, and network and database designs, as shown in Figure 2. At higher levels, different aspects of business strategy may be modeled in spreadsheets (numeric metrics), QFD matrices (stakeholder objectives, software requirements, design alternatives), and Balanced ScoreCards (establish strategic intent and motivate performance goals).

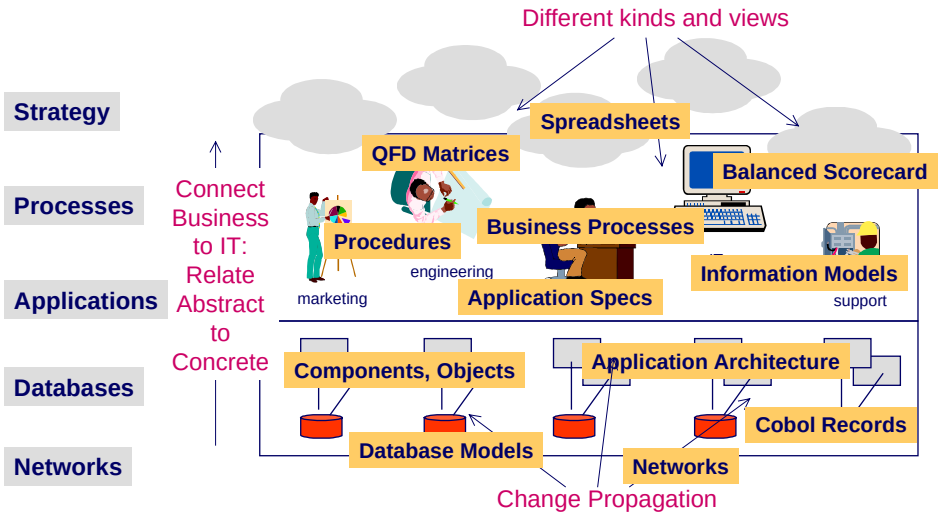


Figure 2 Levels and Kinds of Models

[4] A model need not be executable for instances to be checked against it. An "executable model" e.g. of *shortest path*, is a program, regardless of syntax. We must be *very* careful about which aspects of such a model are normative, and how we define conformance. A declarative model simply defines shortest path ("path = …; path length = …; shortest path = no other path is shorter"), it may not be executable, but it provides a basis for verification or testing. Consider a concurrent system with non-deterministic scheduling, where key observable properties (e.g. termination, computed result, timing) depend on the scheduling of actors created during program execution. How would you test a 3rd party implementation against an "executable model" which, by definition, had to make specific scheduling decisions in order to execute?

4 What is Model-Based Specification, Design, Integration, and Refinement?

This section contains a short example of how models from different (vertical) abstraction levels, as well as from different (horizontal) perspectives tie together. The model-based scenario covers traditional signature-based interface descriptions, implementation-independent specifications of assumptions and guarantees, designs of components, eventual implementations, and tests.

Interface Signatures: Signatures do not adequately specify an interface. Figure 3 shows three specifications. *Thingami* would not be trusted because we don't know what it does. *Editor* might be used because we guess what it does, though subtleties of a simple operation like *addElement* remain unclear. We can guess what *NuclearReactorCore* does but would stay away in fear.

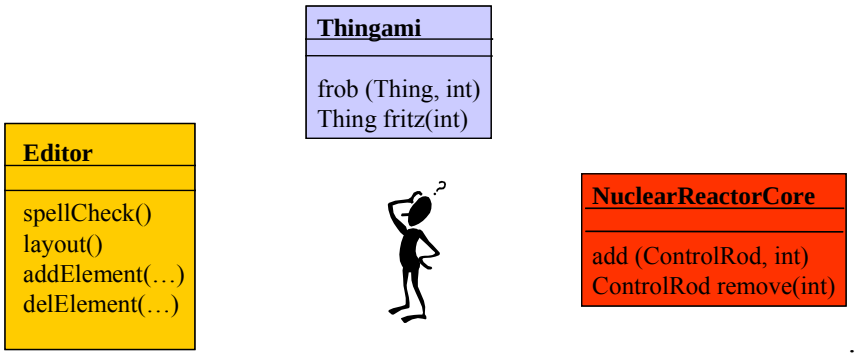


Figure 3 Signatures are not enough

Model-based specification defines behavior precisely, formalizing in the model all terms that must be defined for those behaviors. This includes at least a definition of all inputs, outputs, and state variables, for each operation, abstracted from concrete representation. .

Figure 4 shows how behaviors of the *Editor* are specified in terms of the model. We might start with a (simplified) informal specification of the net effect of an operation such as *spellCheck*. This is written in the style of a test specification that should hold upon completion of *spellCheck* for any correct implementation, regardless of algorithms or data structures:

> *All words are correct by the dictionary*.

To formalize this spec requires that terms like *words*, *correct*, and *dictionary* be clearly defined. This is done in a UML type model, drawn here in the 'attribute' section of the *Editor* type since they are abstractions of the state of the editor. Once the terms are defined clearly, the operation specification itself can be made clear in prose, or in the Object Constraint Language (OCL) shown at the bottom of Figure 4 as a *post-condition*. Moreover, the type model terms must also be used consistently by all other parts of the specifications, such as other operations.

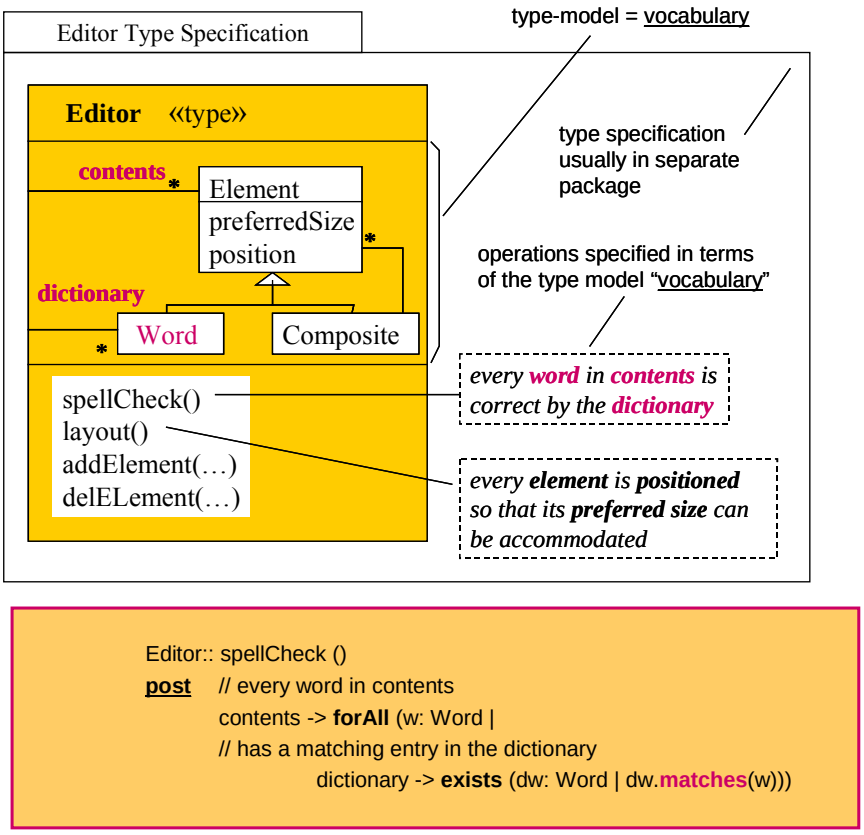


Figure 4 Model-based specification

Model-based Design or Integration. Design is about conceiving an assembly of parts that realizes a specification. In Figure 5 the editor is designed as an interacting editor core, spell checker, and layout manager, whose collaboration is a valid refinement of the editor spec.

The models of these three components (spell checker probably includes a model that includes some definition of dictionary; spell checker and core both include some model of word; layout manager has a structural geometry model) must be integrated and mapped to the editor spec in the "refinement" model. The refinement model is in a separate package; it imports the spec and the design, and adds the refinement relationship and supporting mappings between the parts.

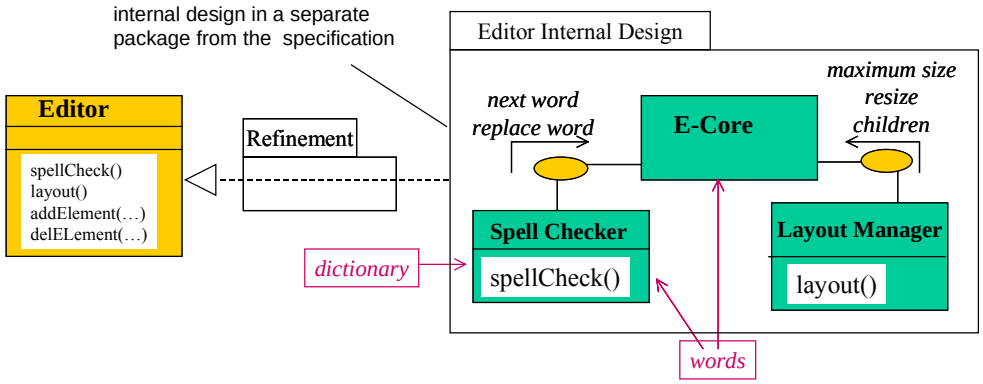


Figure 5 Model-based Design

This view is fractal. The editor core has two interfaces each with a different model-based

specification. The Spell Checkable model is of an indexed sequence of words, and the Layoutable model is of a tree of geometry objects with descendants. Each is specified *independently*, as shown in Figure 6, and has its own test suite. To specify the editor core, we must include not just these two interface specs, but also how their models are related to each other e.g. the relationship between the layout view and the sequence of words that are spell-checked; and set up explicit or implicit relations between their operations e.g. replace word may require re-calculation of layout.

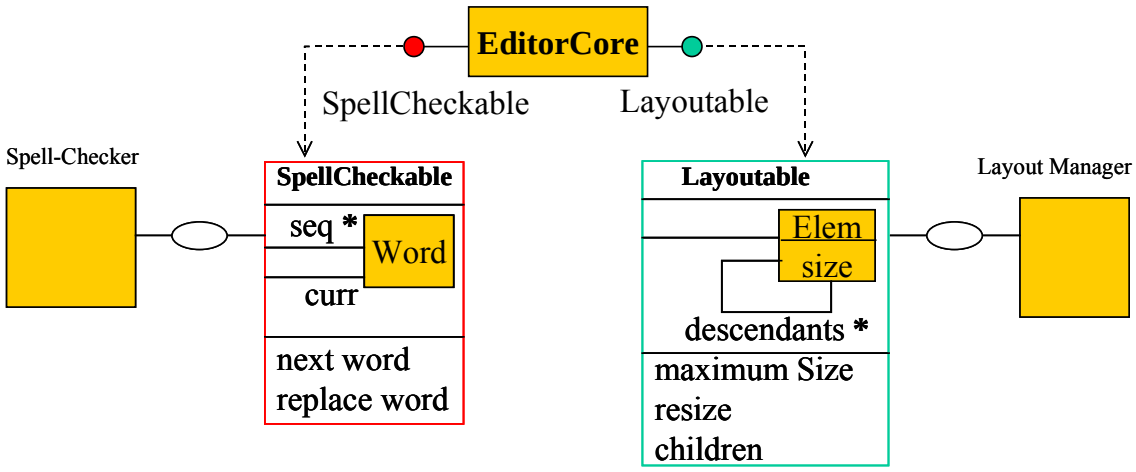


Figure 6 Model of each Interface in the Design Collaboration

Implementation and Test: An editor core implementation might choose a concrete representation consisting of a tree of document elements (paragraphs, words, tables, figures) combining together their logical content with their geometry information[5]. Testing the Spell Checkable interface of this implementation would require a mapping from the concrete tree structure to the word sequence model used to specify spell checking, as shown in Figure 7.

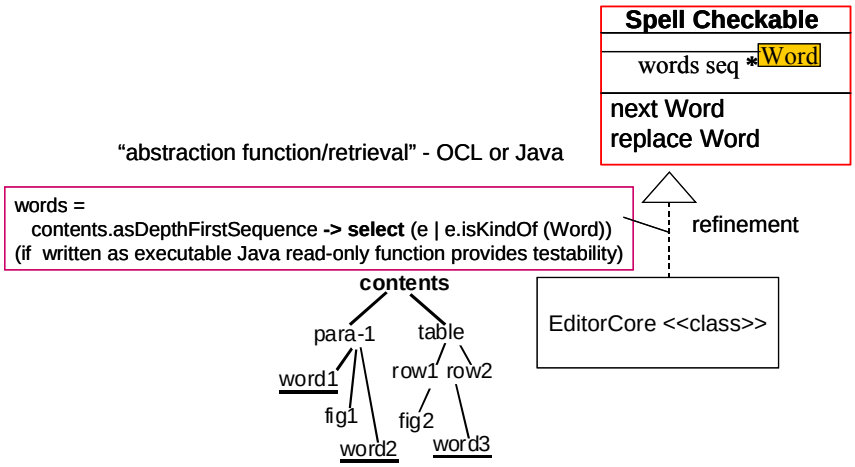


Figure 7 A refinement mapping

Perspective: Note that this example illustrates both horizontal and vertical integration. It shows:

[5] An alternate implementation might maintain separate representations of the model of each interface, with code to maintain the required consistency relationships between these two representations. The typical federated enterprise architecture is a large-scale example of this alternative architectural style.

- Model-based specification adds a degree of precision that is absent in signature-based interface specifications. Ambiguous and length prose explanations become crisp and focused, and inconsistencies are uncovered quicker, since the terms used for all behaviors must be in terms of a shared common models that defines what they mean.

- Good model-based specification are like implementation-independent test specifications – provided they are interpreted carefully to not unduly restrict valid implementations.

- Model-based design assembles parts, each with models of its interfaces, in a way that can be shown to refine the specification of the whole. The models of the parts have to be related to each other (i.e. integrated) to show the relationship to the whole.

- Each interface of an individual component has its own model-based specification, independent of its other interfaces. The specification of the entire component as a box must relate the models of the different interfaces in ways unique to that component.

- An implementation chooses its specific data representations and algorithms. In our example, the interfaces were specified in terms of an abstract model of state, and algorithm details were elided in favor of a *post-condition* test. In order to use implementation-independent tests we need appropriate mappings from the implementations to the abstractions used in the specs.

This small example illustrates several aspects of the MDA goal of platform-independent models and test specification. The MDA allows somewhat more flexibility between the platform-independent specification of interfaces, the platform-specific form of those interfaces, and the eventual implementation of those interfaces.

5 Model-Driven Integration Challenges

Models and integration at the enterprise scale pose different challenges from models of single software systems. One of the characteristics of the multiple-viewpoint approach outlined in Section 2.1 is that descriptions of the same or related elements can appear in different viewpoints and must co-exist. Further, since different viewpoints can impose contradictory requirements on the system, consistency across viewpoints becomes an issue. Figure 8 outlines some key challenges of modeling at the enterprise scale. These challenges include federation, integration, architecture standards via shared models, seamlessness, and synchronization.

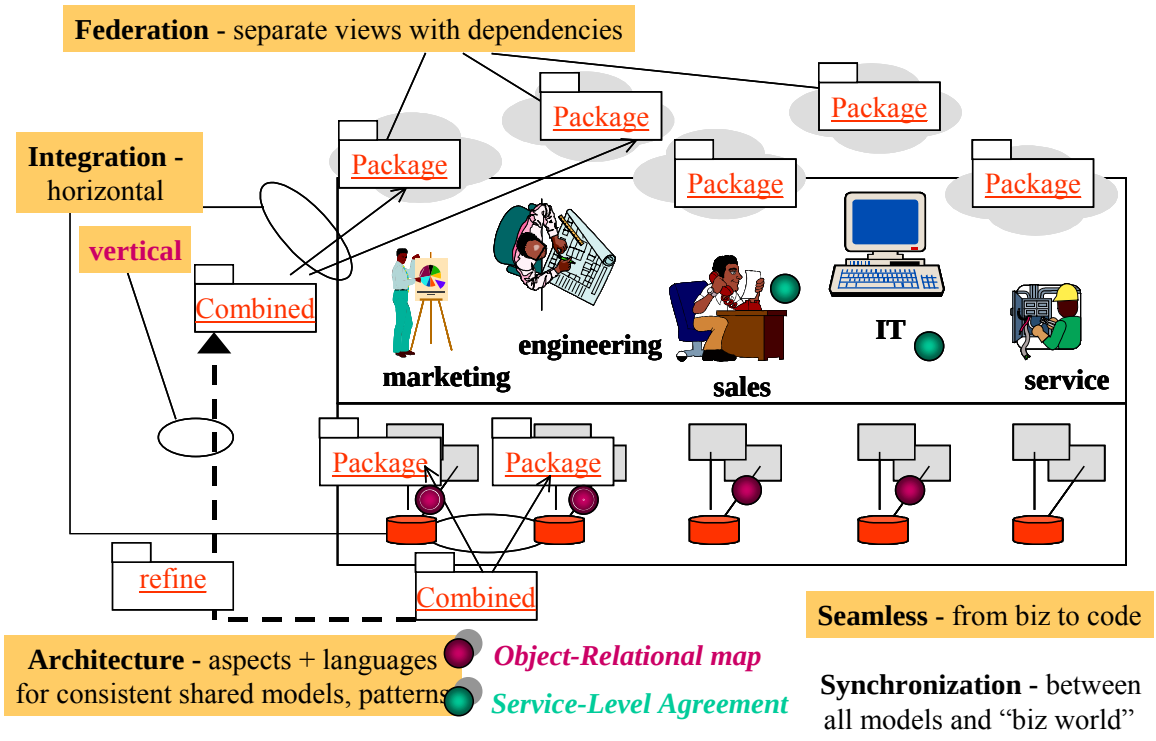


Figure 8 Challenges of Enterprise Modeling

5.1 Federation

Do not build a single integrated model of your enterprise. Even if you did not run out of wall space, you would not provide a natural description of any single concept to diverse horizontal or vertical groups. A single model cannot describe *Product* for both sales and engineering (horizontal separation); or include its network bandwidth and sales projection aspects (vertical separation). Enterprise scale models must be federated, separating both horizontal areas and vertical layers.

U-1. The key modeling concept here is an improved UML package. Each package provide its own views of the same *Product*, and can define those aspects that are relevant to it. (This separation goes hand-in-hand with U-2,3,4). The same separations apply to packages of software: different interfaces of an object are better defined separately, interface definition packages are separated from implementation packages. Package location is irrelevant.

5.2 Integration

Separated models typically overlap in places. When engineering and marketing staff collaborate, they must have a common vocabulary and process to work together; when a new data warehouse pulls in information from engineering and marketing databases, the two schema must be integrated – both examples of horizontal integration. Or, when the information gleaned from this data warehouse is used to shape marketing and engineering strategies, the warehouse model must be clearly related to the strategy models – an example of vertical integration. UML and MOF could use some strengthening to support the items below.

U-2. The basic concept here of is an improved UML package import, allowing (a) packages to share an arbitrary model fragment by importing it from a shared package and decorating it locally with further model elements; and (b) conversely, we need to model the "integration" of two separate packages in a single separate package that imports them both and adds more information. The current approach of being forced to define a subtype in order to say something more about an existing type has problems with scalability (explosion in number of subclasses), understandability (recurring patterns of dependent types are lost), as well as semantics (subtypes should describe subsets). There is no reason to force everything that can be said about a type or action into a single package: consider the separation of core functionality, reporting, event-logging, security, and performance aspects for a *placeOrder* action.

U-3. Horizontal integration composes two packages into a third package, where correspondence between elements in the two imported packages is somehow established in the third. For example, the marketing and engineering models are imported into an integration model package, where the two views of *Product* are reconciled by adding new attributes, associations, or constraints that cut across the two models.

U-4. Vertical integration is provided by an explicit "refinement" model package that imports and maps between concrete and abstract model packages. For example, a domain model for marketing may assert a static relationship between product positioning and market information. A specific realization may have separate business systems for market information and product positioning, interacting to maintain that relationship. The refinement package relates these two models to each other. We need to define different refinement relations and the guarantees they preserve.

U-5. Variant integration requires having precise definition of model configurations and builds, and identifying 'correspondence' between model elements across variants. Publication, versioning, and configuration are package-based. A package is published as a unit, and can then be imported by any other package regardless of location. Packages are the only unit for publication. A published package is immutable and only imports other published packages. A configuration is a published package. Builds and code generation are integrated, at least conceptually, with configuration.

U-6. Model integration demands clear semantics. The core constructs should be simple, with minimal constructs, and with explicit semantics: instances of objects and actions, with corresponding object specifications and action specifications (both subject to refinement), and specified constraints on the instances from the specifications.

U-7. As far as possible, new constructs should be defined by translation to the core[6], rather than have independently defined semantics. For example, UML 2.0 asks for refinement semantics. If the core was just objects and actions this would be simple; if other constructs (state, activity, transition, event, …) were translatable into that core much of our work is done. But if state, activity, transition, etc. were independently given semantics we would have to deal with combinations of refinement across states, activities, post-conditions, attributes, associations. In most cases, new meta-models constructs or extensions can be defined as convenient modeling syntax (abstract or concrete) with a translation to the core abstract syntax, easing integration and interoperability. After all even the Corba Component Model specification expressed its component constructs largely as translations into idiomatic patterns of standard IDL.

U-8. A fractal view of objects and actions, where zooming in exposes finer grained objects and actions, perhaps from other domains (e.g. introduces distribution aspects where the zoomed out view abstracted the distribution domain), greatly simplifies defining new constructs and notations that can translate into objects and actions, and makes vertical and horizontal integration of models simpler.

U-9. UML's activity model and its associations with state machines is broken. Refinement provides a much clearer definition, since what is considered as a single action at an abstract level may need to be reified at the refined level into an object with a visible lifecycle and states. When you realize an abstract action (e.g. *purchase*) with some finer grained actions (e.g. *order, deliver, pay*), the refinement model needs to map some composition of fine-grained actions to the abstract; likewise, at the realization level, you will need to reify the (now-ongoing) action *order* into an object with states, since the intermediate steps of the abstract action are now visible. An activity model then simply shows what composition of finer-grained *actions* (the ovals) realize the abstract *action*. The lines joining activities essentially correspond to *states*[7] at the realization level. Moreover, activity models should share the same generic pattern of component-port-connector-assembly as components and classes.

U-10. At the business level, there will also be a need for modeling constructs that easily represent things such as business rules and policies, in a way that can be clearly refined to models of software systems with a more traditional computation model.

5.3 Shared Architecture Models

To manage models of this scale you must have a coherent modeling architecture, with standard ways to describe shared concepts, rules, patterns, frameworks, mappings, and generators. For example, if you develop different applications with object models mapped to relational tables, you should use a single object-to-relational mapping architectural style from a shared model as a refinement framework. If all your enterprise models maintain a separation of information, engineering, and computational views and then integrate them in particular ways, this common structure should itself be modeled in a shared architectural style model as a package pattern.

[6] e.g. State = Boolean attribute + state invariant; Association = pair of inverse attributes; Activity = n-way Action; Aggregation = tree-structured association with lifetime constraints. Layered translation is an old approach to portability and interoperability e.g. compiling Ada and Java both to the JVM.

[7] Not to transitions, as currently in UML. Profiles such as EDOC are consistent with our interpretation, and layer some additional semantics to arcs related to communication about that state information.

The same principle holds for business models. When sales and information technology departments enter into different forms of service-level agreements, they should be able to adapt a single, generic service level agreement model. Or, all business areas and processes should follow a common set of rules about escalation of certain types of issues.

Lastly, the same principles hold for the definition of the modeling languages themselves i.e. the structuring of the meta-models. The UML core and MOF need to provide facilities for sharing definitions of partial models and patterns that recur in languages and inter-language mappings. These patterns range from fundamental (the *Descriptor-Occurrence* relationship between any specification element and its corresponding instance elements), to modeling constructs (translation of aggregation/composition into a basic association), to a standard package structure for separating concrete syntax, abstract syntax, and semantics. The current UML core and MOF needlessly prohibit many of these from being shared since they impose "strict" 4-level meta-modeling rules.

U-11. Effective shared definitions is key to architecture. Package imports should be improved so a package can be extended as a unit, decorating any model fragment (classes, instances, interactions, etc.) within that package with new properties (attributes, associations, inheritance, postconditions, stereotypes, etc.) This allows separation of different viewpoints or aspects of the same thing. The semantics of 'joining' the shared definitions with separately introduced extensions should be made clear.

U-12. UML patterns need improvement to consolidate concepts of framework, pattern, and template class. None of these constructs is capable of describing a pattern of refinement, which is a cross-model pattern. They should all be unified into the concept of a package framework – you can import that package and substitute certain elements; the result is a composition of that package contents with your package, joined by the substitutions. The pattern definition can parameterize any model element and identifier with substituted elements. This construct unifies template classes, template collaborations, refinement patterns, patterns of business process, package structure, and more.

U-13. Frameworks themselves can be checked, refined, and composed like any other model e.g. an abstract subject-observer framework simply specifies a static invariant to be maintained. One refinement may choose a registration and notification protocol; a further refinement may choose a "2-way-link" framework to keep the connection.

U-14. Every patterns makes some assumptions about the context in which it is applicable, and the result of using the pattern is valid provided those assumptions hold. This means patterns must also specify conditions that must hold true of the substituted elements and their context, much like a (design-time) precondition. For example, the subject-observer pattern is applicable when subject's attribute defines inequality (for 'change'), and has an equivalence with the type of its observer's attribute (to define the result after 'update').

U-15. Supertypes are an important way to define shared architectural constructs. Sets of related supertypes should be defined and extended in a way which preserves the guarantees made by the supertypes. Unfortunately, most OMG meta-models, profiles, and domains instead make extensive use of *oclIsKindOf* and *oclAsType* i.e. type assertions and type casting that undermine the value of the supertypes. Arbitrarily picking an example from the CWM spec:

```
context FeatureNode inv:
self.feature.oclIsKindOf(BehavioralFeature) implies
  (if self.feature.ownerScope = #instance
    then self.argument->size =
          self.feature.oclAsType(BehavioralFeature).parameters->size + 1
    else self.argument->size =
         self.feature.oclAsType(BehavioralFeature).parameters->size
  endif)
```

Doing this costs us in two ways. First, the recurring pattern of several dependent types is buried in subtype OCL constraints. Secondly, there is serious duplication of OCL constraints in subclasses that are identical except for renaming. The solution is to make the type dependencies explicit using frameworks or package patterns.

U-16. The word "architecture" itself continues to be used with widely different meanings, even in the context of MDA discussions. We do know this much: architecture covers different viewpoints; architecture guides design; design is about refinement; designs and specs are both expressed in some language; and some (parts of) design models can be automatically generated. The concepts of architectural style, refinement, viewpoints, viewpoint languages or meta-models, and generators should be properly integrated. We discuss how to do this in more detail in Section 6.4.

5.4 Seamlessness

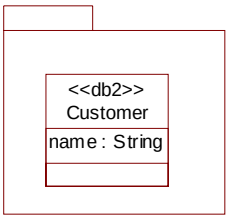
You want to smooth over the "seams" between different kinds and levels of models, such as process, component, object, and data models, and across different business areas. This means handling more than one kind of meta-model, and relating models that are based on different underlying meta-models.

U-17. We must handle multiple modeling dialects, each clearly defined, with their interrelationships clearly defined as well, and consistently relate models to each other within and across dialects. Note that, at the meta-level, these dialects themselves form a space much like in Figure 1. e.g. a reflective language will express a model differently from a non-reflective one; the refinement relationship should be defined on the language constructs (meta-models) themselves, where possible. Alternately, languages may have mappings between them that are not refinements, and these relations could be defined in (meta-model) packages that import the meta-models being integrated e.g. UML, IDEF0, and EXPRESS are related in some way that must be modeled.

U-18. Once again there is a lot to gain by having a small core in the UML, such as fractal objects and actions (Section 6.2), as it enables easier mappings to and across dialects.

5.5 Structure of Meta-Models

Given a package from your favorite tool, does the familiar box labeled Customer represent a UML class? A MOF class? Does the <<db2>> on the box mean IBM's definition of the <<db2>> stereotype? Someone else's <<db2>>? Does the <<…>> even mean UML stereotype?



It is nice to treat a package as a collection of model elements, provided every language construct used in that package is unambiguously identified with a definition of that construct.

U-19. A language or meta-model is defined in a package that can include the rules of concrete syntax, corresponding abstract syntax, and semantics of each language construct. That package can use all the usual package structuring facilities. A model – i.e. an instance of that meta-model – must import the language definition package so every language construct used in the model is identified with its definition in the meta-model.

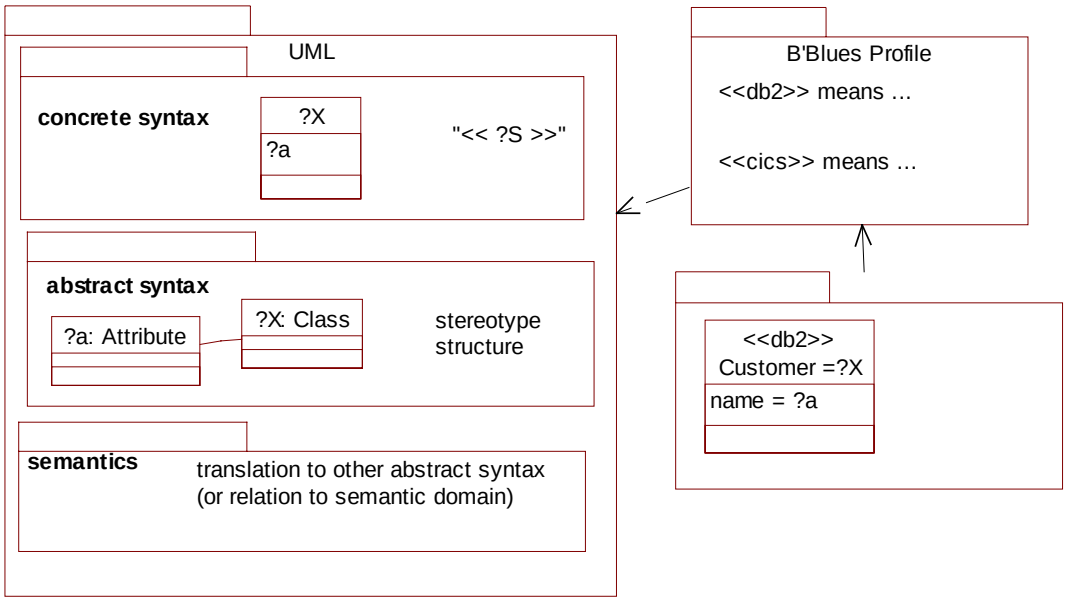
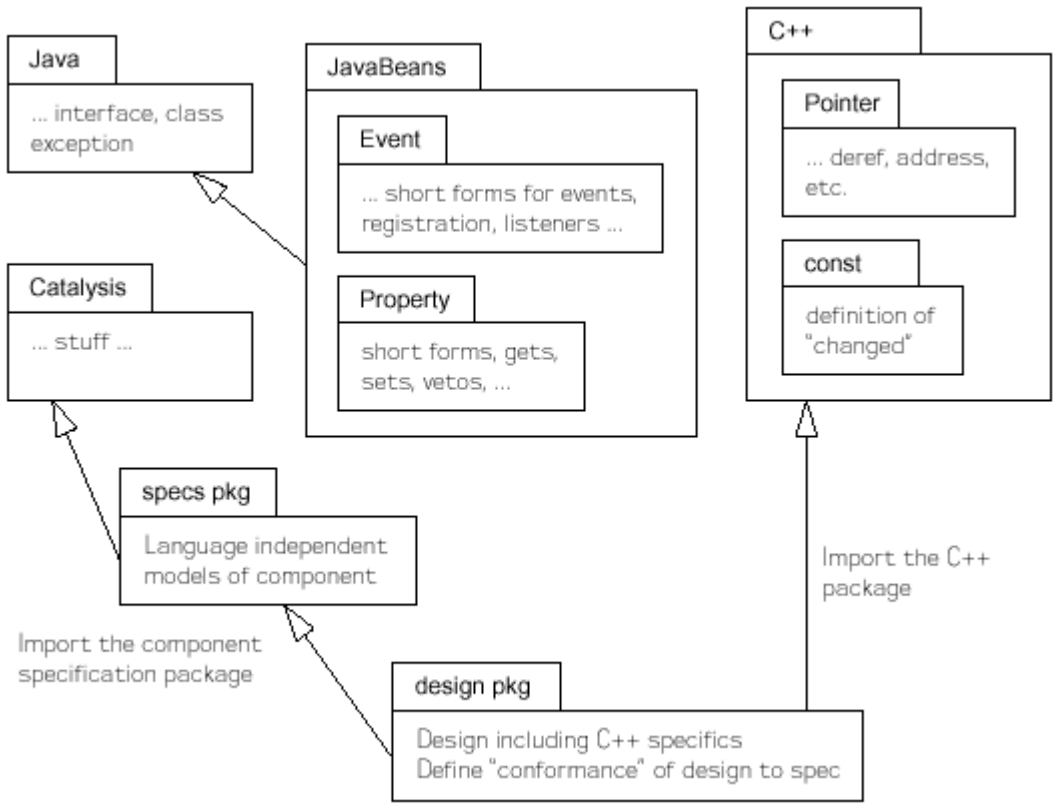


Figure 9. Language Definition and Usage

Figure 9 sketches how this could be structured across meta-models and models.

UML concrete syntax (a class box, or a stereotype) is defined in a package. The abstract syntax (constructs such as class and its attributes, or the inheritance structure for a stereotype) is defined separately. The mapping of concrete syntax to abstract syntax should be explicit, indicated here by the placeholders "?X" and "?a". The semantics of any construct are defined either as a translation into a lower-level abstract syntax, or by its relation to an explicit semantic domain. The former method may be preferable to layer new constructs on a small core, where the core itself might use an explicit semantic domain (see U-6-U-7). A particular profile is defined in its own package with its stereotypes and other extensions. A profile is imported by a user model, so all constructs in the user model are defined within that profile's UML dialect.

The basic idea is quite similar to programming. When you write a source file in C++, you must (a) indicate clearly that it is C++, and sometimes even indicate which version of the language or compiler can handle it ☺; and (b) #include any header files that define macros that you use. Facilities for packages and imports leads to a structure of both incremental language definitions (meta-models) and language uses (models), as shown below in a figure from [Dso98].



5.6 Synchronization

One of the challenges to making MDA real is maintaining a high degree of confidence that models remain true descriptions of the things they denote. So, models have to be kept in sync both with each other across levels of abstraction, and with the real world that is being modeled.

This means our models must have some clear linkage to the things they designate. For example, changes to code should be reflected as changes to models that are supposed to represent that code (and vice versa). But this synchronization is not just a code issue. Business processes might change without their models being updated. It's a good idea to detect such divergence and take corrective action. This synchronization is in both directions: changes to business models should be deployed to the actual processes, which might involve upgrades to hardware, software, and personnel.

> **U-20.** A clear refinement relationship between concrete and abstract descriptions is essential to define inter-model synchronization. A clear link between models and the individuals they denote in the "real world" is essential to keep models in sync with the world. The concept of Architectural Style (Section 6.4) lets us regularize some of the refinement patterns that are used, and make this synchronization more practical.

5.7 Interoperability – Design and Run-Time Metadata

Design time repositories will maintain rich information about models and inter-model relationships. Today's run-time repositories maintain information about deployed objects and their signature-level interfaces. However, run-times are getting increasingly flexible, with self-description and auto-discovery of services, negotiation of protocols, and even run-time generation

of adaptors. Today's commercial adapters are at the level of data types and protocols; it is entirely feasible for more sophisticated protocol and service adapters to be generated on-the-fly by smarter mediators. So the information needed at run-time is potentially much more than signatures.

For example, when generating the adaptors between two components, a mediator such as a smart trader service may need to recognize the modeling language and version of the language used for each component, uncover relationships between <<stereotypes>> used in each model, discover that each has its own view of a shared *Customer* concept, and reconcile those views in the adaptor.

Agent behaviors such as negotiation need explicit machine reasoning about models as well as about the inferences that can be drawn from a model. A smart agent needs to understand the languages used for each model, the ontologies described in those languages and how they are related, and what implicit conclusions can be drawn from the explicit model definition.

U-21. Both design and run-time repositories should have full access to model information, including inter-model relationships and the mappings used to generate code.

U-22. The modeling architecture (and hence, meta-models and models) should clearly distinguish conceptual models from callable interface specifications, such as may be used for a design or run-time tool. Conceptual models often introduce model elements, such as abstract state attributes) that were not meant to be exposed to a published interface.

U-23. Interoperability should not be reduced to a leas-common-denominator. Instead, it should allow a model expressed in a given modeling language or dialect to be at least partially understood by a machine that did not understand that entire language or dialect. For example, a design-time search for components that meet a certain real-time spec, in a tool that did not fully understand the real-time profile, should still retrieve candidates that appear to meet all other search criteria. Similarly, a run-time agent that understands attributes but not state machines, when faced with a state-machine model, should be able to reason with the attribute-equivalents of those states. This requires that language definitions themselves be structured to share partial language definitions and extend or translate into other languages.

6 Key Concepts and Principles Elaborated

6.1 Composition or 'Join' of Model Fragments

One of the keys to architecture is shared definitions and rules. You have probably surmised by now that we want to share and reuse partial models at many levels of granularity.

- A subclass may specify a property which is also specified by a superclass; we need to 'join' these to define a resulting subclass value for that property. For example, it is a common sub-typing discipline for a superclass to specify a partial guarantee for a method, and a subclass to extend that guarantee with some addition pre/postconditions[8].

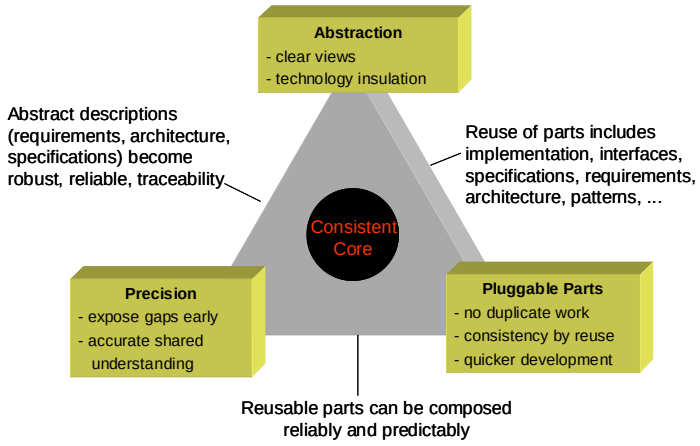
- Two different patterns may be applied to a class C e.g. *subject-observer*, and *proxy-remote*, and C's role in each pattern imposes different requirements on C. The different patterns can influence overlapping elements on C e.g. the roles of *subject* (with some interesting state attribute) and *proxy* (with at most a cache of remote state) interact with each when remote state changes. Hence we to 'join' what each pattern says about those elements.

- If a property of some model element is defined in two different packages, we need to be able to 'join' them in a package where both are visible; this is, in fact, a special case of applying multiple patterns.

6.2 Three Principles

Three simple core principles – abstraction, precision, and pluggable parts – combine to give more than the sum of the parts.

Platform-independent models are about abstracting away technology details. Business process models abstract even more details away. However, abstractions without precision are of very little value, specially within the context of MDA, so abstractions must be precise. Modeling at this scale, and dealing with the relationships and transformations between models, will demand shared definitions. All models, including business models, platform-independent specs, mappings, refinements, should be built by assembling smaller shared parts together.



[8] UML and MOF do not currently permit many useful partial specifications.

6.3 Refinement – Fractal Zooming In and Out

Figure 10 shows how any system (software or otherwise) can be modeled at a detailed, or "zoomed-in" level; and can also be modeled at a more abstract, "zoomed-out" level. The refinement model relates the two levels, providing full integration from domain models to code.

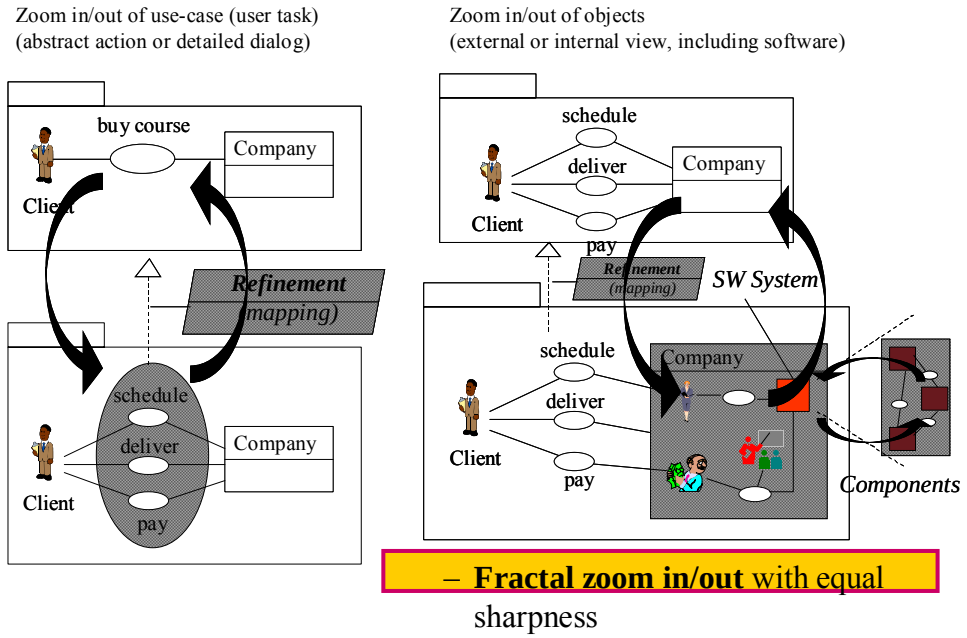


Figure 10 Refinement and Zoom-in and Zoom-out

For example, a sequence of business actions, *schedule, deliver, pay* for a training course may be a refinement of a single more abstract business action, *buy course*. Similarly, a software system may be modeled as a single abstract object from the outside, or refined and designed internally as a set of internal interacting parts. This lets you handle large components as simple connected objects; and larg-grained interactions as simple actions. It also lets you zoom into components to see architectural details of multiple tiers, with complex realizations of the simple abstract connectors.

If the refinement mappings are done well then the abstract behaviors become more "testable"; the (non-trivial) test fixtures for this would effectively push the concrete tests and observed behaviors up through the mappings as a 'simulation' of the abstract behaviors. For example, "company sell-stuff to customer" is not directly observable at a concrete level. Instead, you have lots of lower level actions (e.g. ordered, delivered, paid). A test fixture could map recognized patterns of those concrete events to an occurrence of the abstract "company sell-stuff to customer"; and map concrete distributed heterogeneous data representations to the abstract model attributes of company, customer, stuff; and check abstract state changes and action sequences. Conversely, you can define tests at the abstract level, and then "test the refinement by refining the tests".

The basic package structure for refinement is shown in Figure 11: the abstraction and realization models are in separate packages; the refinement package imports them, and models the refinement relationship between them. The note marked "full model of refinement" could include classes, attributes, associations, interactions, state charts, activity diagrams, etc.

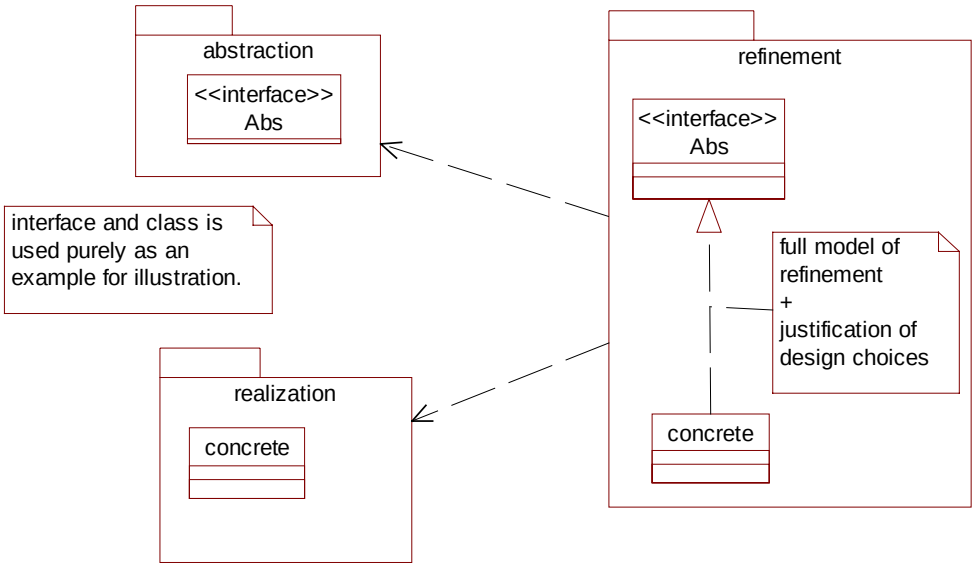


Figure 11 Refinement package structure

6.4 Architecture and Viewpoints

"Architecture" is one of those slippery terms that is sometimes used to bestow instant importance to powerpoint drawings; the MDA needs a far more precise definition, one where conformance of an implementation to an architecture at least has a clear and unambiguous meaning. What follows is a set of interrelated concepts that lead to a clear definition of architecture style and viewpoints.

Architecture constrains design. Architecture guides design. We know that architecture-free designs choose random and inconsistent ways to address similar problems; and that architecture-driven designs provide much stronger guidance in making consistent design decisions. In fact, we can recognize that, informally,

> *Architecture includes the set of principles, rules, or patterns about any system that keep its designers from exercising 'needless creativity'.*

Of course, the specific issues addressed by these rules and patterns differ at the level of business design, distributed systems architecture, and database design. Because the term 'architecture' is often used very broadly to mean some high-level description of a specific system, we will use the term "architectural style" to mean the rules, principles, and patterns that govern the design and evolution of that system at some level of description.

Architecture is about refinement. Refinement is a relationship between a particular design and its specification, and includes the mappings between them and justifications for design choices made. Designing, which is conceiving an assembly of parts that realizes a specification, is fundamentally about refinement. So, architecture style is intimately related to refinement.

Architecture style defines spec and design language. Both specifications and designs are expressed as models in some modeling language; the choice of language is related to the architectural style used. For example, all designs expressed directly in IDL share something in

common in their architectural style. Similarly, all specifications that are expressed as UML-EDOC profile models also share something in common in their architectural style.

Architecture style constrains refinement mapping. This follows from the previous discussions. All designs that follow a consistent set of rules relating their EDOC specifications to their IDL realizations, share even more of their architecture styles.

Some designs can be 'generated'. In some limited cases, a detailed enough specification can be transformed into a lower-level implementation automatically[9]; programming language compilers do this, and it can be done at least in some part for other kinds of models.

Spectrum of styles. So, clearly there is a spectrum of architecture styles, some more constraining than others. Consider a system in which you have *Persons* and some related set of *Accounts*, and attributes *Person.wealth* and *Account.balance* are specified as being in sync with respect to some derivation function. The implementation will have these objects distributed on a network. Here are four qualitatively different architectural styles for the design:

- The style has no constraints on design or spec: You can use any language to describe the spec and the realization, and can map between realization and specs any way you want.

- The style mandates a design language, but has no refinement constraints. This means you must use the language constructs from that style to describe your design, but the style does not constrain how you use those constructs to meet different parts of the spec, and so probably does not even define any language for the spec. *"EJB"* is such a style.

- The style defines what language you use for the spec and the design. Additionally, the specification of that style admits certain spec-realization correspondences, but not others. Hence, the style defines a predicate on refinement: it does not generate the design for you, it does have a well defined set of rules (such as may be checked in a design review, or by a tool) that can check whether your realization and the way it maps to the spec conforms to the style. E.g. *"EJB, but with no entity beans"* is an example of this kind of style.

- The style defines a full translation scheme i.e. the architectural style is fully determined, and is essentially a function (a compiler or generator) from spec to a realization of that spec. The correctness of the compiler's function serves as the refinement mapping. The least useful variants of this are when the spec language is isomorphic to the design language e.g. adding stereotypes in the spec language for every design language construct.

Styles will not be this simple in general, and will often be composed from other styles. The style specification in general can refer to elements of spec, realization, and the refinement mapping. A style may choose to ignore the specification side of the refinement, in which case it only constrains the set of realizations. A more interesting style for keeping attributes in sync could say:

> *Whenever:*
>
> - you have a requirement to keep 2 attributes in sync with each other (spec),

[9] Although this is neither doable or practical in the general case.

- and the attributes are used on 2 sides of a distribution boundary with frequent intra-boundary references and infrequent updates relative to inter-boundary references, (realization),

then:

- use the "2 copies + update protocol" pattern (realization).

Figure 12 shows a prototypical package pattern – specification, realization, and refinement, and its relationship to an architectural style. You create every design– whether a business process that implements a business requirement or a class that implements an interface – in a package, in the context of an architectural "style" package. The style can define the spec language and design language, including extensions like patterns or stereotypes; it can also define design rules that your refinement must conform to. The way you design to the specification – the refinement model – should conform to that architectural style: it is an 'instance' of the style, or a member of the set specified by that style[10].

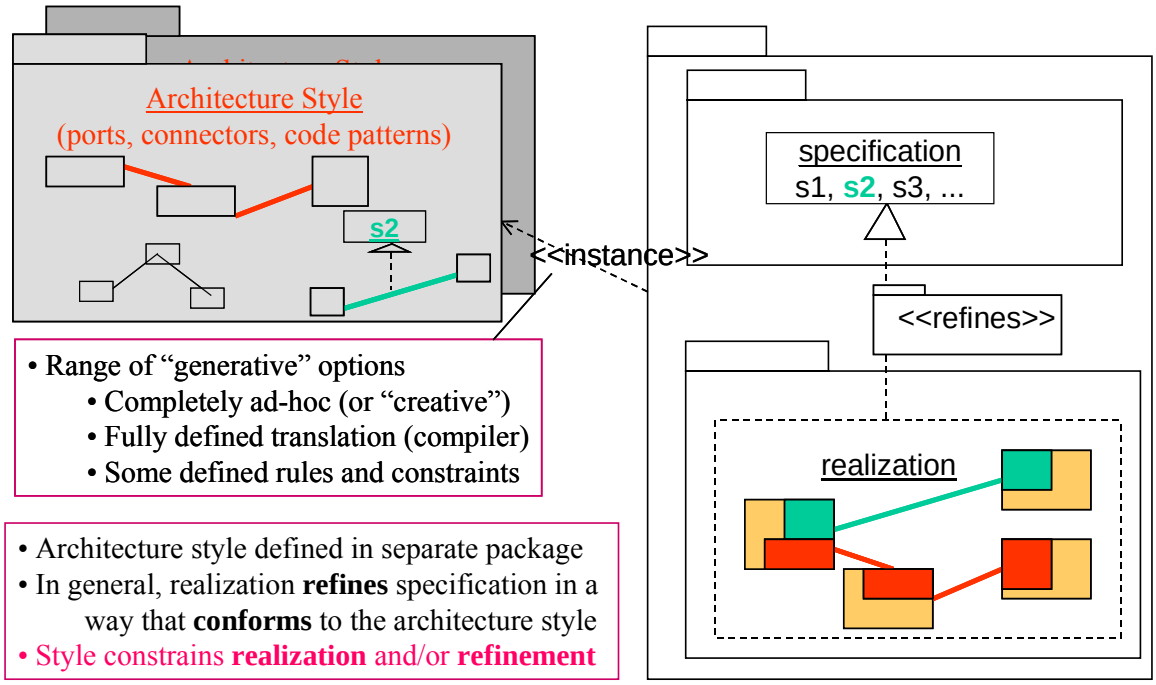


Figure 12 Architecture Style and Refinement

Viewpoints. A system can be described from many viewpoints; each defines what characteristics should be included in its views versus what issues should be ignored or treated as transparent. In Figure 13, if the model *Concurrency View* (which may itself include a refinement model mapping concurrency requirements and how they are realized) is an *instance of* architectural style *Concurrency Style*, we say it *conforms* to that style. Additionally, if we believe (by gut feel, check-list, design review, or harder stuff like full refinement verification) that the architecture is a valid abstraction of the implementation, we say the implementation conforms to that style.

This gives us a clean definition of architecture, architecture style, and viewpoints.

[10] Similar to Type denoting a set of objects, where the type specification defines required properties of those objects.

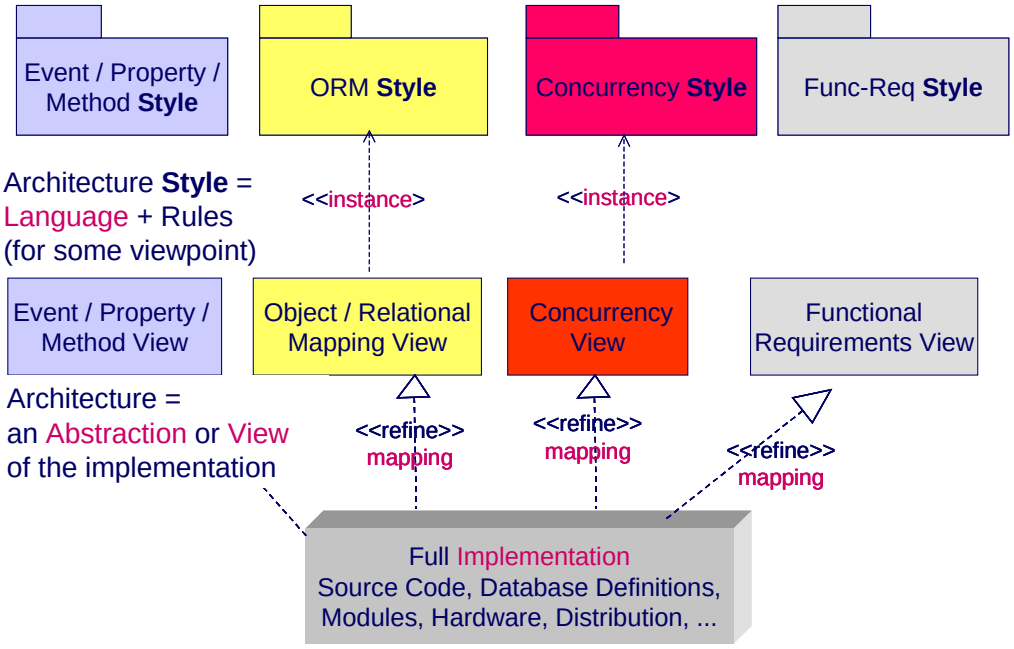


Figure 13 Architecture and Viewpoints

6.5 Canonical Package Structures

If MDA is to model refinement explicitly, and treat architectural style as a first class modeling construct, we arrive at a pattern of packages of specification, realization of that spec, refinement mapping, and architectural style conformed to. Figure 14 shows this recurring pattern: a Domain Model provides things known about the environment of the system i.e. things that can be assumed by systems deployed in this environment. The Process Model describes specific processes, manual and automated, *using* the specification of some machine(s), to define what effects that machine is required to have on the environment, when combined in some specific way with other machines and humans into the process.

The Machine Specification package specifies the machine independent of any particular process it may be deployed into, including just the minimal model of its environment. This package is distinct from the Process Model package, because the same Machine can clearly be used differently in different environments to solve different business problems.

Businesses solve their problems with some combination of business processes that integrate selected software applications. Clearly, you could combine many alternate processes and software applications to solve the same essential business problems.

This continues recursively, with the machine being designed as some assembly of sub-components, based on some component architecture (i.e. its assumed “platform” at this level of description).

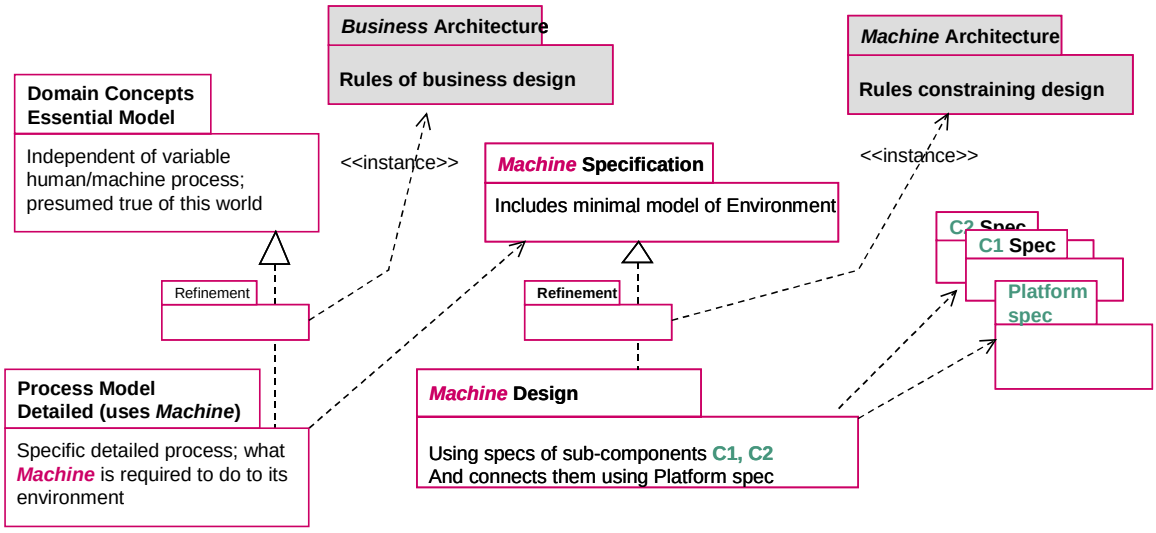


Figure 14. Canonical package structure – refinement and architecture style

Dashed arrows are normal package import; dashed arrow with <<instance>> indicates that the refinement model conforms to the architecture style spec; dashed arrows with solid heads are refinements, where the refinement model is in a separate package. Note that domain models are independent of any particular vendor's software or detailed process. Product models (called *Machine* spec) are based on some minimal portion of the domain model that they were designed to address. Your actual business solution will be the result of deploying and integrating selected products in concert with your particular business processes, to solve your domain problem.

7 Previously Asked Questions

Here are some questions raised by others who have read this paper, with some answers.

1. *This approach described refinement as a kind of 1-to-many relationship between an object (or, class) and it's "realization" (my term here). The object (class) in the higher layer is mapped to one, or most likely, more objects (classes) in the lower layer. The implication is that refinement is the only relationship allowed between layers of abstraction.*

   Refinement is many-to-many. It is the primary relationship (between models at different levels of abstraction) whose purpose is to establish that the realization meets the guarantees made by the abstraction. It can be as formal or informal as needed.

2. *Take, for sake of argument, a distributed automated teller system. At the level of abstraction of the banking business, there are things like deposits and withdrawls. Assuming that deposits and withdrawls are represented as "business objects" at the higher layer of abstraction, one implementation choice at a lower level of abstraction (call it the distributed communication layer for lack of a better name) is to map both deposits and withdrawls onto TCP/IP packets. I'd have a hard time calling a TCP/IP packet a "refinement" of a deposit or a withdrawl. A TCP/IP packet is a completely different concept in a completely different "domain" (using the Shlaer-Mellor sense of the word).*

   A good example to make this clearer. As you point out "TCP/IP" is a domain unrelated to Banking. But lets look more closely at the parts and how they are combined:

   ```
   package Banking_Biz { ... } // pure business terms

   package TCP_IP { ... } // pure technology terms

   package Distributed_Banking_On_IP {
      import Banking_Biz; import TCP_IP; do mapping.
   }

   package Design_Review_Package {
     import Distributed_Banking_On_IP;
     import Banking_Biz;
     assert Distributed_Banking_On_IP  refines  Banking_Biz (with models etc.)
   }
   ```

   Now, if you have full confidence in your mapping of deposits to TCP/IP (e.g. you have mapped oodles of other stuff to TCP/IP packets, tested it up the wazoo, satisfied yourself that the <X> → TCP<X> mapping is correct for any <X>, etc.) then you would not bother to say any more about this particular refinement; you have said all you need to in the mapping algorithm just once in the shared architectural style, and are confident it is right. In any case, a refinement mapping could just say "Because Joe says it's good" (seriously).

3. *Certainly, deposits & withdrawls are mapped on to TCP/IP packets, but the mapping is not one of refinement in the normal sense of the word.*

If I hand mapped deposit→TCP and withdraw→TCP, differently from each other and from all the other TCP mappings I've done before (i.e. I did not have a validated architectural style), then I would check the refinement claim in a design review or with tests.

4. *A consequence of this is that the zoom-in/zoom-out approach is not nearly so clean when the relationship between layers is many-to-many rather than one-to-many.*

   Refinement is a many-many relationship. Any design is a refinement of any number of (partial) specs. Any spec has any number of possible designs. For a given design, each design element will participate in fulfilling more than one part of the spec, and vice versa. None of this hurts zoom-in or zoom-out at all – you are just zooming along the refinements that interest you.

   There's a nice informal concept map at http://www.catalysis.org/overview/concepts/concept-map/graphical-concept-map6.htm, and a more detailed discussion at www.catalysis.org/books/ocf in Section 6.1.

5. *Figure 1 seems to imply that the different subject matters are at the same level of abstraction [which does not seem right].*

   Figure 1 is pretty informal. More properly, descriptions A and B may be of totally unrelated "domains" (banking and TCP), on they may overlap with neither being more abstract than the other (sales and marketing, in which case you have to do 'horizontal integration'), or they may be related by refinement (banking policies and rules vs. banking detailed business processes which supposedly conform to the rules, in which case you have 'vertical integration').

6. *[Are diagrams with a 1-to-1 mapping to code useful?]*

   Not as useful as they could be. However, if model relationships (like refinement) were done well, and architecture style is modeled so it could define model transformations, then the idea of code generation could be a bit more interesting. For example:

   1) relationships between models (e.g. refinement) are themselves 1st class models

   2) refinement is modeled (primarily) as a mapping from realization model to abstraction model (a formalist would want various proof obligations etc.)

   3) an architectural style denotes a set of <abstraction, realization, mapping> triplets

   4) architectural styles are themselves specified as constraints on that set.

   So, in general, the "human" still creates high-level models, defines architectural styles, and (depending on how generative that style is) creates+generates+checks the realizations and the refinement mappings.

7. *Our existing modeling technologies are not really sufficient to generate complete code. And I agree that advances can still be made in this area.*

   Yes. One reason I like the "arch-style-of-refinement" approach above is that it covers the

spectrum quite well, allowing "generation" but supporting more realistic in-betweens too.

8. *The concern here is that some elements of the transformation (the "architectural style") might reasonably be task-specific, not just Java-specific.*

   An architectural style spec defines which abstraction-realization-mapping triplets conform to that style. The style specification could refer to elements of all 3 parts (including task-specific abstractions if absolutely necessary, though careful analysis will often abstract the task-specifics to some more general characterization). Styles which only refer to certain (if any) aspects of the abstraction are thus a special case.

9. *That is, the transformation of *this* model to a Java-targeted model uses standard Java styles for classes 1-27, and [some other] specialized transformations for classes 28-31. If the transformation uses standard Java styles *without exception*, then the supposedly higher-level model was a Java model in the first place.*

   An optimizing compiler does this kind of thing by analysis of enough of the context of each piece of source, all the way through full-program inter-procedural analysis. So, I believe if enough context is modeled explicitly to capture the discriminator between your classes 1-27 vs. 28-31, this will not happen. Though that may not always be practical, the concept holds up. Some styles could be non-trivial to specify. Worst case, a style could even enumerate exactly what you said above ☺

   This is one of the reasons why 'patterns' should include as part of the pattern any assumptions about the elements substituted into that pattern and about their context. For the popular subject-observer pattern, these assumptions would model the presence of some state attributes A and B of subject and observer, the definition of "change" for A, and some equivalence relation between A and B which should be satisfied when the 2 state copies are "in sync". With these pieces of context to name and refer to, you can fully define and apply the subject-observer pattern.

   More to the point, nobody can generate adequate executable code from abstract models in the general case. Which is why I like this arch-style definition: it admits any degree of 'generativity', including a style which is *not* capable of generating the realization but *is* capable of a Y/N conformance check if any given abstraction + realization + mapping.

8 Glossary

[To be completed]

Package. A container and namespace for model elements, including relations between those elements.

Composition. The combining or merging of two models to produce a third, based on "joining" elements in the two models that are supposed to correspond to each other.

Join. An operation that combines the definitions of two model elements into a third.

Framework. A package whose elements include some designated as placeholders. The framework is applied by importing the package and substituting for those placeholders. A Framework can define the assumptions it makes about the properties of substituted elements.

Refinement. A relation between a concrete and an abstract model of the same system, where each abstract element is mapped to by some composition of concrete elements. More formally, you will have some rules to guarantee that some abstract properties are faithfully maintained in the concrete version.

Architectural style. A set of constraints on designs. A given design "conforms" to that style, or is an "instance of" that style, if it satisfies those constraints. A code-generator or compiler is the ultimate embodiment of a fully determined architectural style, leaving no creativity to the designer; other architectural styles constrain, but do not fully determine, the design.

9 Acknowledgements

Ed Barkmeyer, Steve Cook, Sridhar Iyengar, and Steve Tockey have provided useful and interesting comments on this paper.

10 References

[ODP] The ODP Reference Model

[Dso99] Desmond Dsouza, "Enterprise Integration", Software Development, 1999

[Soley00] Richard Soley et al, "Model Driven Architecture", 2000

[MDA01] Model Driven Architecture - A Technical Perspective, OMG AB paper, 2001

[OOPSLA99] "Precise Component Architectures", Desmond D'Souza and Ian Maung, 1999

[MDAUML] Papers on MDA and UML http://www.catalysis.org/omg

[Dso98] Desmond Dsouza and Alan Wills, "Objects, Components, and Frameworks with UML – the Catalysis Approach", online at http://www.catalysis.org/books/ocf

[Shaw9x] Mary Shaw and David Garlan, Architectures and Connectors

[Shaw96] Mary Shaw et al, "Abstractions and Implementations for Architectural Connections", 1996