



How to Apply Model Driven Architecture

Rigorous Software Development with Domain Modeling

By Peter Fontana

Version 2.2

January 5, 2005

PathMATE[™] Series

Pathfinder Solutions LLC
33 Commercial Street, Suite 2
Foxboro, MA 02035 USA

www.PathfinderMDA.com
508-543-7222

Table Of Contents

Preface	iv
Audience.....	iv
References.....	iv
1. Introduction	1
Objectives.....	2
Analysis.....	2
Design.....	2
Development Steps.....	3
2. Requirements Definition	4
Entry Criteria.....	4
Requirements Changes.....	4
Exit Criteria.....	4
3. When to Analyze a Domain	5
Realized Code.....	5
Legacy Code.....	5
When MDA Isn't Appropriate.....	5
4. Domain Separation	7
Domain Modeling Goals.....	7
Entry Criteria.....	7
The Domain Chart.....	8
Partition the System.....	9
Domain Model Validation.....	12
Exit Criteria.....	12
5. Domain Development	13
Entry Criteria.....	13
Class Modeling.....	13
Scenario Modeling.....	13
State and Service Modeling.....	13
Action Modeling.....	14
Dynamic Verification.....	14
Exit Criteria.....	15
Integration.....	15

6. Good Modeling Practice	16
Managing the Process	16
Object Blitz	16
Information Modeling	17
Domain Requirements Matrix	17
Bridge Definition	18
Activity Sequencing	18
Iterative Development	18
Summary	19
A. Analysis With UML	20
Domain Model	20
Class Model	21
Scenario Model	22
State Model	23
Action Model	24
Contact Us	24
B. Glossary	25

Preface

This paper explains how to apply the principles of Model Driven Architecture (MDA) to the development of domain models. MDA is an effective method for developing high performance, real-time, and other types of challenging software applications. It applies disciplined, object-oriented analysis and pattern-based transformational design to create systems that comply with the Object Management Group's standards.

Audience

The discussion is addressed to individuals who want to use Model Driven Architecture to develop rigorous and complete domain models. The paper is written for practitioners in several areas of responsibility, including systems analysts, design engineers, software developers, and project managers.

References

The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, and Ivar Jacobson, Addison Wesley, 1999 (ISBN 0-201-57168-4).

UML Distilled, Martin Fowler, Addison Wesley, 1997 (ISBN 0-201-32563-2).

"UML Summary Version 1.1," Object Management Group, Inc., 1997 (available at www.omg.org).

"Accelerating Embedded Systems Development with Model Driven Architecture," Carolyn Duby, Pathfinder Solutions, 2003 (available at www.PathfinderMDA.com).

Objectives

MDA improves the software development process through rigorous and complete analysis. The analysis required for domain modeling helps MDA practitioners realize the benefits of disciplined software engineering:

- Apply model based software engineering in a consistent manner, leading to higher quality models and deliverable software.
- Reduce the overall software engineering development time.
- Position the developed software to readily respond to future product requirements.
- Decrease the effort to produce and maintain software engineering documentation.
- Improve control and predictability in the software development process.

Analysis

Analysis defines *what* the system needs to do, but not how the system will do it. Consequently analysis produces models that are platform independent. Application-specific and virtually free of implementation details, analysis in MDA is rigorous and complete. The analysis is executable, and therefore verifiable through execution.

Design

Design is a strategy for mapping analysis to implementation. Because of the rigor and completeness of MDA analysis, design is virtually free of *application*-specific elements, and instead focuses exclusively on *execution*-specific requirements. The primary elements of an MDA design include a policy document defining the overall design strategy, a set of foundation base mechanisms, and a set of code templates. A transformation tool uses the design-specific code templates to generate complete, deliverable code from the application-specific models. This transformation process supports:

- Higher software quality through a uniformly applied design
- Elimination of hand-coding errors
- Increased application performance through uniform and configurable design optimizations
- Reduced integration and debugging effort through the configurable injection of object-level instrumentation and debug support

Development Steps

The overall software development process is broken into four major activities:

- *Domain Separation* - Partition the entire system at the highest level into domains of separate subject matter.
- *Domain Development* - Model each analyzed domain with class, scenario, state, and action models. Refine the models through iterative builds.
- *Design* - Develop a strategy for mapping analysis to an implementation and for assembling system components. Design development and preliminary validation is parallel to and independent from the analysis conducted during Domain Development and is often available commercially.
- *Integration* - Assemble all system components and verify that they work together using a controlled set of iterative development cycles.

This paper concentrates on the first half of this process: domain partitioning, and the domain development that results in well articulated models.

2. Requirements Definition

Embedded MDA requires attention to detail even at the level of information modeling. Software organizations that use MDA exemplify the need for detailed, consistent, and firm definition of requirements earlier in the development process than with less formal approaches. This rigor is healthy, since poorly drawn requirements translate into an unhappy project, and early detection leads to less painful cures. No approach alleviates the need for solid requirements – some just hide the problem longer.

The definition process results in requirements that are:

- Broken down to the atomic level.
- Sufficiently detailed to support information modeling.
- Sufficiently general to bound the problem, but do not specify solutions.
- Traceable to the product concept document.

Entry Criteria

An approved product concept document must be available before system-level detailed requirements can be started in earnest.

Requirements Changes

Changes in requirements after a set freeze date are likely, and any process that addresses actual software engineering must take that possibility into account. Even so, developers should strive to limit the scope of, or defer such changes. Changes of scope always have a negative impact on schedule, cost, and quality, and this impact reaches even beyond the current release. To deal productively with requirements changes:

- Firmly identify baseline requirements versions
- Document all changes, and accept no changes without a diligent impact assessment.
- For all official requirements changes, accept them in an official requirements review, including a re-release of the software schedule.
- Requirements defects are a manifestation of our humanness. Let project managers be conscientious about requirements bugs, just as developers are conscientious about software bugs.

Exit Criteria

Once the system-level requirements document is approved, it's time to stop this work – until a requirements change is imposed.

3. When to Analyze a Domain

Domain analysis is a very effective, general purpose approach for developing software. Therefore use analysis for any domain where you can readily imagine two or more objects to form a sound basis for understanding the subject matter and the problem.

Realized Code

In cases where an existing package is used, and only minor or no changes are required, conceptually allocate realized code into separate subject matters. Then construct bridges to tie the realized domains to the analyzed ones. Occasionally, it may be necessary to create an analyzed interface domain to provide the realized capabilities at a level and in a form compatible with the rest of the system.

Legacy Code

Often a block of existing code must be changed substantially to qualify it for reuse. A common inclination, especially among managers and developers intimately familiar with this code, is to expect that some economy will be realized by trying to save big pieces of it, then just rearrange things. This inclination is a false hope. Any significant restructuring of an existing system is most economically achieved by laying a sound foundation in analysis from the outset.

Think of the implementation layer or code in a system as a concrete casting. A bit of grinding here and there is fine from release to release. However any significant restructuring of this layer fundamentally weakens the overall structure of the system. The mold needs to be changed, and a new piece cast. Consider the analysis of a domain to be the mold, and the process of transforming models into code as casting. The mold is what's important – casting is relatively cheap.

Don't run up costs and place quality at risk by going from analysis or design concepts to code in ad-hoc ways. Or were you planning not to design your major changes at all...? Lesson: treat old code like old underwear – if it starts to wear out or need alterations, just chuck it.

When MDA Isn't Appropriate

There are many cases where packages or environments are available that provide very specific and effective support to develop code for certain specialized domains. An example of this is Microsoft's Visual Workbench. This is a Rapid Application Development environment supporting the quick generation of GUI-specific code. There are many examples of these environments, ranging from database and GUI realms to specialized numeric algorithm support.

Another case where a domain may not use analysis for new code is when the project's design templates will not provide a satisfactory implementation layer. This could be due to space or time performance requirements, or other issues. The first response a project should make to this condition is to attempt to adjust the design, or try a

different transformation approach for this domain. For example, not all domains in a system need to use queued asynchronous events. Some domains may not even have active objects – domain and object services could do all that is needed. In some cases, however, analysis is simply not the most effective way to solve a problem.

4. Domain Separation

Domain modeling is the most powerful of all MDA elements. It is also the least mature and least covered in terms of published papers and texts. Proper separation of subject matter supports powerful and simple constructs within domains, minimizes bridge complexity, and provides the only technically sound basis of significant reuse in the software industry.

One of the first tasks in domain modeling is to identify the boundaries of the system under construction. That can be an easy task when you are building a simple system with one application, and do not expect significant growth across releases. It can be somewhat more difficult when your system involves many applications on many processors, with significant increases in system complexity across releases. To bound a complex system, adopt a single, large system perspective, and take it to as high a level as is practical. That is, focus on one system, not necessarily on one application or device. This practice contrasts with other partitioning approaches that might bound systems based on processor or executable boundaries.

The benefits of making the system larger instead of smaller come from the ability to place all elements of the problem into a single conceptual space and exercise the relationships between them. On the constraining side, the upper limit on system size will primarily be bounded by the abilities and authority of the system architect – the system cannot extend beyond what this person can understand and control, or at least influence.

Domain Modeling Goals

Keep these goals in mind as you and your team embark on a domain modeling effort:

- Identify the boundaries of the system under construction.
- Identify the separate subject matters in the system.
- Partition the system into manageable components.
- Identify which components you will analyze, purchase, hand code or otherwise generate.
- Establish top down flow of requirements from abstract, executive domains to concrete, server domains.

Entry Criteria

Start domain modeling if an approved product concept document is available. This document must contain sufficient detail to have illuminated the major subject matter areas in the system. If more system-level requirements work will likely uncover substantial system structure not otherwise apparent, complete the bulk of that work before you undertake domain modeling.

The Domain Chart

The domain chart is a diagram showing all software components in the system separated into *domains*. These domains are directionally connected with *bridges* showing the flow of requirements from the higher level domains to subordinates that provide required lower-level services. The domain model is a domain chart with descriptions for all domains and bridges.

The domain chart for a system represents the capabilities of the target system to be delivered in a major product release cycle. Although the domain chart is complete very early in the life cycle for the release, the models within the domains mature with each step of integration. Early in the iterative builds for a release, some domains may not be complete, but at the end of the release, all abstracted domains should be mature. As the project moves from one release to the next, the domain model is modified to consider new subject matter areas as necessary.

Roles of the Domain Chart

The domain chart can seem very familiar to people used to dealing with system-wide issues and high-level design. This typically leads some people to derive unintended meaning from a domain chart. One of the significant benefits of MDA is that the definition of the method itself is rigorous. That rigor extends to the definition of the domain chart. A domain chart tells us:

- The population of domains in the system.
- How domains are related through the hierarchical flow of requirements.

The domain chart does *not* specify:

- Allocation of software to tasks, processes, processors, or networks.
- Run-time flow of data or control.

Group Portrait

In the case where a development organization is formed to support one consistent family of closely related products, a proper domain chart is a self portrait of the development organization itself. This chart reflects the relationships between all significant software efforts underway or planned in the near term. In this case, the application domain should reflect the identity of this group – what constitutes the essence of the group.

Describe Short and Long Term Views

Domain modeling is strategic in nature. It is the only area where releases beyond the current effort are considered. If possible, keep two explicit versions of the domain chart. The first is a master plan domain model that reflects the organization's best perspective on what the system will look like in the medium time frame – usually one full release cycle forward. Once the initial domain model work is done, do

a second domain model as a subset of the master to reflect the specific composition of the system in the first release. Once the organization is ready to start work on the second release, the master is revisited, updated as necessary, and a release-specific domain model is again carved out - to define the next version of the system.

Partition the System

The separation of a system into discrete subject matter domains is the most powerful technique in MDA. Proper separation of subject matter supports powerful and simple constructs within domains, minimizes the complexity of interaction between domains, and facilitates large-scale reuse. The domain is a *component* of the system, the unit of reuse.

The origins of a domain fall into two categories:

- Methodologically defensible universes of software, with purity of subject matter, broken out by careful study and conceptual analysis.
- Clumps of legacy code, off-the-shelf software and other realized code, usually hammered into a set of realized domains.

Separation of Subject Matter

Think of domains as conceptual universes – defined by the domain description or mission statement. When a domain contains a particular capability or abstraction, or if you allocate any other item defined by its subject matter to a domain, it cannot appear in any other domain.

This separation does not prevent the manifestation of elements of a server domain in a client through the presence of some sort of a *handle* or *magic cookie*. The handle must be atomic in the client, and the client must not know any more about the handle than what is published in the server bridge.

If a subset of abstractions appears in one domain but seems to be needed in another, remove these abstractions and allocate them to a to a common server. See *Delegate to Server Domains* for more explanation.

Realized domains representing yet-to-be developed code are subject to the same rules as MDA domains regarding purity of subject matter. They should not implement abstractions present in other domains, and the domain should stand consistent by itself.

When you create domains for packages or modules of existing code, you need not map all the capabilities of one package into a single domain. For instance, the MFC as delivered with Microsoft's Visual C++ can populate several realized domains: GUI Foundation, DB Engine, File System Utilities, Task and Process Controls.

What Is the Application Domain?

The application domain contains the highest level abstractions in the system. These should be the identity concepts of the system, and most should seem familiar to anyone knowledgeable about the product. The application domain also bears the burden of most of the system-level requirements.

Some practitioners may place half or more of the total object population in the application domain. For a simple system, this may be workable, but for a system of any real complexity, do not allow that to happen. Any domain should be kept to a manageable size. Experience shows over eighty objects is probably too much complexity in a single domain. The key to managing domain complexity is also the secret of a successful executive: delegation. *Delegate to Server Domains* explains how to break out pieces of a client and move them to a server.

Levels of Abstraction

The level of conceptual abstraction in the top level domains is the greatest, with more detailed and mechanical concepts located toward the bottom of the domain hierarchy. This arrangement works well when system requirements flow downward, and when a client domain delegates tasks in the same direction.

Domain Names

Maintaining a high degree of conceptual purity in a domain keeps the concepts more simple and powerful. This pursuit of purity can be a difficult, continuous struggle, and every reasonable aid should be employed to keep things in order. An effective name for a domain can do a lot to enforce the proper level of abstraction and conceptual purity. While a well-crafted domain description helps identify a domain's conceptual space during domain model development, the name carries most of the burden of communication. For instance, don't call your top-level domain *Application*. It's a waste of space.

Good application domain names, however, can be difficult to arrive at. Frequently, the first name for an application domain may come from the most prominent visible aspect of a product, or the product name itself. For instance, an application domain for an air traffic control radar system might be called *ATCRadar*, but a more complete analysis may show the highest-level domain is more appropriately termed *AircraftTrafficManagement*, with a *RadarTracking* server domain.

When you name a server domain, try not to select a name that only identifies its capabilities from the client's perspective. Instead convey the server's capabilities without restricting how they are used. This practice prevents the client's subject matter from leaking down into the server. For instance, a *VehicleSpeedControl* domain might rely on a server named *SpeedDetectorMonitoring*. We may find more clients for these operations if we name the server *AsynchronousIncidentBuffering*. That keeps the server's subject matter free from the specifics of the type of device it serves.

Evaluate the Domain Definition

The first test of a domain is to read the domain description out loud. Is it defensible? Does it meet the requirements that the system and all of the domain's clients impose on it? Does it provide usable boundaries on the constituent abstractions, including a conceptual lower bound? Can you construct a core domain mission statement without including vocabulary or concepts from other domains? (Note that it's useful to augment a core mission statement with system-level descriptions and examples, drawing on perspectives from clients and other domains for clarification if necessary.)

If a domain definition cannot be readily written, perhaps additional effort is needed to identify the major objects needed in the domain (see Object Blitz in Good Modeling Practice). You needn't identify details such as attributes or even relationships – simply populate the domain core. This conceptual exercise may provide a context that is sufficiently concrete to define a domain.

A conceptual exercise that tests the integrity of a domain transplants it to another system. For a high level domain, envision how well the integrity of the subject matter survives the replacement of all server domains with different but equivalent substitutes. For instance, how well does the application domain survive if you change the operating system, the user interface, the database, or the underlying hardware? For a server domain, envision its reuse in a different system. Can it satisfy similar requirements imposed from a different application?

Delegate to Server Domains

The relationship between a client and a server should mirror that between a supervisor and a skilled worker. The supervisor needs to know something about what the worker is doing, but does not need to know everything. As long as the worker does the job correctly and on time, the supervisor does not care how it happens.

Domain analysis may show a need to push capabilities to a lower level of abstraction than the current domain. This realization presents an opportunity to delegate these capabilities to a server domain. Issues of scope often influence the decision to create another domain. If the subordinate capabilities represent a sufficiently large effort, the overhead in creating and managing another domain is justified. The size and manageability of the current domain may also affect such a decision.

Analysis may also show that a set of abstractions seems loosely connected to the rest of the domain, but quite tightly coupled within itself. This constellation of sub-subject matter may be an essential quality of the domain, inherent in the problem space that the domain addresses. Or it may indicate an altogether separate constellation within the domain. If you see that this sub-subject matter is already allocated to another domain, then move it there. If another domain is not a likely choice as a new home, then decide whether or not a new server domain makes sense.

To decide if it is appropriate to move a sub-subject matter cluster to a new domain, determine whether:

- The cluster to be moved to the server can stand by itself as a coherent set of capabilities, not dependent on its former context.
- The set of abstractions represented by the cluster is at a lower level than the client.
- The client can easily be changed to eliminate any former dependence on the cluster.

The last case where delegation is appropriate is when two or more domains appear to have a need for a common set of abstractions. Then it is often desirable to move these objects and their relationships to a common server domain. Determine if the needs of each of the potential clients are sufficiently similar to allow a single, consistent set of requirements for the server. After you identify the common subject matter in each client, factor it out and move it to the server.

Domain Model Validation

Once the initial domain model is complete, you can take a number of steps to validate your subject matter separation. While the initial analysis may appear rather subjective, the validation steps are more objective. Their early and iterative application steadies the domain modeling process. The evaluation process includes an assessment of the system overall, and an assessment of each domain in the following areas:

- Conceptual clarity
- Level of abstraction
- Subject matter purity
- Reusability
- Analyze or realize
- Scenario testing

In addition to the above modeling-specific evaluation criteria, you can evaluate subject matter partitioning by assessing the degree of coupling and cohesion in the model. *Coupling* is the amount of undesirable interaction between domains and their constituent elements – something to be reduced. *Cohesion* is the degree to which elements within a single domain rely on each other and belong together – something to be increased. *Low coupling* between things in different domains and *high cohesion* within a domain are both good qualities in a domain model.

Exit Criteria

While the domain model must be regarded as a living document, consider it essentially complete when the authoring team decides the diagram and descriptions are complete and consistent, and after the team resolves all major review items.

5. Domain Development

System level detailed requirements are in place, and the domains you have defined are ready for development. Start with the top level, application domain, and go through these phases of development:

- Class modeling
- Scenario modeling
- State modeling
- Action modeling
- Dynamic verification
- Software system integration
- Hardware system integration

Entry Criteria

Begin development of a domain when the system requirements specification is approved, and state modeling is completed in all of its client domains.

Class Modeling

The development of each domain begins by identifying the classes that populate the domain. Descriptive attributes are added to each class, and classes are related with associations and inheritance hierarchies. Classes define the domain from the perspective of data.

Scenario Modeling

Once the data abstractions have been constructed, a strategy for inter-class communication is developed on a scenario basis. Class instance creation and deletion, events among class instances, and other significant activity between the domain's bridge interface, classes, and server domains are laid out in a sequential manner following a limited set of key scenarios.

The scenario models establish a foundation for state modeling. Carefully drawn scenario models effectively increase efficiency and quality during later phases of the development process.

State and Service Modeling

The strategy developed in the scenario modeling phase is broken out among the states of active classes, class-based services or methods, and domain-based services. Events are defined, and detailed behavior for all scenarios is defined. The actions for each state and service are summarized.

Initially, lay out positive processing steps in the state models, and review those steps with their corresponding scenarios. Add error processing and other unusual cases in a subsequent pass. Lastly, complete state transition tables to account for ignored or deferred events, and to help structure another form of error analysis – handling untimely events. Then go back and update the scenario models to reflect the state models. The scenario models will be valuable during integration.

Complete all state modeling in a domain before doing any process modeling in that domain – this will help avoid rework. Once the state modeling is complete for a domain, server domains can be started.

Action Modeling

The complete and executable specification of each state action, class-based service, and domain-based service is expressed at the level of analysis in the action language. This textual language supports a convenient, complete set of analysis processing primitives, and enforces the separation of analysis from implementation. The semantics of the action language have been standardized by the OMG as a part of the UML.

Action modeling is an analysis step, and the action language should only deal in the abstractions of its domain. Leave manipulations of other domains in those domains. Perform low-level operations, those below the level of analysis, in the Software Mechanisms domain.

Dynamic Verification

During dynamic verification, the models are executed to verify their correctness. The analysis is translated into code that runs in an instrumented executable. The patterns of communication laid down during Scenario Modeling are followed through each scenario. More particularly, the flow of control within the domain and the run-time values of analysis data elements are examined and verified. During verification, check the run-time values of these domain elements:

- Attributes
- Event parameters
- Service parameters
- Variables

Employ your development environment's static model analyzer throughout the domain modeling process to ensure correct MDA syntax and consistency. Use dynamic verification when the analysis is complete to verify the correctness of your behavior analysis. This is a technique in which the actual behavior of your analysis is *executed* – not simulated as it is commonly referred to – in your development environment.

Dynamic verification is a form of testing – likened to unit testing. As in any verification process, follow these steps:

- Use the requirements document to define the desired behavior of the system.
- Devise a test plan to define and structure the execution of scenarios.
- Run the tests.
- Evaluate the output.
- Determine the outcome of the tests and record the test results.

Use the domain requirements matrix and its background documents to define the system's expected behavior. Scenarios from the scenario models also provide a good basis for the dynamic verification. Compare your inventory of all pertinent scenarios with the domain requirements matrix to ensure sufficient test coverage.

Exit Criteria

Development of a domain is complete when the dynamic verification tests for all scenarios are passed.

Now the domain is ready for Software Integration.

Integration

Integration begins when the domains are verified and any realized domains are complete. Submissions from domain developers are assembled in a careful, stepwise process.

The software integration phase focuses on the system elements that you can run or simulate on the development platform. Assemble the components of the system and verify as much as you can without running on the target hardware. This effort is focused to illuminate all problems possible in the relative luxury of the development environment.

The hardware integration phase takes the system to the target platform, where the final, hardware-specific verification is done. The flexibility of code templates facilitates the injection of select instrumentation and execution control code into the target system, supporting object-level debugging there.

The overall integration cycle from dynamic verification through software integration and finally onto the target platform is repeated in its entirety for each iterative build in the release.

6. Good Modeling Practice

The entire process of domain analysis and development requires patience. Team members must organize many interlocking pieces of information. The advice in this section makes this process more efficient and more accurate.

Managing the Process

Given all of the above, it is clear that project leadership must consider the domain model to be an area where the most rigorous development process must be applied, including:

- Appoint a single leader for the domain model, typically the overall project technical leader, with responsibility and authority to make decisions - even in the absence of consensus.
- Identify a core subset of the project technical staff – typically no more than four people – to participate in domain modeling.
- Write down identify the goals of the effort.
- Provide a bounded period of two weeks or less to ensure focus and retain momentum.
- Apply the highest degree of professionalism, to ensure the proper balance of discussion of alternatives and cooperative forward motion.

When the domain model is complete, make updates as necessary, and only after a reasonable review process.

Object Blitz

The purpose of the object blitz at the beginning of the development process is to gain an understanding of the scope of effort in a domain. Restrict blitz activities to a single session, one to two hours per domain at the most. Brainstorm the objects that might belong in the domain – do not go into descriptions, relationships or attributes. Once a blitz identifies possible objects for a domain, examine the list to eliminate all those that are not valid objects. Quickly eliminate all objects that meet one or another of these criteria:

- No definable attributes or operations.
- Objects that belong in another domain.
- The object supports no requirements, or supports requirements not in the current release.

Once you make the first level cut, the number of remaining objects is the blitz count.

Information Modeling

The information model is the highest-level work product of a domain, and as such should tell the story of the domain. Once the domain mission has been reviewed, the natural abstractions of the domain should be captured. Do not immediately fret over mechanical issues, response time optimization, and other distracting concerns. Make your first trip around the landscape at a conceptually pure level.

Once you feel the population of the domain is reasonably complete and consistent from the first pass, quickly cobble together a couple of rough scenario outlines, just enough to exercise the new objects. Informally walk through these scenarios and review the information model in a more critical light. Review each bridge service into the domain and further refine the information model.

Don't over polish the information model. Move on when things seem reasonably complete, and the object, attribute and relationship descriptions are accurate. Allow for changes from later modeling phases – even substantial restructuring if necessary – so don't wax the bodywork yet.

As in all MDA phases, use defensible, accurate and concise names. Take time to be sure descriptions convey what is necessary for an external review, or for the busy and distracted developer who needs context to understand the abstractions. Model to satisfy the requirements of the immediate release, as structuring for future development is cheating. Sometimes it can be a good cheat, but sometimes our foresight is not as clear as we'd like.

Domain Requirements Matrix

The analysis of an individual domain is a rigorous process that is driven by requirements allocated to the domain within the context of a single iterative build. The domain requirements matrix is a table of requirement references for a single domain in a single build. Use this short document to:

- Provide a list of all system-level requirements that bear directly on this domain.
- Identify and describe all bridges into the domain.
- Record all assumptions and issues identified during the analysis phase.

Knowing the specific requirements that a domain must satisfy gives the analyst a clear direction when creating models for the domain. Think of the requirements matrix as the domain's development contract. The document should not, however, just duplicate information found in other sources. It is a set of tables tailored to the needs of the project, annotated with prose that identifies issues and assumptions.

Bridge Definition

A bridge ties one domain to another. Mechanically speaking, the externally published domain services act as the functional interface to a domain. Two factors determine the content of this interface:

- The system-level requirements that bear directly on the domain, as listed in the domain requirements matrix.
- The set of services required by the domain's clients, as outlined in the client's scenario and state models.

Define these bridge services before you begin information modeling, to place a mechanical context on the requirements. The description of each service outlines the service action at a high level, without duplicating the detail of the service's action model.

Activity Sequencing

The basic constraint for starting any domain is the detailed understanding of all requirements and constraints that bear on that domain. That means you must know the system-level requirements that bear on the domain before you begin development.

You should also define all the bridge services imposed on a domain by its clients. Typically, completing the state models for the clients serves to flesh out all bridge service needs. If greater overlap is needed between client and server domain development, you might use the results of scenario modeling to establish server bridges. The risk for overlapping client and server domain development is rework. You make the call.

Iterative Development

To manage the complexity of analysis and integration, partition development of a domain into several iterative builds. Each build should span about three months. Distribute release functionality evenly across builds, and test each build on the target hardware.

Summary

Domain modeling is one of the most difficult parts of the MDA process to manage properly. This is true for a number of reasons:

- It is the least mature of all the different elements of MDA.
- Far more than any other single aspect of MDA, domain modeling has the highest strategic impact on your organization's effectiveness, productivity, and flexibility.
- It is the most subjective area of MDA to apply - the guidelines for proper domain modeling are difficult to apply in a group setting, requiring a high degree of professionalism, and an effective leader.
- The domain model and the information models are so closely interrelated: it is difficult to start a domain model without information models, but you cannot write useful information models without a well-defined domain model.
- Since domain model is the conceptual layer that grounds all remaining analysis, changes in the model carry a potentially significant rework impact. On the good side, it is extremely difficult to work around a domain modeling problem of any real significance.

Given these difficulties, why not separate system elements based on other criteria? Because we have no generally applicable separation schemes that are easier to apply, or that yield beneficial and repeatable results. As demanding as domain modeling might be, it is still much better than the alternatives.

A. Analysis With UML

The full range of primitives in UML is quite wide. UML diagrams can be created from a number of perspectives including *Conceptual*, *Specification*, and *Implementation*. Analysis in MDA is performed from the UML *Conceptual* perspective, and is expressed through the following subset of UML.

Domain Model

System

The system is expressed through a domain chart where every software requirement of the system is assigned to a component, or domain. A Class Diagram with Packages and Dependencies represents the domain chart. Please see Figure 2.

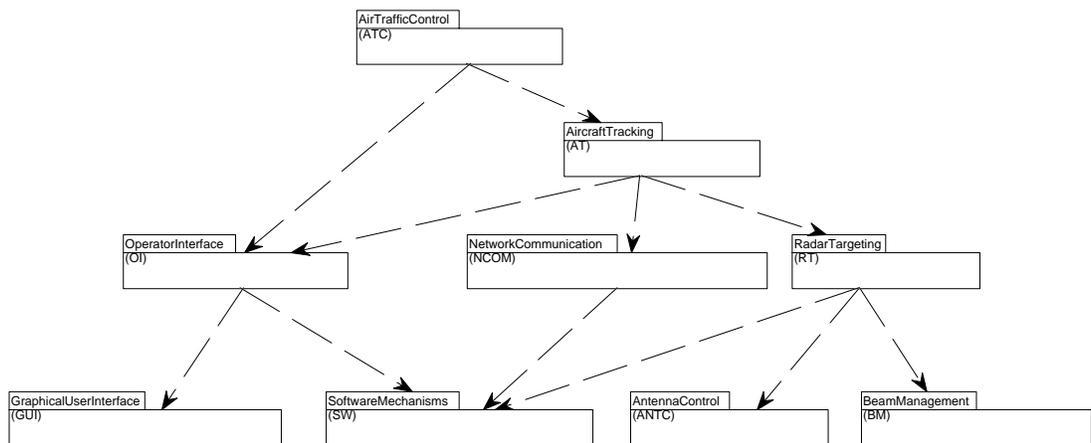


Figure 2. Domain Chart Class Diagram

Domain

The domain is abstracted using the Package, capturing its name, description, prefix, a list of domain-specific types, and a list of services. Each service has a name, and (optionally) a set of parameters. Each service parameter has a name, a data type, a description, and an I/O mode. Please note the Package symbols in Figure 2 - domains at a higher level of abstraction are shown nearer the top.

Bridge

The bridge shows the flow of requirements from more abstract to lower levels. Please note the Dependency arrows in Figure 2.

Class Model

The class model is expressed with a Class Diagram. Please refer to Figure 3.

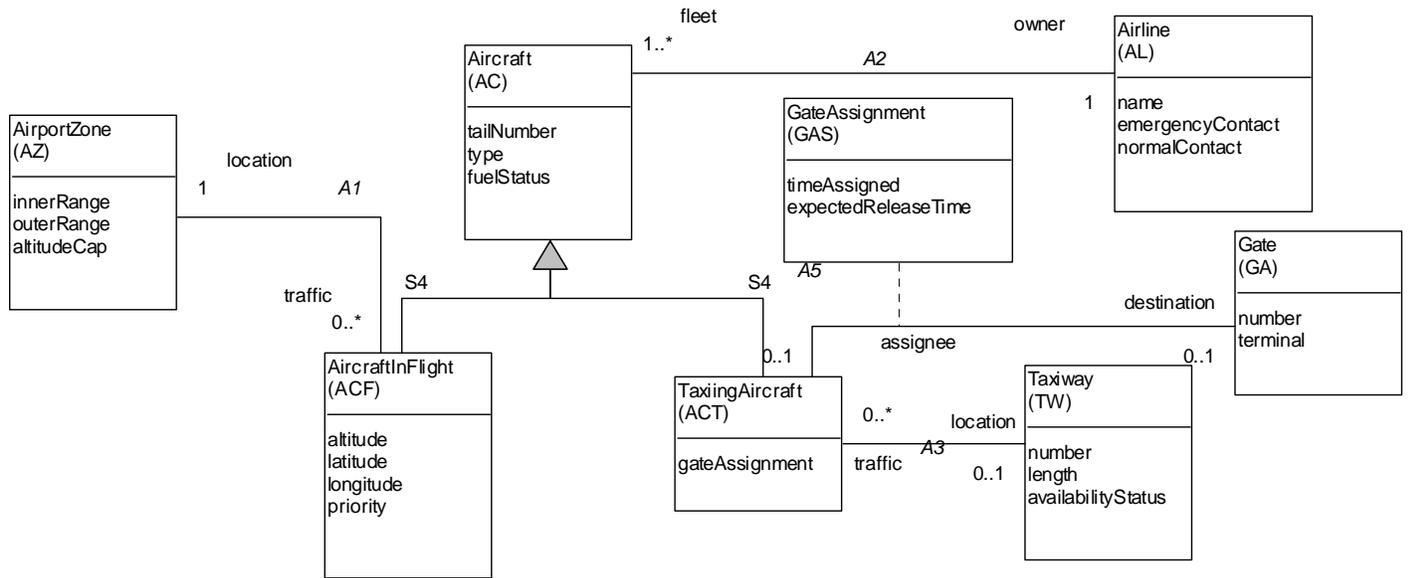


Figure 3. Class Model

Class

The primary unit of abstraction is the Class. Each class abstracts and describes the objects that inhabit the domain, capturing these objects with its name, description, prefix, a list of attributes, and a list of services. Each attribute has a name, a data type, and a description. Each service has a name, an indication of whether the service is instance-based or class-based, and (optionally) a set of parameters. Each service parameter has a name, a data type, a description, and an I/O mode. Please note the Class symbols in Figure 3.

Inheritance Relationship

A supertype Class abstracts the common attributes, relationships, and behavior of its subtype Classes. This form of relationship is shown with a set of Inheritance arrows, one for each subtype pointing to the common supertype. All arrows pointing to the same supertype have the same relationship identifier. Please note the S4 Generalization arrows that relate the supertype Aircraft to its subtypes AircraftInFlight and TaxiingAircraft.

Association

An Association arrow abstracts the binary relationship – an association between two Classes, or with one Class to itself. The Association has a

shorthand identifier (of the form "A<number>"), a description, and participant information at each end. For each participant, there is a role phrase, multiplicity (how many), and conditionality. Please note the Association lines in Figure 3.

Associative Class

Sometimes a binary relationship has its own data characteristics that are abstracted in an associative class, captured as a Class connected to an Association arrow. Please note the GateAssignment Class connected to Association R5 in Figure 3.

Scenario Model

The pattern of communication between services and state models within a domain can be captured two different ways with UML™ diagrams. The Sequence Diagram is used to create a table of the domains and objects in a scenario and sequentially list the events, Class instance creations and deletions that occur on a scenario basis. Please refer to Figure 4.

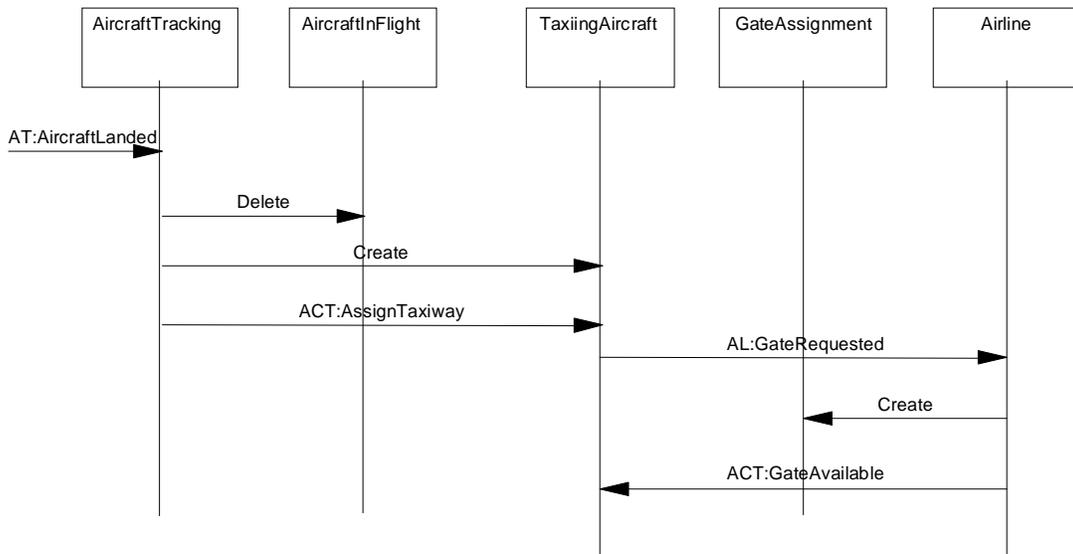


Figure 4. Sequence Diagram

Sometimes it is helpful to consider the class interactions for a scenario in the context of a topological layout of the Classes for a domain, to imply relative capabilities/intelligence and responsibilities. In this case the Collaboration Diagram is used instead of the simple tabular approach of the Sequence Diagram. These perspectives are interchangeable, and many UML tools support automatic updates between perspectives. Please refer to Figure 5.

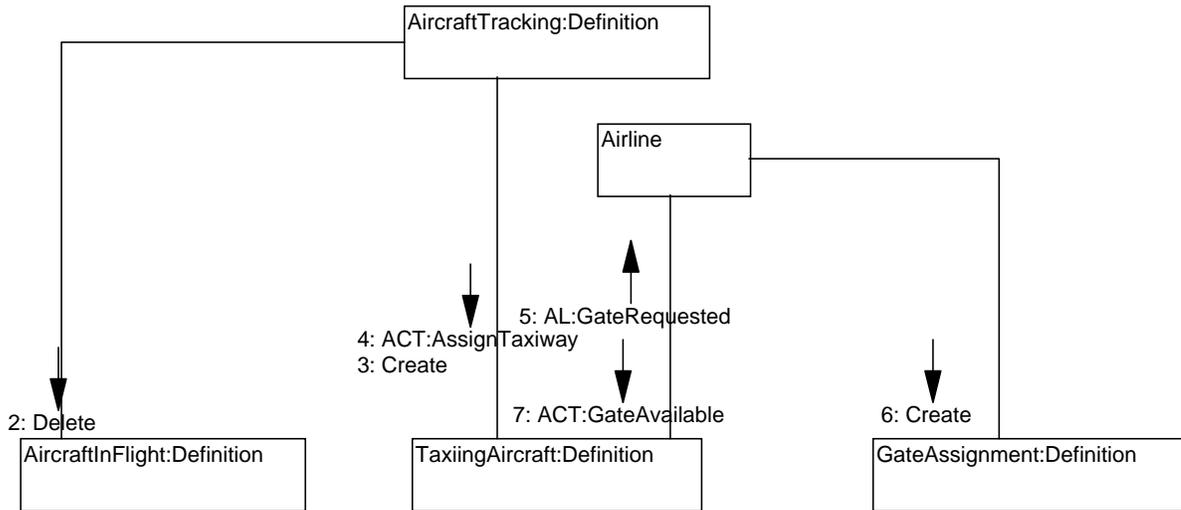


Figure 5. Collaboration Diagram

State Model

State dependent behavior of a class forms its lifecycle. The UML™ State Model is used to capture these lifecycles in terms of states, events, transitions, superstates, and substates. Please refer to Figure 6.

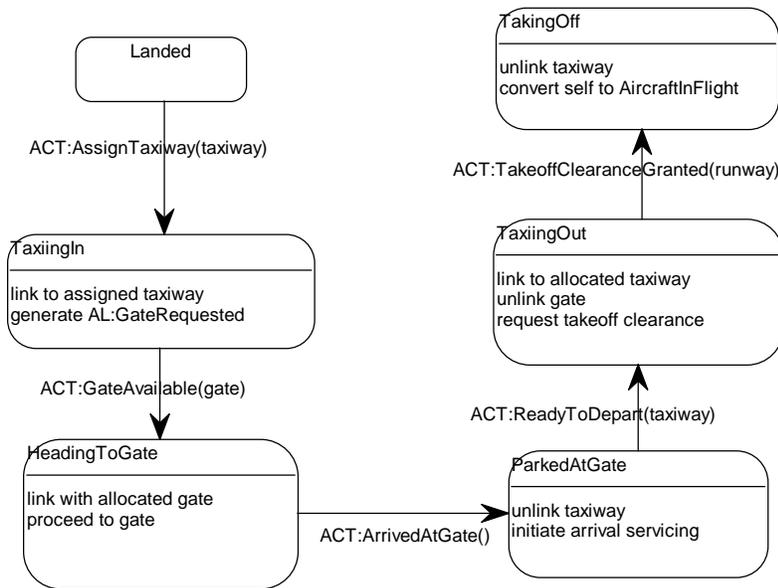


Figure 6. State Diagram

State

The State symbol is used to abstract a single stage in the lifecycle of the object. Please refer to the State symbols in Figure 6.

Superstate/Substate

To clarify diagrams where many states have a common set of transitions to other states, a superstate can be used to show the commonality among that group of states. The superstate can not have any activity associated with it. Substates refer to states within a superstate.

Event

Events are defined on the State Diagram and are associated with Transitions from one State to another. Each event has a prefix that matches the owning class, a name, and (optionally) a set of parameters. Each event parameter has a name, a data type, and a description. Please refer to the Events and their associated Transition arrows in figure 6.

Action Model

The Action Model is a detailed specification of a procedure – a state action or service action - at the level of analysis. Action Language is used to abstract analysis-level processing primitives, and enforce the separation of analysis from implementation. The Action Language statements for an action are captured in a textual container associated with the action. The Action Model below captures the detailed behavior for the *TaxiingOut* state in Figure 6 above.

```
// State action for ACT.TaxiingOut
Ref<Gate> my_gate;

// link to allocated taxiway
LINK this A3 taxiway;
// unlink gate
my_gate = Find this->A5;
UNLINK this A5 my_gate;
this.gateAssignment = NO_GATE;
// request takeoff clearance
ATC:RequestTakeoffClearance (this.tailNumber)
```

Contact Us

For more information on Model Driven Architecture, please call Pathfinder Solutions at 508-543-7222, email us at info@pathfindermda.com, or visit us at www.pathfindermda.com.

B. Glossary

Analysis	The process of developing UML Analysis Models and their Dynamic Verification, for each analyzed domain in the system. This typically is conducted largely in parallel with Design.
Analysis Models	A complete set of UML Analysis, including the Domain Model (for the entire system), and for each analyzed domain an Class Model, Scenario Models, State Models, and Action Models
Application-Specific Requirements	All requirements that define the system under development in terms of features, specific capabilities, and all aspects of system operation and behavior that are not exclusively Execution-Specific
Base Mechanisms	The set of language-specific base and utility structures that provide the operating infrastructure of the system, including event queuing and dispatch, inter-task and inter-process communication, basic analysis operation support, memory management, and general software primitives such as lists and strings.
Build	The process of compiling and linking the translated implementation code, realized code, and implementation libraries into the Deliverable System
Deliverable System	The set of executable elements that constitute the software product to be verified and delivered
Design	The process of defining and deploying a strategy for deriving an implementing from the Analysis, including Structural Architecture, Design Templates, and Base Mechanisms. This typically is conducted largely in parallel with Analysis.
Design Policies	A set of Design Patterns that define how the language-specific implementation code for the Analysis will be translated from the Analysis models. These are captured as template files in the specific notation of the UML Essentials Springboard translation engine.
Dynamic Verification	The process of exercising an analyzed domain Model in isolation to ensure that it behaves correctly. This is usually done with an external driver taking the part of the clients and servers of the domain being tested.
Execution-Specific Requirements	All requirements that define how the system under development will execute in its specific deployment environment, including task and processor topology and allocation, general capacities, performance, operating system interfaces, and application-independent capabilities
Implementation libraries	Realized system components supporting a specific compiler, language, or operating system environment
Realized elements	System components that have not been analyzed, and are typically hand-written code, generated from a specific environment (like a GUI builder or math algorithm environment), or purchased from a third party
Translation	The process of executing the Springboard translation engine to generate the complete implementation code for all Analysis Models