Managing a Transition to Model Driven Architecture

By Peter Fontana

Version 1.2

May 1, 2004

PathMATE™ Series

Pathfinder Solutions LLC
90 Oak Point
Wrentham, MA 02093 USA

www.PathfinderMDA.com
508-384-1392

Table Of Contents

1. Introduction

Model based software engineering technology can bring many benefits to an embedded software engineering organization, but there are many risks as well. How well will your culture adapt to the new approach? How can you integrate legacy and third party components along with new development? Will it even work? Should all engineers be “converted” at once, or is there a way to stagger adoption? How can the overlap of learning curves be minimized?

These are some of the issues commonly faced as embedded software engineering organizations take large steps forward in development technology, including adopting the UML. This paper identifies common risks in adopting this technology, and presents practical mitigations for these risks. The goals of the paper, then, are to:

- Survey the risks faced in adopting UML modeling technology for embedded software development.
- Illuminate practical mitigations for these risks.

2. Technology Related Risks

This section focuses on the risks associated with using modeling as the primary form of expression for high-performance software development.

Why Do Modeling?

Embedded and high-performance systems are getting more complex. Execution environments more often encompass networks of processors, running multiple threads of control. Development teams are getting larger requiring a medium that communicates architectural and technical direction. All of these technical challenges come in the face of increasing market pressure, and shorter time-to-market windows. Traditional embedded software development techniques – including the careful hand-crafting of highly optimized assembler and C code nuggets by experienced artisans - fail in the face today's challenges.

Developers of embedded and high-performance software require a higher level of abstraction, and a software development process that can be applied repeatedly and reliably by engineers with varying levels of capability. They require development technology and techniques that help them decompose their problem space and manage complexity. Software modeling effectively helps developers deal with the increasing scope of the problems they must solve.

The Unified Modeling Language (UML) is a standard form of expression for software models, based on object-oriented concepts. Many software development processes are based on UML as an underlying representation. Most currently supported or emerging modeling tooling is targeted to the UML. There is currently a large and growing base of industry experience with the UML. All of these are compelling reasons to consider using the UML for high performance and embedded software development.

Conventional Wisdom

We've all heard the straight scoop from those who already tried OO technology – and had a rough go: "Oh – so you're thinking of OO/UML/modeling. Well let me tell you about this project from hell that started out just like yours…". After you listen to a few horror stories, you can readily see why many experienced developers of embedded and high-performance software believe the following:

- ***"I can't afford to keep the models in sync with the code."*** In a code-based culture, doing a set of rigorous or detailed models just creates a large dual-edit situation. Who has time to keep up with models when there's barely time to get the code done in time?
- ***"I can't use UML because we're not implementing in an OO language."*** Many embedded and high-performance systems are deployed on platforms where there is a C compiler and not much more. Some development organizations don't have (or don't want) people with experience in C++ or other OO languages. Doesn't an OO model require an OO implementation?
- ***"UML models are nice, but an OO implementation won't work on my box."*** Common practice in object-oriented programming can readily lead to designs and implementations that cannot meet performance requirements for high performance systems. Many OOP techniques strive for flexibility and generality at the expense of time and space efficiency. I don't have control over the code generated from UML models, do I?
- ***"I can't use models because I can't start over and model the whole system."*** Virtually all projects involve large elements of legacy code, components developed by or purchased from third parties, or code generated from a specialized environment (such as a GUI builder). Starting from scratch is rarely a sound business alternative. If I use a modeling approach don't I have to model everything?

It is easy to listen to such assertions, and retreat back into a hole, fearing they are true. In fact, these negative generalizations (any many more) are based on a large body of experience – bad experience. "*So it **is** true?! You can keep that OO stuff – give me my hammer and chisel back and I'll chip ones and zeros out of stone the old fashioned way!"*

You may not have that luxury. Times have changed, and we've already accepted that we need some of the benefits that modeling can bring us. So the relevant question isn't, "Is it possible to fail with the UML?" We know the answer is yes. The relevant question is, "How can I succeed with the UML?" The answer comes in a focus on simple engineering disciplines, recognizing a key aspect of high performance software development, and in the ability to learn from the failure of others.

Make Modeling Practical

This section focuses on general ways of applying the UML, and what common issues are with each type of approach.

Ad-Hoc Architectural Diagramming

It is possible to use the UML as an ad hoc architectural diagramming notation, creating a small number of graphic aids to explain high-level aspects of your software architecture. This type of approach involves very little effort (relative to the creation of code), and the resulting "back-of-the-napkin" diagrams are generally insensitive to most code changes. In the event you do change your architecture, having just a few, high-level diagrams makes keeping them up to date a small task. However, most organizations looking to the UML for substantial and pervasive benefits cannot find meaningful gains with such a parsimonious investment. High level, ad hoc architectural diagramming generally only touches on high-level systems decomposition, leaving behind many possible benefits of UML:

- Serve as a uniform medium of expression for software elements.
- Illuminate detail and alleviate complexity.
- Apply increased communication and visibility to better leverage the efforts of each individual developer across the entire project.

Elaborative Modeling

A step beyond ad-hoc architectural diagramming is to apply the UML in a high-level "design" phase, and then continue on using these design diagrams as input to the coding phase, where developers elaborate on the design – interpreting the design diagrams as they hand code the implementation. Diagrams are created that express architectural and partitioning strategies, and outline problem-space (feature) logic at a high level. Very often there is a hazy boundary where design concepts end, and where coding concepts begin. In early phases of a project, the models can feel quite comforting, outlining solution strategies for a wide fraction of the overall problem. Relative to ad-hoc architectural diagramming, there is much greater detail in high-level "design" models. This means a greater effort is required to create and maintain them. Due to the gray zone at the code boundary, it is difficult to define precisely what should be modeled, what should be coded, and how these worlds interface or overlap.

Due to the extremely high cost of maintaining the design diagrams - trying to keep them in sync with the ever-changing code - they are often left behind by the development team. With no upkeep, or with

partial, unreliable manual diagram maintenance efforts, the models become a liability, increasingly inaccurate and misleading. Eventually the code becomes the only true view into the system.

By investing in a substantial modeling effort, but ending up no models to help during debugging, maintenance, and follow-on development, the organization pursuing elaborative modeling can easily end up with the worst of both worlds. Clearly there is a benefit – sometimes substantial – to the initial coding effort. But for a small incremental cost, this modeling investment could have been directed to a more durable end, applying a Complete Modeling technique in an effort to make models a practical investment by ensuring the durability and longevity of their models.

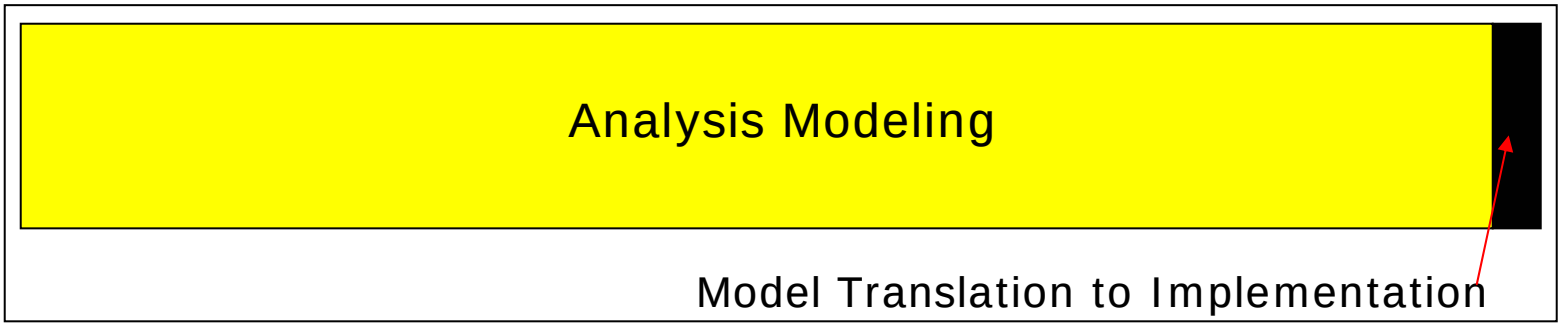
Complete Modeling

This paper is focused on the use of the UML as way to completely analyze, design, and deploy the capabilities of major system components. Typical projects starting out may model 30% to 50% of their software in an initial project, eventually reaching a blend of 80% modeled.

With such a large population of UML models, the underlying modeling process needs to ensure:

- Information captured in models is not then re-captured in a subsequent coding phase.
- Changes to models are rigorously (preferably automatically) propagated to code.
- Changes to code are either disallowed, or are rigorously (preferably automatically) propagated to models.

There are many methods for using models, but they can generally be broken into two major variants: Analysis Modeling, and Implementation Diagramming,



An Analysis Model is a solution to a problem in terms of the problem itself. The level of abstraction for the models matches the level of abstraction for the domain (component) they're addressing. Analysis models are independent of implementation, and a separate, template-based design maps these models to implementation via a transformation process. In this approach, the generated implementation code should not be changed – only models or the transformation mappings are changed. This type of process is also known as *Forward Engineering*, referring to the forward flow from analysis models to implementation code. Analysis Modeling offers some key benefits:

- Analysis Modeling maintains an appropriate level of abstraction.
- Complete Analysis Models are verifiable through execution.
- The models are free from specific implementation dependencies.
- Components that are developed via Analysis Modeling are simpler, easier to reuse, and more flexible in the face of changing requirements.

For Analysis Modeling to be practical, the models must be complete, specifying detailed behavior. Support tooling should give the project full control over the generation of implementation code. The discipline of an Analysis Modeling approach eliminates the problem of duplicate edits to models and code, by proscribing manual edits to implementation code for analyzed components.

Implementation Diagramming

"Round-Trip" Coding

This approach applies the UML to capture every aspect of the implementation code literally. The level of abstraction can sometimes approach that of the domain, but the need to capture all implementation details intrudes on the simplicity of the domain, compounding its complexity with that of the implementation domains. However implementation diagramming offers some benefits:

- It can be an easy sell to novice modelers with strong programming backgrounds due to its implementation code focus.
- Its directness and simplicity are a good match for relatively simple systems (or components) with a uniform level of abstraction.
- The support tooling is simpler and easier to learn.

For Implementation Diagramming to be practical, there needs to be a combination of process and tooling that will prevent the need for dual edits on UML diagrams and actual code. Currently there are tools that take advantage of the fact that the implementation diagrams and the implementation code are at the same level of abstraction, and they will allow edits in either UML models or code to be readily propagated to the other. This level of tool support is sometimes called *Round-Trip Engineering*.

Blending

On systems with high complexity, very often there are domains (components) that are appropriate for Analysis Modeling, others that are better suited for Implementation Diagramming, and even some that are best simply coded. The appropriate application of modeling, and the flexibility to choose approaches can be key to success.

Achieving System Performance

The creation of executable elements from UML models is not a difficult hurdle whether you are using Analysis Modeling or Implementation Diagramming. Getting the “feature logic” to run properly is important but generally this is a straightforward exercise. The difficult hurdle for high performance and embedded systems is to achieve the run-time space and time performance required.

Specific techniques for engineering high performance systems are advanced topics of considerable scope unto themselves. However there are some basic themes that can be carried back to this level, and offer the new UML adopter some simple guidance. If the previous version of your system has to face similar performance challenges, then the architecture and strategies applied here can provide a foundation to work from, in addition to the general strategies below.

Architectural Control

Very often the basic architecture of your system will dictate how it will perform. Your modeling approach and tooling must support your control over the fundamental makeup of your system. A modeling strategy that is topology independent can allow flexible repartitioning. Support for synchronous function/method calls and event driven behavior allows you to achieve the appropriate mix. Look for a modeling approach and supporting tools that afford the project the architectural control required.

Mechanical Implementation

Very often, a proprietary optimization can be critical for performance. Capturing this as a transformation pattern can support its use as an alternative as needed during code generation. Sometimes a step towards required performance is as simple as the replacement of a general-purpose mechanism with a simpler and more direct construct. In any case, having control over the mechanical details of generated implementation code can be critical for achieving proper system performance.

Freedom From Modeling

There are times when the proper form of expression of a component is simply the implementation code. General purpose UML models may complicate and obfuscate something that can be simply expressed in code. The project needs the ability to decide which components are modeled and which are coded by hand.

Vendor Reliance

Large UML tool vendors with seductive marketing messages can quickly snare a project in a proprietary world of interdependent tools and processes. Be sure to understand the technical merits of processes and tools before you choose. Seek input from independent references. If you feel overwhelmed by the sheer volume of data and cannot make an informed choice, retain guidance from an experienced source during your selection process.

As you consider model based processes and tools, there are choices you can make that decrease your vulnerability to any single vendor:

- Use standard UML – avoid variant notations.
- Select model transformation technology that gives your project complete control over the generated code, so you can go to new platforms or incorporate project-specific optimizations without vendor support or interventions.
- Use independent expertise providers.
- Choose a proven development process, based on a wide base of relevant industry experience.

3. Change-Related Risks

This section focuses on the risks related to cultural change within a development community.

Culture Shock

Change – even for the better - is often painful. Be sure you understand your culture's tolerance for change, and manage the rollout scope and speed with this tolerance in mind. Use a "wedge" approach, starting with a small effort, and then incrementally increasing the scope of your modeling efforts as each prior step achieves success.

Be sure to document the objective technical and business reasons for moving to UML. This will help all affected parties to better understand the benefits of enduring changes, and will also help keep the efforts focused.

Learning Curves

The introduction of new processes and technology often presents learning opportunities. As teams adopt UML based approaches, adjust productivity performance expectations in anticipation of overlapping learning curves including:

- The Unified Modeling Language itself
- The underlying model based software engineering process
- Your UML editing tool
- Code/report generation tooling
- Model level debugger

There may be other coincident challenges:

- Changing implementation language: C to C++
- New execution environment/RTOS
- New target hardware
- New product requirements

Human Factors: Incidence of Error

Very often proponents of new technology anticipate a wide range of benefits without accounting for the inevitable initial stumbling that accompanies change. As human beings, we are most prone to error in situations of substantial change or heightened stress. An effective modeling process incorporates work product reviews, and can improve the effectiveness of your team in the detection and correction of modeling errors.

4. Strategies for Success

We've identified key technological and cultural issues with the adoption of a model based software engineering process. Now we will identify strategies and techniques to overcome these challenges, and mitigate these risks.

Find the Right Process

As we start to consider UML and modeling, often the glistening marketing machines of various vendors lure us to focus on tools. UML model capture and execution tools are fun to play with and can have sexy demos, but this is not where the new adopting organization should start their search.

You must first understand the software development challenges your organization faces. Do you require large-scale reuse? Do you need the abstraction and power of analysis models, or will code-centric implementation diagrams suffice? Does your system need the flexibility to move to new execution environments? Do your product strategies require a scalable architecture or other forms of topology independence? How much control do you need over the architecture and implementation strategies of your product?

Select a development method that meets your specific needs. Talk to other projects using it, and see how well it works for them. Also consider if it is

- Supported by a well defined process:
  - ✓ based on proven industry experience
  - ✓ with available training, mentoring and consulting
- Flexible, accommodating non-modeled code as well as modeled components
- Leverages the capabilities of the top contributors across the entire development team
- Scalable, supporting a full range of project sizes

Once you have selected the process that meets your needs, then the tooling choice becomes much easier.

Baby Steps

An incremental introduction/growth process can help mitigate the risks of new technology and culture impact. A culture with some prior experience with similar methods or technology may not need to start with a pilot effort, or may progress faster than outlined below, but these steps indicate how a cautious organization can take it one step at a time.

Initial Investigative Consulting

The most important initial step is to understand your unique needs. Retaining an expert experienced in the successful introduction of model based processes and technology can be crucial to the focused and timely investigation of your organization's specific challenges. Spending a few days to gain an objective assessment is a great first step.

Training

Before setting up any project team with software development processes and technology that is new to them, be sure all practitioners, reviewers, and direct managers are trained. A new adopting team can easily spend months wandering, exploring and learning independently, and still not gain the skills and effectiveness that can be conveyed in one focused week of well-delivered training. Training is a must as a foundation element of all the steps listed below.

Pilot or Application Fragment

Take an initial step with technology advocates focused on a pioneering pilot project, or modeling a fragment of a larger deliverable effort. The scope and duration of this effort are constrained to manage technology risk. Allocate two or three motivated and trained developers to go through a 9 – 12 week build cycle. Secure consulting and mentoring from experienced practitioners to avoid common pitfalls, and ensure rapid progress. Complete a full integration effort – down through target hardware (if available) within the first build cycle, to experience all aspects of the development process.

Include specific review activities to answer:

- Whether the chosen development process and technology are suitable for your type of application.
- How effective the training, consulting, mentoring and tools are.
- How well the modeled system integrates and performs on the run-time execution platform

If a pilot effort is selected, it should still be tied to a real deliverable to ensure proper focus and management support is provided..

Initial Project

After a successful pilot effort, step up to a full project. Typical scope for this effort is 6 – 12 developers working on a 9-12 month release. In any case, plan incremental builds on 3-4 month intervals. Allocate different sets of feature requirements to each build, and fully integrate and test each build – all the way down to target hardware (if available). Be sure the team is properly trained and supported with consulting and mentoring.

Include specific review activities to answer:

- How well the method and technology scales up
- If there are any flexibility or performance issues not seen on the pilot, especially in the abstraction of and interface with non-modeled components
- The effectiveness of the method and tooling integration with support processes and tools, such as:
  - ✓ Configuration Management
  - ✓ Requirements Tracking
  - ✓ Defect Tracking
  - ✓ Project Management

Full Adoption

After the first 2 or 3 builds of the initial project, it should be clear if the chosen method and support technology will do what you need from it. A go/no-go decision can be made at this point, and the process can be tuned or customized to better fit the needs and culture of your organization. If the “go-point” is reached, widespread adoption can lead to strategic benefits, unattainable from limited efforts, including:

- Large scale, cross product reuse
- Rapid and efficient reallocation of resources from one project ramping down to a new project ramping up
- Project startup consulting available from your newly cultivated internal resources

Use Appropriate Technology to Retain Control

The combination of an appropriate development process and its support tooling can dramatically affect your chances of success. In addition to meeting your organization's unique development challenges, your modeling approach should ensure your project retains complete control over the following key factors.

How an Individual Component is Developed

The development process must support the decomposition of the system into high-level components. The project must retain the ability to choose how any individual component is developed: analysis modeling, import from legacy code, third party purchase, generated from a specialized environment, etc. The technique of *Domain Modeling* creates top-level components that are opaque to each other, supporting this flexibility.

Code Resulting from Models

In order to avoid the dual edit problem of models/code, all implementation code that is derived from a modeled component must be automatically translated from that component. This makes the model the "source" for that component, and the resulting implementation is treated much like a compiler-generated object file. If the code is wrong, then a model change is made to fix it.

Of course for this to work, the project must have complete control over the transformation mappings (see Target System Architecture below).

Target System Architecture

Very often in embedded and high performance systems, run-time performance requirements are as important as individual feature capabilities. Therefore software architects and designers must retain complete control over the architecture and implementation strategies of the target system. For system components (domains) that are hand coded, this of course is not an issue. However in the case of domains that are modeled – and therefore translated to implementation code – this is a critical factor.

The path from models to code is often frozen in the form of a vendor provided code generator program. If project specific architecture or implementation strategies are necessary, a model transformation approach is needed that affords complete control over generated code. A template-based approach can provide complete control to the project, supporting any architecture and even allowing implementation language changes.

Embrace the New Paradigm

Once you have taken the first few steps, and reasonably verified the suitability and effectiveness of the method and technology, step up and live in the new paradigm. Maintaining two separate development processes (model base, and legacy) is highly inefficient. Your development culture needs to penetrate the model based process and vice versa.

Very often new adopting organizations have reasonable initial success with a model based approach. But after a year or two, the actual use of modeling diminishes, and the benefits disappear. This is commonly due to a lack of long-term focus and commitment – and not an objective evaluation of method effectiveness. Treat the way your organization develops software as a business decision, and don't let good initial decisions get undone due to lack of attention and discipline.

Existing Code – Worship or Eliminate?

The flexibility to integrate modeled and non-modeled components is key to the method's effectiveness in most projects. As a modeling project goes to a second release, do you continue maintaining legacy code elements? Sometimes these legacy elements offer a stable foundation to work forward from, but sometimes you end up carrying forward diseased components, propagating legacy problems.

You chose your model based software engineering process because it is the best way to understand and deliver most new components. Likewise, it is very often less expensive to model replacement components for an existing mass of code requiring substantial repairs/extensions, than it is to continue code-hacking at the mass itself. However, it is appropriate to avoid modeling when one of the following conditions applies:

- Use available legacy code when the required capabilities are available from existing code that is complete, maintainable, properly packaged, and validated.
- Use available off-the-shelf components when they provide the required capabilities in the proper form.
- For subject matters that have dedicated and tailored development environments (like GUI, parsing).
- For domains that cannot meet their performance or other requirements through an automated mapping of models to implementation.

Many organizations deny the true cost of maintaining legacy components, and tend to sacrifice too much on the altar of existing code. The best balance here is difficult to achieve technically and culturally. Step carefully and stick to objective criteria.

Metrics

As your organization adopts a model based approach, take inventory of code-based metrics and substitute the appropriate model based metrics. Instead of SLOC, measure classes, operations, and states. Incorporate development lifecycle measurements based on your modeling process.

Estimation and Status Reporting

Estimate project scope in terms of model based elements. Use your own model based metrics, or for initial efforts apply industry-wide input from external modeling experts. Take advantage of reusing previously developed domain components and reduce future efforts with the benefit of large-scale reuse. As you track project progress, report in terms of the development lifecycle of your modeling process.

Pragmatism

It is compelling to get caught up in the excitement over a new model based process and technology. The focus of a development team can wander, and frantic movement is mistaken for progress. A modeling approach is just another tool in the development team's arsenal for delivering the product that your customer needs.

Requirements Focus

Object-oriented culture encourages the development of highly flexible, general-purpose classes. Developers can try to anticipate how their creations may be used in future releases, and build in lots of capabilities and flexibility. This culture can run counter to the goals of highly constrained projects, with limits on resources for development, testing, and execution.

In general, a development team should constrain their solution to the requirements at hand, and not try to anticipate as-yet-undefined future requirements. Certainly solution alternatives offering greater long-term flexibility should be sought, but not at the expense of development schedule or run-time performance. With experience, development teams will develop general modeling techniques that offer greater longevity and flexibility without resource impact. But initially, teams should remain focused on the requirements at hand.

The Baby and the Bath Water

In a rush to adopt new processes and tools to support model based development, organizations can "lose" existing development processes and techniques. Even with full adoption of a model based approach, there are still many supporting elements of the overall software engineering process that need to remain, and connect to the new modeling processes.

The organization should inventory all elements of their current development and project management processes, and carefully identify which are displaced by a new model based process, which need to be modified, and which should be retained. Even basic technical skills, such as scripting, test automation, configuration

management, brainstorming, peer and management review still play key roles in model-based development.

A pragmatic focus should extend to model-based tooling as well. Learn what the tools do for you, and what their limitations are. Don't reinvent what can be purchased, but also don't develop an over-reliance on a single tool environment. Know when to go to modeling, and when not to. Modeling is a powerful tool and can bring many benefits, but it is only a means to an end: successful delivery of your software product.

5. Summary

Moving forward to a model based development paradigm is a complex and challenging undertaking. But an organization can approach this like the important business decision it is:

- Develop an understanding of their own key issues and challenges.
- Gain a technical overview of the key process and tool requirements.
- Select a well-proven model based process that meets your needs, then secure training for your practitioners in this process.
- Select flexible tooling to support your chosen process.
- Learn from the experience of others through research and experienced consultants. Listen to those who have failed, and avoid their mistakes, but only follow those who have succeeded.
- Manage risk through an incremental adoption process.
- Retain control over critical aspects of their architecture and implementation.
- Keep a pragmatic focus.

References

"Model Based Software Engineering: Rigorous Software Development with Domain Modeling," Peter Fontana, (available at www.pathfindermda.com).

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

UML Distilled, Martin Fowler, Addison Wesley, 1997; ISBN 0-201-32563-2

"Introduction to OMG's Unified Modeling Language," (available at www.omg.org).