Model Driven Development with Ada

Andy Lapping
I-Logix UK Ltd.
Cornbrash Park Bumpers Way
Chippenham
Wiltshire SN14 6RA England
Tel: +44 1249 467 600
andyl@ilogix.com

ABSTRACT

System and software development has become an increasingly complex science. With so many emerging devices, processors, systems specification languages, software implementation languages, and tools for all of these, there needs to be a common denominator in the development process that brings focus back on the application. Model-Driven Development (MDD) based on the UML has emerged as the preferred approach by a growing number of systems engineers and software developers for addressing this growing complexity. The UML has proven to be the standard visual representation language capable of providing both systems and software teams with a coherent set of interchangeable artefacts that fully describe an application with rich enough specification to be able to design and implement it in Ada.

This paper examines the pros and cons of a Model Based Approach, the problems that might be encountered and some possible solutions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]:

General Terms

Design, Reliability, Standardization, Languages,

Keywords

Model Driven Development, Ada, Unified Modeling Language, UML, Process

1. INTRODUCTION

1.1 "You Start Programming … I'll Find Out What They Want"

As our applications and systems grow in size and complexity, we are forced to re-evaluate the way in which we develop. On every project, we are forced to do more with less.

Applications are many times larger (in terms of lines of code) than their predecessors. The complexity of applications is growing at a frightening rate.

And yet, these applications have to run as fast (if not faster) than before and frequently have to fit in exactly the same space as their last incarnation. When was the last time you were told "Hey – this new project – well you can have as much memory as you like and take all the time you need to make it – no rush!)

With all these problems, re-use of code has become more important than ever.

Developers are becoming increasingly aware that the old processes and techniques are just not cutting it any more – something has to change.

1.2 Don't Look Down

It has become increasingly clear that it is no longer adequate to focus solely on the code. For an application to be successful (and by this I mean it meets all its functional and quality-of-service requirements, within the time allotted for the project), developers have to keep in mind the 'bigger picture'.

Developers are raising their eyes from the code to a higher level of abstraction. Rather than focussing on the code they are shifting perspective – they are modelling. In essence they are designing before they implement!

Modelling a system provides a mechanism for good communication, and also prepares a design for future enhancement or modification with the minimum of effort.

The UML has become the industry standard for visual modelling and has attained unprecedented inter-disciplinary market penetration. However – many people now view the UML as a universal panacea, silver bullet, insert-your-own-metaphor-here. True, when implemented correctly it can provide massive benefits to the project.

The UML allows the developer to capture a visual model of the application, allowing him to view it at a higher level of abstraction – bringing together all the aspects of his project.

Requirements can be captured and expressed in UML.

The essential Design can be captured using UML notation.

BUT it is naïve to think we can forget the target language entirely – we have to implement this design - eventually we have to get to code.

BUT! Just because a developer has created a UML model – this does not mean that the code will be good code; it does not mean that all the team will create consistent code; it does not even guarantee that the code produced will actually reflect that model. This is a huge problem. When the code deviates from the model –then the

model becomes a ‘step along the way’ rather than an integral part of the process – it might as well be discarded.

And what about Testing? Can we use the UML to describe tests? Can these tests be used then to actually test the implementation? Or must we move away from the UML.

What about legacy code? This ‘un-modelled’ code still suffers from all the problems we have previously described. Do we accept this and live with it? Or can we take legacy code and re-express it into a UML model? Is it worth our time?

1.3 From Design to Implementation (From Model to Code)

When Ada (83) began its life in 1978 it was (and it could be argued – still is) a procedural language, although it did contain some of the concepts of an Object Orientated language (e.g. encapsulation via packages). The advent of Ada95 introduced more OO concepts, but Ada is still not a ‘true’ OO language. (Anyone that has tried to implement a symmetric relation will agree). So how do we map UML constructs, which are very OO, to Ada concepts? The answer is of course in the same way that we map the UML to any other implementation language. With rules.

When implementing a UML model in source code, developers need rules. Rules that map each UML construct into its coding equivalent. Rules that dictate coding styles, and coding standards.

Ada is very flexible - the way in which we might implement a UML model in Ada code is very flexible. If we were working in C++ for example, the mapping between a UML Class and code is a direct one. Even in C most people agree that it would be represented by a struct. But what about in Ada?

Almost everybody (but *not* everybody) agrees that a class in Ada 95 corresponds to a package containing a main type. In most of the cases the main type is declared private or with a private extension. Other types may also be declared within a package specification, but we refer to them as regular types, used for interfacing purposes or to define internal data structures.

But is it that obvious? – Do we use Tagged Types? Controlled Types?

Some Classes should never be tagged – if we are not going to extend them – why tag them?

If they are tagged and you derive from them, you will have to overload all their primitive binary operators (such as "+" and "-“)

So do we never tag classes? Obviously not (although some in house coding standards disagree).

So the way in which we could map each UML construct into Ada is highly flexible – even for the simplest of constructs. But of course all developers need to be using the same set of mapping rules.

This is where the concept of Code Generation comes into its own. If we have enough rules and they are complete and rich enough we can utilise auto-generation of Ada code from the concepts defined in the UML model. Imagine a code generator that can automatically process a UML design and create from it exactly the Ada code the developer had in mind.

1.4 Considerations for Auto Code Generation of Ada Code

We have already seen that the translation of a UML model into Ada code could be achieved in many different ways using just one ‘flavour’ of Ada. What if we wanted Ada83 today and Ada95 tomorrow? Add in the different Ada flavours, e.g. SPARC, the Ravenscar tasking profile etc and the possibilities become almost endless. More and more layers of ‘what if?’ – More and more work for our mapping rules.

Also consider the fact that we must cater for our developers needs – the model that they create should be rich enough to be capable of expressing what they had in mind in code. Any set of rules that we define should be capable of implementing that desire.

Could a static ‘template based’ approach work? Unlikely. Any solution would have to be dynamic in the way it processed our rule set and interpreted the UML model. Our solution should be intelligent, capable of making possibly complex decisions.

What we are talking about is a “transformation engine”. A dynamic machine that can take a UML metamodel and interpret it using dynamic, intelligent rules.

How about the rules themselves? How could they be expressed? Well the UML Metamodel is Object Orientated, so should be the rules. If our rule set could be expressed using an OO language, then we could make use of advanced OO concepts like Inheritance and Polymorphism. We could define a single rule based upon a UML Classifier that would be inherited by any Classifier e.g. a Class, Actor or Use Case.

1.5 Reverse Engineering

As we can have a set of rules that govern what UML constructs should look like in Ada code, so we could also define a set of rules that govern what Ada code should look like in a UML model. With such a rule set we have a mechanism for our transformation engine to reverse engineer our legacy Ada code and re-express it as a UML model!

A related concept is one of ‘roundtripping’. Once we have a fully defined rule set for forward generation and an intelligent transformation engine, we could take the set of rules and ‘invert’ them – or tell the transformation engine to work in reverse. Thus we could make code level changes to the generated code and have those changes automatically change the original UML model – keeping the model and the implementation in synchronisation. This concept would guarantee that the UML model always reflects the implementation code.

1.6 “Now I’ve Got To …. Keep Control”

With a fully defined rule set and an intelligent transformation engine, we have the mechanism for Model Driven Development. Now comes the question of control. Who owns the rules? Clearly if we want to maintain consistency of code between team members, access to the definition of our rule set has to be tightly controlled. It is in no-ones interest that every developer maintains his own rule set. With a common rule set, we can be certain that the code accurately reflects the model. We can be certain that any code produced by a team of developers would be written in the same way, the same coding style, the same coding standards, the same constructs.

The idea of Model Driven Development with a common rule set also provides other advantages. We can employ domain experts for their domain knowledge rather than their coding skill. Developers new to Ada can produce 'expert' code without being Ada experts. The true Ada experts would be the ones with the 'keys to the coding cabinet'.

Of course the developer must be capable of defining what he wants in enough detail that the rule-set can comply. There may be times when a developer wants something very specific in his code, and if he has no access to the rule-set, he should still be capable of getting the code he requires (unless there are very good reasons why – e.g. the company enforcing a particular strategy).

The UML provides a very rich set of concepts for defining models, and also has built in extension mechanisms that could be utilised to overcome any perceived 'shortcomings' in what the developer wants to describe.

1.7 Further Advantages

If we can guarantee that our code accurately reflects the model and is always consistent, then we can use the model for more than just documentation.

We can use the model to debug the code.

We can use the model to test the code.

1.8 Conclusion

Model Driven Development with Ada has the potential to provide incalculable benefits to any company that produces Ada code. But for these benefits to be realised, any Ada code produced must be consistent with the Model. The implementation MUST accurately reflect the design. The best way of ensuring this is to generate the code from the model. But for this to be successful, any code generation schema we use must be powerful enough to cope with the specific challenges inherent in translating a UML model into Ada code. It must be dynamic, it must be intelligent and it must be capable of 'roundtripping' any code changes back into the UML model.

Given these considerations, Model Driven Development is the future of Ada projects.