MDA Journal

April 2004

David S. Frankel
David Frankel Consulting

df@DavidFrankelConsulting.com

Introduction

This month's MDA Journal departs from previous practice by offering two articles instead of one. I would like to take this opportunity to thank Business Process Trends for continuing to host MDA Journal, and for allowing it to expand in this manner.

The article by Jorn Bettin of SoftMetaWare describes an approach called Model-Driven Software Development (MDSD), which brings together ideas from MDA, Domain-Specific Languages, Product Line Practices, Agile Development, and Open Source software. Although systematically integrating these ideas is an ongoing process, the approach has the advantage of being something that Jorn and other practitioners actively use in their consulting practice. Jorn points out that you cannot look to the OMG to provide a full-featured methodology for using model-driven techniques, because that is not the typical role of a standards body. He thus offers an introduction to MDSD.

The article by Steve Cook of Microsoft is a reply to Mike Guttman. In the February 2004 issue of MDA Journal, Mike wrote a response to Steve's first article, which had appeared in the January 2004 issue.[1] Expect some more articles in the coming months as part of this healthy and ongoing discussion about the present and future status of MDA and model-driven approaches in general. In the April issue, IBM will weigh in.

Until then…

David Frankel

[1] All MDA Journal and Business Process Trends articles from past issues are available on the Business Process Trends Web site.

MDA Journal

April 2004

Jorn Bettin

Managing Director
SoftMetaWare

jorn.bettin@softmetaware.com

Contents

Model-Driven Software Development

Executive Overview

In recent years we have seen the Object Management Group (OMG) endorse the value of model-driven approaches to software development. The evidence can be found in the marketing effort invested by the OMG in its Model-Driven Architecture® (MDA®) initiative, and the success of standards such as the Unified Modeling Language (UML®), which provides the foundation for MDA.

However, the focus of MDA is on standardization of notations and on tool interoperability. In keeping with its traditional reluctance to standardize methodologies, the OMG offers little in terms of methodological support for model-driven software development. Thus, tool vendors define their own approaches, which typically address the idiosyncrasies of specific tools, rather than providing comprehensive support for an end-to-end software development process.

Model-Driven Software Development (MDSD) is a new software development paradigm for distributed project teams involving 20+ people, with roots in software product line engineering, which is the discipline of designing and building families of applications for a specific purpose or market segment. In MDSD the following classification scheme is used as a tool for planning investments in software:

- **Strategic software assets** – the heart of your business—assets that grow into an active human- and machine-usable knowledge base about your business and processes,
- **Non-strategic software assets** – necessary infrastructure that is prone to technology churn and should be depreciated over two to three years, and
- **Software liabilities** – legacy that is a cost burden.

The relationship between MDSD and software product line engineering can be compared to the relationship between Component Based Development and Object Technology: One builds on the other, and the terminology of MDSD can be seen as an extension of the terminology for software product line engineering. The concept of *core assets* from software product lines carries through into MDSD and is directly reflected in *Industrialized Software Asset Development*, the subtitle of MDSD.

What sets MDSD apart from classical software product line engineering is the emphasis on a highly agile software development process. One of the highest priorities in MDSD is to produce working software that can be validated by end users and stakeholders as early as possible. This is consistent with the major shift towards agile software development methodologies in the industry. MDSD provides the scalability that is not inherent in popular agile methodologies such as Extreme Programming.

The Elements of Model-Driven Software Development

Model-Driven Software Development [1] provides a set of techniques [2] that enable the principles of agile software development (www.agilealliance.org) to be applied to large-scale, industrialized software development. MDSD relies on less well-known techniques from software product line engineering [3] [4] to automate the repetitive aspects of software development and to prevent architectural degradation in large systems.

A fundamental concept in software product line engineering is *domain*, which is defined as a bounded area of knowledge. Domains can relate to knowledge about vertical industries (business domains) and also to knowledge about specific software implementation technologies (technical domains). Productivity gains in building software product lines are achieved by raising the level of abstraction through the use of formalized domain-specific modeling languages that are not only understandable by domain experts, but are also machine readable. This represents a significant departure from the traditional approach of building software using general-purpose programming and modeling languages or using Computer Aided Software Engineering (CASE) tools based on proprietary languages. Domain-specific modeling languages are usually defined through a meta model, i.e., an abstract description of the language elements and the rules for composing expressions using the elements of the language. In MDSD we recognize that in some cases domain-specific notations add significant value if their development goes beyond the lowest common denominator that is harmless to share with competitors in an industry.

The translation between domain-specific modeling languages and software implementation concepts is achieved through model transformations, which either transform a domain-specific model directly into programming language constructs of a target platform [5], or use a staged approach via intermediate, less-abstract models to finally reach the level of abstraction of the target platform. Direct transformations between a domain-specific language and programming languages are often expressed in *template languages*, which have proved to be a critical element in model-driven approaches to developing software. In fact, template languages have a long track record in code generation technology, and are not an invention of the MDA era as claimed by some MDA tool vendors. Template languages enable code generation from domain-specific modeling languages. In MDSD, model transformations and code templates are first-class artifacts, as important as the models that are transformed.

Domain-specific frameworks are another important part of MDSD. A domain-specific framework is basically a template application that is only partially complete, with the remaining bits left to be specified by a software developer or by a code generator. Thus a domain-specific framework is a framework for applications or parts of applications that relate to a specific domain. Typically, domain-specific frameworks are implemented in traditional object-oriented languages, and in MDSD they are used to raise the level of abstraction of the target platform, which, in turn, keeps the complexity of model transformations within manageable bounds.

Domain-driven design of frameworks [6]—that is, explicitly representing domain-specific concepts in a set of classes and interfaces—requires deep domain knowledge. Traditionally, the success of frameworks has not been spectacular, simply because they are not necessarily easy to use by humans. Unless the code to fill the placeholders of a framework fulfills all the assumptions made by the framework developer, the resulting application will not behave as expected or may not work at all. A domain-specific framework implements commonalities that are shared across a family of applications, and that have been uncovered in a process of domain analysis. Insufficiently documented frameworks, or frameworks requiring application developers to adhere to too many complex rules, are common issues encountered in practice. In this context, using model-driven generation represents a huge leap forward by automating the use of frameworks. Framework developers can extract the rules for using the framework from a sample application and can then specify these rules in unambiguous terms in code templates. This allows framework users to model applications in domain-specific modeling languages, rather than hand crafting implementation code to complete the missing bits in the framework. The link between domain-specific modeling languages and domain-specific frameworks is provided by code generators that navigate the application models and apply appropriate code templates to generate framework completion code.

Lastly, MDSD recognizes the importance of the agile principle of "maximizing the amount of work not done," and advocates the development and use of Open Source infrastructure.

In summary, MDSD can be defined as a multi-paradigm approach that embraces:

- Domain analysis and software product line engineering
- Meta modeling and domain-specific modeling languages
- Model driven generation
- Template languages
- Domain-driven framework design
- The principles of agile software development
- The development and use of Open Source infrastructure

Software Product Line Engineering

The concept of a software component factory and tool-assisted assembly of template parts can be traced back to the earlier days in computing [7], and has since then evolved and been put into practice in the form of software product line engineering [8]. Many key features of MDSD come from this field— in particular, the differentiation between building software applications and building a product platform including relevant application development tools. In MDSD, the product platform for a software product family or software product line is developed using domain-driven design principles, and the application engineering process is automated as far as possible using model-driven generative techniques.

One way of looking at MDSD is as a set of techniques that complements software product line engineering methodologies such as FAST [3] or KobrA [9], providing concrete guidance for:

- Managing iterations
- Coordinating parallel domain engineering and application engineering
- Designing model-driven generators
- Designing domain-specific application engineering processes

MDSD is intended to be compatible with software product line engineering methodologies. Therefore, the main focus of MDSD is on the description of techniques, and not on the specification of work products which can be adopted as required from methodologies such as FAST.

The Significance of Open Source Infrastructure

In today's world of highly distributed systems, a significant proportion of a typical software system simply provides basic infrastructure services for distributed computing, security, persistence (the ability to permanently store data in databases), and so on. It no longer makes sense to develop proprietary infrastructure for the following reasons:

- Infrastructure development is expensive.
- Software infrastructure is typically far removed from what defines the competitive edge of an organization, unless the entire business is devoted to infrastructure development.
- Customers increasingly demand software that is based on open standards, and want to be able to integrate applications purchased from a range of suppliers.

Critics perceive the economics of Open Source software (www.opensource.org) to be limited to the scope of operating systems. However, other types of software are increasingly being commoditized. Open Source desktop office tools and additional examples of Open Source business software are starting to emerge [10]. The Eclipse platform [11] provides a good example of the Open Source concept being successfully applied to software development tools.

In MDSD we see model-driven generators as being a major part of the basic infrastructure for industrialized software development. In this space the number of Open Source offerings is growing, as is evident when searching the web for the terms "model driven" and "open source." A process of Darwinian selection will lead to a robust set of tools that leaves little room for expensive MDA products. In our assessment, the future belongs to Open Source model-driven generator tool kits, where not only the basic generator is Open Source, but, also, commonly required model transformations are Open Source. From the economic perspective of a commercial business, all software that does not encapsulate a unique business model or competitive advantage is best either purchased off-the-shelf or taken from a stock of public Open Source resources. Once an Open Source

resource has reached a certain level of maturity and is used successfully by a non-negligible number of organizations, proprietary alternatives quickly lose their appeal.

Today, putting off the Open Source movement as a fringe phenomenon is irresponsible from a commercial perspective, and software organizations who are governed by a culture of looking down on everything that's "not invented here" are risking the future viability of their business model. Too often a decision to build is the default option and does not have to be justified according to the same criteria that are applied to decisions to buy external software or to use Open Source software.

Of course, using Open Source software is not cost-free, but sharing the burden of infrastructure maintenance and evolution across a large base of user organizations is, possibly, as far as we can go in terms of "maximizing the work not done." Vendors of commercial software development tools can survive if they change their focus, by providing value-added components with a minimized risk of vendor lock-in on top of Open Source infrastructure.

Agile Software Development

MDSD embraces the principles of agile software development. It does not prescribe detailed team structures, and does not prescribe the micro-activities in the software development process. Instead, MDSD molds roles and teams around individuals. Although tools to automate repetitive tasks play a key role in MDSD, the tools are custom-built to the domain-specific requirements of those who need to use the tools–application development teams. One of the highest priorities in MDSD is to produce working software that can be validated by end users and stakeholders.

A number of MDSD practices are designed specifically to create a collaborative environment that de-emphasizes the importance of formal legal contracts in favor of a pragmatic approach, allowing a project's scope and priorities to change dynamically in line with changing business needs. The traditional nature of legal contracts is mostly adversarial. In practice, the approach most conducive to collaboration is a legal construct that explicitly supports collaboration, and that reduces the opportunities for litigation for both parties.

The differences from other agile approaches lie mainly in the degree of formality MDSD requires for validating software-under-construction and the concept of *scope trading* between iterations. The best practices outlined below are described in more detail in [2], including specific scalability guidelines for distributed, multi-team projects.

1. **Iterative Dual-Track Development**
   Develop the infrastructure as well as at least one application at the same time. Make sure infrastructure developers get feedback from the application developers immediately. Develop both parts incrementally and iteratively. In any particular iteration, infrastructure development is one step ahead. Introduce new releases of infrastructure only at the start of

application development iterations. In practice, to achieve sufficient agility, iterations should never be longer than four to six weeks, and it is a good idea to use a fixed duration for all iterations (timeboxing).

2. **Fixed-Budget Shopping Basket**
   Work with a fixed budget, defined by the available resource capacity for each iteration, and use timeboxed iterations of constant length to provide shippable code at least every three months. "Validate iterations" by providing a checkout procedure to confirm the items that meet expectations, and to put unsatisfactory items back on the shelf of open requirements.

3. **Scope Trading**
   Scope trading is about trading scope among future iterations. It consists of buying features for the next iteration (filling the fixed-budget shopping basket) and of deferring features to later iterations or completely removing features from the scope of the project to keep within the fixed iteration budgets. At the beginning of each iteration, use a formal scope trading workshop to agree the scope of each iteration. Ensure that not only end users but also other relevant stakeholders are present at the workshop so that all may agree on priorities. Formally define document results of the workshop, and then proceed within the timeboxed iteration in accordance with the priorities defined in the scope trading workshop to ensure that estimation errors don't affect the most important requirements and items of critical architectural significance.

4. **Validate Iterations**
   A timeboxed iteration is concluded with a formal iteration validation workshop to confirm progress and to document the parts of the software that are acceptable to users and stakeholders. Let an end user that acted as the on-site customer drive the demonstration of implemented features. Explicitly communicate to the end user and stakeholder community that new requirements can be brought up at any point. Encourage exploration of "what-if" scenarios: Stakeholders may develop a new idea while watching the demonstration, and ,similarly, the software architect may want to use the opportunity to raise issues that may have escaped the requirements elicitation process and that have been uncovered by the development team.

The Software Lifecycle: From Inception to Obsolescence

Normally the term *software lifecycle* is used in discussions about the approach and software development method used in a specific project or organization. To better understand the economics of software development, it is interesting to take a somewhat different perspective and track the life of a specific software system from its inception to its decommissioning.

Significant amounts of the software in use today are not based on scalable architectures that rigorously enforce subsystem boundaries as a tool for active dependency management. A quote from a senior software architect in a 1000+

person software organization (2003): *"Lack of dependency management is the biggest problem in our organization - this is not purely technical, it's a mindset thing."* Hence, if we track the life of a typical software system that started out from a nice, small, and innocent looking code base over several years, we find that the natural process of growth and aging has taken its toll in the form of spurious complexity. (See Figure 1.)

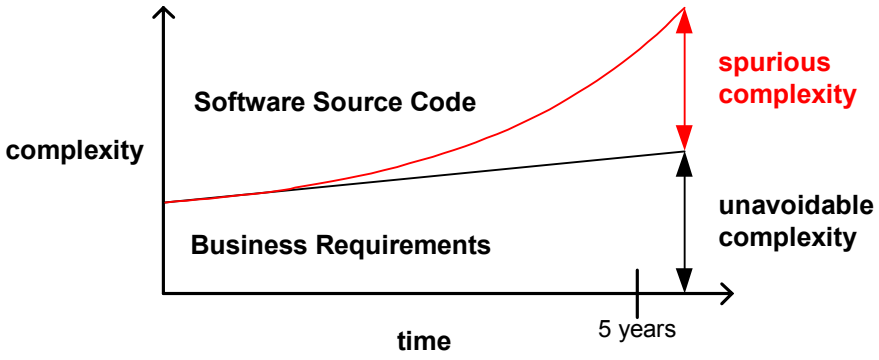

Figure 1. Spurious complexity accumulating over time.

From a certain point onwards, a sufficiently large set of new software requirements, therefore, justifies a complete or partial rewrite of the software. If software maintenance and enhancements are performed purely in the context of working towards short-term results, then the quality of the original software design degrades significantly over time. Unfortunately, treating software as a depreciating asset, where maintenance only delays obsolescence of the asset value encourages this trend [12]. This traditional accounting perspective is incompatible with the idea of incrementally building and nurturing a domain-specific application platform and reusing software assets across a number of applications within a product family. By leveraging domain-specific knowledge to refactor parts of the existing software into strategic software assets (models, components, frameworks, generators, languages, techniques), the value of the software actually increases, rather than decreases. Thus, maximizing return on investment in strategic software assets requires a long-term investment strategy, in addition to the requirement for agility in the software development process.

Of course, not all software used in an organization is worth developing into a strategic asset. The following classification scheme is useful for planning investments in software:

- Strategic software assets – the heart of your business,
- Non-strategic software assets – necessary infrastructure that is prone to technology, and
- Software liabilities – legacy that is a cost burden.

The identification of strategic software assets is closely associated with having a clear business strategy, knowing which money-making business processes are at the core of an organization, and being able to articulate the software requirements relating to these core processes. Strategic software assets are

those that define your competitive edge. Off-the-shelf business software—even very expensive enterprise systems—should only be considered strategic in terms of information about your business stored in the databases of such products, because the functionality provided is not unique and could be supplied from a number of vendors. In contrast, a unique system that reflects your specific business model may be an example of a strategic asset that is worthwhile to refine and evolve, so that it remains a strategic asset, and does not degenerate into a liability over time.

We emphasize asset development in the subtitle of MDSD, *Industrialized Software Asset Development (ISAD)*, because the model driven approach is geared towards making tacit domain knowledge explicit and capturing this strategic knowledge in a human and machine-readable format. In a nutshell, MDSD provides the tools to manage strategic software assets, with off-the-shelf products as the main source for economical provisioning of non-strategic software assets, and established Open Source infrastructure as a public asset than can be leveraged to reduce the cost of building and maintaining strategic software assets.

Achieving Organizational Agility

Recently, a lot is being written about Business Process Management, the importance of organizational agility, and the need for software systems that can evolve at the same speed at which business processes are being refined and reconfigured. The MDSD paradigm assists in the alignment of business and IT by:

- Being domain-driven, and leveraging business domain knowledge to build a software architecture that is aligned with the business process architecture.
- Assigning a very high value to domain knowledge that constitutes the competitive edge, and, at the same time, recognizing the potential to reduce software development costs by explicitly including a buy/build/Open Source decision in the development process for all software assets.
- Defining a software development process that builds on the principles for agile software development, and that scales to distributed software development in-the-large. MDSD relies on a fast iterative development cycle, and institutes a self-correcting process for balancing rights and responsibilities between business stakeholders and the software development team.

So how does Business Process Management fit into this picture? There are significant differences in opinions between people in the MDA camp and the people in the BPM camp. The goals of BPM are laudable. Achieving them, however, requires hard work, and, in our view, the fruits of model-driven approaches to software development. BPM talks about executable business process models and is based on domain-specific languages for the domain of process modeling. Regardless of whether business process models are directly executable or need to be automatically translated (compiled) into working software, the fundamentals rest on domain-specific languages. Software product line engineering gives us

the techniques and tools required to design and implement domain-specific languages; therefore, the technical implementation of BPM tools primarily consists of the creation of an appropriate domain-specific modeling language, and either the use of model-driven generation or the use of a run-time process execution engine.

It is important to point out that the languages used for BPM will not make other domain-specific languages redundant. A process-driven approach makes sense at a certain level of granularity, and it is only possible if robust software is available to provide the underlying services that need to be executed as part of a business process. Designing and building the "right" underlying services requires domain analysis, domain-driven framework design, and model-driven techniques – if it is to be done economically. Even after the service infrastructure is in place, business process changes will require software changes if the changes are of a fundamental nature that affect the characteristics of the infrastructure services.

Conclusions

Software needs to be an enabler and not an inhibitor of organizational agility. MDSD achieves this goal by classifying software into strategic assets, non-strategic assets, and liabilities, and by providing a framework for channeling investments in software development into those areas that define an organization's competitive edge. Any model-driven, asset-based approach to software needs to be accompanied by a strategy for incremental elimination of liabilities, and by appropriate investments to evolve established strategic software assets, so that they don't deteriorate into liabilities over time. In this context, it is important to value quality over quantity of strategic software assets. Unclear priorities and an emotional bias of "build" over "buy" and "Open Source" can easily lead to software project failure, and can even endanger the existence of an entire software development organization.

Further Reading

[1] Model-Driven Software Development, http://www.mdsd.info
[2] Jorn Bettin, 2004, *Model-Driven Software Development: An emerging paradigm for industrialized software asset development*, http://www.softmetaware.com/mdsd-and-isad.pdf.
[3] D. M. Weiss, C.T.R. Lai, 1999, *Software Product Line Engineering, A Family-Based Software Development Process*, Addison-Wesley
[4] Carnegie Mellon Software Engineering Institute, *Software Product Line Practice*, http://www.sei.cmu.edu/plp/plp_init.html
[5] Platform in the MDA sense. See MDA Guide, OMG document ab/2003-01-03, page 6-6
[6] Eric Evans, 2003, *Domain-Driven Design*, Addison-Wesley
[7] M.D. McIlroy, 1968, *Mass-Produced Software Components*, http://www.ericleach.com/massprod.htm
[8] Carnegie Mellon Software Engineering Institute, *Product Line Hall of Fame*, http://www.sei.cmu.edu/plp/plp_hof.html

[9] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wuest, J., Zettel, J., 2002, *Component-based Product Line Engineering with UML,* Addison-Wesley
[10] Compiere ERP & CRM, http://www.compiere.org/
[11] Eclipse project, http://www.eclipse.org/
[12] Brian Henderson-Sellers, 1996*, Object-Oriented Metrics, Measures of Complexity*, Prentice Hall

Author

Jorn Bettin is a software consultant with a special interest in techniques to optimize the productivity of software development teams and in designing large-scale component systems. He is Managing Director of SoftMetaWare (www.softmetaware.com), a consultancy that provides strategic technology management advice, with resources based in the US, New Zealand/Australia, and Europe. Prior to co-founding SoftMetaWare in 2002, he worked for over 13 years as a consultant and mentor in the IT industry in Germany, New Zealand, and Australia. He has implemented automated, model-driven development in several software organizations, has worked in methodology leadership roles in an IBM product development lab, and enjoys leading international teams dispersed across several locations.

MDA Journal

April 2004

Steve Cook
Software Architect
Enterprise Frameworks
& Tools Group
Microsoft Corporation

Model Driven Architecture and Domain Specific Modeling

In February, Michael Guttman wrote a robust response to my article "Domain-Specific Modeling and Model Driven Architecture," which appeared in the January issue of this journal. Here, I answer the major points he made.

Let's first take *"Microsoft itself offers no tools to move from UML to code or anything else"* and *"Microsoft has chosen to ignore the rest of the industry's steady march towards model-driven development."* Actually, the version of Visio that comes with Visual Studio Enterprise Architect provides an excellent implementation of UML, and supports both code generation and reverse engineering [1]. Visio has been part of Microsoft's offering for several years, and has long been one of the most popular and widely used modeling tools on the market. More than ten years ago, I used Visio myself for model-driven development when I had my own consultancy company that developed the Syntropy approach [2]. Prior to the Visio acquisition, Microsoft shipped Visual Modeler (a UML based tool derived from Rose) and worked with Rational on XDE.

Microsoft does not wish to compete with the OMG's modeling initiative. Instead of being "against interoperability," we will make it our business to interoperate with modeling products – including those that implement versions of OMG specifications–in areas where it makes sense for our customers. In particular, as well as our existing implementation of UML, we anticipate that partners will implement versions of UML on our modeling infrastructure; we expect to be able to interchange XML descriptions of models (including, but not restricted to, XMI) in future versions of our tools; and we will publish our own schemas and metamodels to encourage interoperability. We are not in any way opposed to UML, although we would like to put into perspective its value and its capabilities.

It would be wonderful if MDA gave "the ability for users to choose any set of 'best in breed' solutions, while knowing in advance that they will interoperate." Unfortunately, claims that the MDA specifications deliver "industry-wide tool and platform interoperability" are, in practice, exaggerated. The level of actual conformance to OMG modeling specifications in the market is low – mostly because it is actually very difficult to check conformance claims. There is no effective process for doing so. The UML 2 specification (and, as a primary author of it, I know this only too well) lacks the clarity to be implemented repeatedly.

There are many products in the marketplace that use UML-like models to front-end a code generation process. This is a valuable thing to do, as it saves a lot of time. (Microsoft tools do it too.) Some of them are very good, implementing diagrammatic programming languages based on UML notations. Are they MDA-compliant or not? Actually, interoperability between these tools is negligible. There are no published standards that actually specify the details of this code-generation process.

These vendors have found that the main benefit of UML is actually in its notational conventions – the fact that users can recognize and have a basic understanding

of the syntax. But anyone who has used such tools can tell you that the boxes and lines mean something different in each tool. Any benefit of UML is, therefore, not in any formalization. It seems that if you generate any kind of code from some part of UML, then you are doing MDA. (See also [3].)

At Microsoft – as at most other tool vendors – we focus on tuning up our process to be seamless and reversible, rather than MDA-compliant. We feel that a claim of MDA compliance would add no more than rhetorical value.

Early feedback from our customers to our Whitehorse offerings [4] has been extremely positive. We rarely get the complaint "but it is not UML." The overwhelming majority of users of UML recognize it as a set of notations, and neither know nor care about the UML metamodel.

Finally, on a personal note it may be worth saying that I joined Microsoft neither to go over to the dark side, nor to sell my soul, but because I perceived the company's approach to model-driven development to be realistic, focused on customer value, and based on a rich and compelling vision.

References

1. http://msdn.microsoft.com/vstudio/productinfo/overview/eaoverview.aspx
2. S. Cook & J. Daniels, "Designing Object Systems", Prentice-Hall 1994.
3. http://martinfowler.com/bliki/ModelDrivenArchitecture.html
4. http://msdn.microsoft.com/vstudio/productinfo/enterprise/default.aspx