

The Real Reason to Move to .NET

Resilient Software, Value Focus, and the .NET Advantage

Scott Came

ANZUS Technology International

October 1, 2002

Executive Summary

As Microsoft's .NET platform matures and becomes more ubiquitous, more and more Microsoft-oriented software development organizations wonder if the time is right to embrace .NET. Of all the .NET features that could be offered to justify a move to .NET, one stands out above all the others: the object-oriented language constructs and rich API of .NET *make software easier and cheaper to change over time*. In this paper, we will examine just what about .NET makes this true. We will also look at some of the important implications of inexpensive-to-change software, and we'll see how software becomes a collection of enterprise services oriented towards the maximization of return on business investment over their lifetime. Finally, we'll suggest that resilient software doesn't just happen magically by installing .NET development tools on developers' workstations; rather, it will be important for organizations to learn how to use the power of .NET appropriately and effectively.

Introduction

In late 2001, after many months of marketing fanfare and promises to the software development world, Microsoft unveiled the .NET platform. The full weight of one of the world's most powerful marketing machines went to work, demonstrating the great benefits awaiting those who would embrace the new platform. Some of the promised rewards were:

- Code that expressed business problems better by explicitly modeling the hierarchical and compositional relationships between real-world business entities
- A virtual machine architecture that executed language-neutral “intermediate language” (IL) bytecode authored in any one of several “managed” languages
- The capability to expose business functionality as “web services,” which would finally (after a decade of false starts) deliver a useful distributed component architecture
- Easier harnessing of the power of the Windows platform
- A platform that encouraged the development of component-based, service-oriented software that would naturally live longer by enabling—and being enabled by—the modern adaptive enterprise

This last notion—of adaptive, long-lived software components at work in today's wired business environment—formed the core of Microsoft's .NET marketing campaign. “An Agile Enterprise is Never More than One Degree Away with .NET,” proclaimed one series of Microsoft media blitzes. Yet despite its prominence in the .NET marketing strategy, the notion of using .NET to develop a new kind of resilient, service-oriented, value-maximizing software is perhaps the most difficult promise of .NET to visualize from a vantage point in the world .NET leaves behind.

And so the goal of this paper is to discuss the following:

- What features of the .NET platform make software easier and cheaper to change?

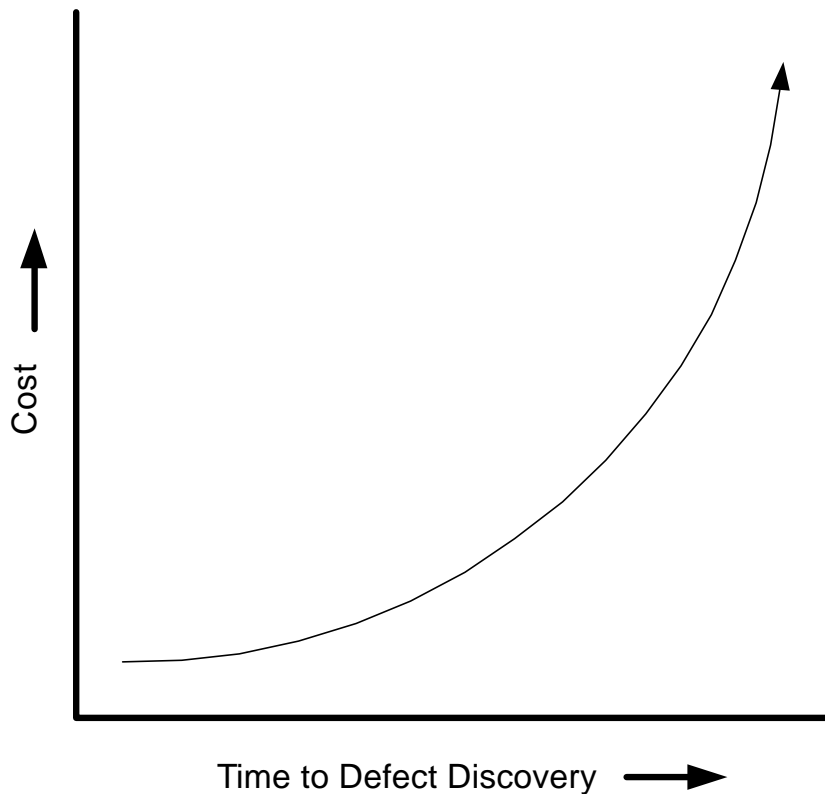
- What implications does this have for the way we approach software development and maintenance?
- .NET alone does not deliver resilient software, it *enables* it. How do information technology organizations position themselves to take full advantage of the power inherent in .NET?

The Cost of Changing Software

Barry Boehm demonstrated in the early 1980s a strong relationship between the cost of some change to a piece of software and the elapsed time in the development cycle.

Boehm showed how the later in a software project a defect is discovered, the more costly it is to fix. (By “defect,” Boehm meant missed requirements or design flaws in addition to logic errors or “bugs.”) Boehm supported this common-sense relationship with data collected from actual software projects. The relationship is diagrammed in this figure:

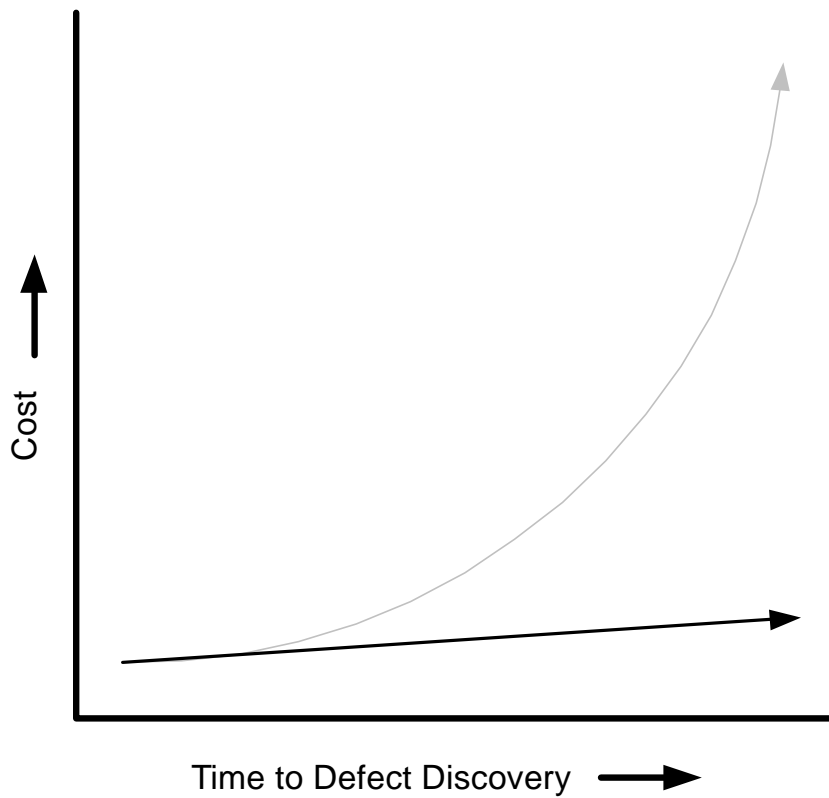
Figure 1: Software Change Cost



Object-oriented development platforms in general—and .NET in particular—have enabled the *flattening* of the cost of change curve. On projects that use OO technology effectively, experience has shown that the relationship between time of defect discovery and cost to change is not as strong or robust. That is, OO technology in the hands of

experienced, skilled developers can in many cases allow for changes to software even late in the development cycle with roughly constant costs. The flattened cost of change curve is diagrammed in this figure (with the original curve in the background for comparison):

**Figure 2: Comparison of Software Change Costs
MS Legacy Platforms versus .NET Platform**



How is this magic possible?

First of all, it is probably unfair to credit object-oriented development with all of the curve-flattening. The cost of change curve has been decreasing in slope gradually over the past 20 years, as more and more application functionality became standardized and available off-the-shelf. Perhaps the best example of this kind of advancement is the ubiquity, standardization, and low cost of relational databases. As late as the mid-1980s,

it was common for enterprise application developers to design their own data persistence mechanisms or utilize one of several non-standard proprietary solutions. Either of these resulted in more application code to maintain, and since core services like persistence tended to touch many areas of an application, a change to either the persistence logic or business logic could be very expensive indeed. As low-cost SQL databases became standardized, developers could focus more on maintaining application logic and less on details of the persistence infrastructure.

Also, OO technology like .NET *enables*, rather than guarantees, a lower cost of change. To produce results, it has to be used effectively by skilled, experienced practitioners and supported by an organizational attitude that welcomes progressive change in the development process. It is certainly true that badly-done OO development will result in brittle software that is more expensive to maintain and change than if it had been developed by talented practitioners using non-OO tools. We will have more to say on these topics later in the paper.

Nonetheless, object technology has provided another major leap forward in lowering the cost of changes throughout the development lifecycle. .NET in particular promises to continue this trend. The following features of object-oriented development platforms contribute to a flatter cost of change curve:

- **Encapsulation** or *information hiding* isolates the implementation of functionality from code that uses that functionality. All of the managed .NET languages include syntactic features like access modifiers that make implementation details inaccessible to code uninvolved in the implementation. Since encapsulation encourages the isolation of functional areas from one another, each area is immune from changes in other areas. This significantly limits the contingencies and side effects which must be considered and tested for by developers.
- **Abstraction** features like polymorphic methods and class inheritance encourage appropriate sharing of behavior across functional units, thus reducing the amount of similar or outright duplicated code that has to be changed to fix a bug or

- implement a new requirement. Polymorphism in particular offloads much of the situational logic onto the type system and the runtime itself, resulting in much easier handling of new business scenarios.
- **Design by Contract**, as represented by interfaces in the .NET languages, establishes explicit relationships between software components in terms of behavior, not implementation. That is, interfaces specify *what* a component does, rather than *how* it does it. Design by Contract helps to minimize the impact of changes in non-functional requirements (like performance, security, infrastructure choice, etc.)
 - **Enabling of design patterns** allows .NET applications to take advantage of established, accepted solutions to common design problems without “reinventing the wheel.” Use of Design Patterns in software result in code that is much easier to understand (and therefore easier to change), because it acts as a sort of abstraction layer on top of entire systems of classes. Design Patterns also offer “direct routes” to orthogonal, well-factored software components that have no duplicated code and well-defined responsibilities and collaboration between classes. Most Design Patterns cannot be implemented (and, in fact, make little sense) in a non-OO language.
 - **A full-featured, ubiquitous API** like the .NET framework provides robust, well-tested building blocks needed in modern enterprise applications. A rich API reduces dramatically the amount of code that must be written to deliver desired behavior. It also allows software teams to move faster, to experiment more in searching for optimal solutions, and to focus more on direct business value rather than infrastructure.

Implications

In many ways, the community of IT professionals and the customers we serve are so used to the sharply-rising cost of change phenomenon, it can be a challenge to grasp fully the implications of the world enabled by object-oriented platforms like .NET. Paradigm shifts that question first principles that have been held dear for a long time can be unsettling—even frightening—experiences. In this section, we will suggest some of the more important implications for software developers, the organizations that employ them, and the software projects undertaken by them.

Consequences of a High Cost of Change

The practices of the software industry to-date have resulted from a fundamental need to deal with the fact that defects are more expensive to fix later in a project. For example:

- When software is expensive to change, we are compelled to front-load the development process with analysis and design, so we make sure to capture all of the requirements and get the design right the first time. We find it valuable to require a great deal of *ceremony* on projects, including compilation of extensive documentation outlining requirements and design followed up by rigorous review and organizational “buy-off.” We organize projects into formal “phases,” each of which produce a set of artifacts that have to be judged complete before subsequent phases can begin. We expend large amounts of time and money to make sure we *get it right from the beginning*, because the cost of change relationship tells us that getting it wrong results in tremendous waste.
- When software is expensive to change, we the developers encourage (or, in many environments, force) our customers to “sign off” formally on system requirements before design and software construction begin. Because we know that missed or incorrect requirements will seriously hurt later in the project, we do what we can to compel the customer to think through the business problem as much as possible, to prevent mistakes. Often the sign-off places responsibility for

expensive changes uncovered later in the process on the customer. Typically, changes in software requirements are vetted in highly visible “change control” processes so the expense is well understood and agreed to by appropriate stakeholders. Sign-off obligations and change-control procedures, like high-ceremony analysis/design phases, incur a large amount of overhead. But the overhead cost has been justified in light of how expensive changes are when discovered late in a project.

- When software is expensive to change, we focus on building monolithic software systems or applications. We spend a great deal of time and money determining what is “in” or “out” of the scope of a particular application. Once an application is “finished,” it lives out its life doing what was in scope during its development, and eventually (in most cases) dies a slow death as it is replaced by something else. We can’t afford to build software components that are designed to be adaptive and grow with the business, because components are expensive to change as new business requirements emerge.

Consequences of Low Cost Change

As we saw earlier, the principal benefit of .NET is that it enables the creation of resilient, adaptive software, due to features in the .NET environment that make the cost of change relatively constant over time. The implications of a low cost of change are important for how we approach the creation of software:

- When software is inexpensive to change, we can afford to let application requirements and design *emerge*, rather than engaging in the high-risk, high-ceremony, high-overhead processes of the past. We can evolve the architecture and infrastructure as needed—rather than predicting them up-front—thus avoiding costly investment in risky functionality that may never be needed.
- When software is inexpensive to change, we can respond to market forces quickly, allowing us to seize new business opportunities in advance of our

competitors. That is, we can view software as a tool for enabling the agile, adaptive business enterprise needed for success in today's competitive world.

- When software is inexpensive to change, we can allow customers to change their minds and make mistakes. Instead of investing heavily in ceremony that “pins down” the customer up-front and makes it financially or politically expensive to change requirements, we redirect resources into delivering customer-prioritized features in short increments and ask the customer to “steer” development toward the goal.
- When software is inexpensive to change, we can easily modify software we've written in the past to satisfy new business needs that have arisen in the meantime. Software generates return on investment over a longer period, as it is molded to benefit the business in new ways that were unforeseen during its initial creation. We can stop thinking of software as monolithic “applications” or “systems,” and start focusing on enterprise software as an interconnected system of components that continuously evolves new ways of collaborating on behalf of the business.
- When software is inexpensive to change, the traditional “phased” approach to software projects, in which considerable resources are invested in artifacts that must be “signed off” before subsequent phases can begin, looks much less attractive. Instead, the business is more comfortable making smaller, more discrete investments, and gauging success on business value delivered rather than the completeness of artifacts per se. The resulting approach involves smaller projects with more focused goals aimed at getting value to the customer more quickly.

Clearly, the fundamental implication of a flattened cost-of-change curve is that software development organizations that know how to capitalize on it—and choose to do so—stand to cash in on a wealth of opportunity awaiting them. Adopting the requisite attitude towards value-oriented development is simple in principle, but compels a bit of organizational soul-searching. In the next two sections, we'll suggest how to get started.

Business Value: The Prime Directive in a Low Cost-of-Change World

So far, we have seen that .NET provides us with a software development platform that makes the cost of changing software relatively constant over the software's lifetime. We have also noted some implications of a world in which this is true. In a nutshell, we can afford to view software as something that grows and adapts with the business, as opposed to something that is specified in detail up-front, built as specified, and modified as little as possible over its life.

So, how should a software development organization and its customers behave in a world like this? What over-arching principle or principles should guide its work? The answer, in our opinion, is this:

Create software in such a way that the resources invested in its creation pay the highest possible total future returns to the business.

In a software development world grounded in this principle, building software becomes very much like new product development or R&D in the manufacturing world. In particular, software development in such a world takes on the following characteristics:

- Development consists of a series of small investments made by the business rather than “big bang” projects, since it is more cost-effective to let software requirements and architecture emerge rather than forcing their complete detailed specification up-front.
- Initial investments tend to satisfy core business needs or solve high-priority problems, so that the highest return is realized sooner as opposed to later.
- Subsequent investments still add new fundamental capabilities to the portfolio, but over time investments focus more on implementing higher-order collaborations between existing fundamental components. As these higher-order collaborations are built, the components participating in them will inevitably

have to change; this is where the power of a flattened cost-of-change curve is harnessed to maximum effect.

- Provides frequent communication between the business representatives and developers, as well as effective feedback for both groups to maximize synergies and minimize the waste resulting from miscommunication and misunderstanding.
- The overall investment portfolio is continuously monitored for underperforming investments as well as for potential new opportunities. Inevitably, this information will lead to the cancellation of some development initiatives, with reallocation of investment to existing or new initiatives. But because change is relatively inexpensive, we are often able to salvage work from terminated initiatives in pursuit of other initiatives.
- The continual feedback emitted by this kind of process suggests that management's chief role is to *steer* the portfolio, making minor course corrections revealed to the organization as it learns and observes its environment. Management is less about creating detailed long-range plans and enforcing compliance with them. Organizational performance is measured by business value returned for investment, not by conformance to long-range plans.

Adopting this kind of behavior pattern will come more easily to some organizations than others. Being successful at value-oriented, adaptive development requires certain important characteristics in an organization. It requires that business people and software developers (and their respective management) *trust* each other, which in some organizations can be difficult because of a history of mistrust. It requires that software development organizations view themselves as *service* organizations, which exist to serve their customers by using technology to position them for success in whatever business they're in. And it requires a culture of *agility*, where change is welcomed as a learning opportunity and plans are treated as rough guidelines rather than prescriptions for how the future will unfold. More than anything, these fundamental organizational characteristics—trust, service focus, and agility—will resonate in an organization to the extent they are consistent with the temperaments and aims of the people in the organization. How to help developers cultivate these characteristics—as well as the skills

needed to take advantage of the true power of .NET—is the subject of the remainder of the paper.

Helping People Harness the Power of .NET to Build Resilient Software

Being successful at software development with .NET, and taking advantage of the full power of the platform, requires both new skills and a somewhat novel approach to the practice of development. As we have seen, the real benefit of .NET is inherent in its ability to facilitate the creation of resilient software that is inexpensive to change. This benefit encourages organizations to take a more adaptive, value-centric approach to software development, thus reducing much of the overhead and inflexibility associated with front-loaded development approaches.

However, it is important to remember that .NET only *enables* the creation of software that is inexpensive to change. It does not guarantee it. Actually delivering resilient software requires software development practitioners to possess a certain set of skills and attitude towards their craft.

It has always been true that success at software development means hiring and retaining the best talent available. Development with .NET is no exception. The power inherent in .NET does not by any means make software development accessible to a broader, less skilled developer audience. On the contrary, effectively and safely delivering non-trivial enterprise software with .NET requires a higher, deeper level of skill than previous development platforms from Microsoft have required.

First of all, working in adaptive software development requires developers to be good communicators. They have to be comfortable discussing business requirements, issues, and tradeoffs with customers directly. They have to be comfortable with customers who change their minds, and with organizations that may change direction when the business environment dictates. They have to be good at advocating a point of view, illuminating alternative perspectives for customers and helping evaluate scenarios. They have to be good designers and analysts as well as coders. They have to inspire trust from the business customers. They have to understand that their job ultimately is to deliver

business value to the customer, which means having a service-orientation towards their work.

In short, to take full advantage of the opportunities afforded by .NET, developers should trust and be trustworthy, they should view themselves as being in service to the business, and they should be comfortable working in a dynamic, changing environment. To be sure, not every software developer can succeed in this kind of environment. Many developers working today are more comfortable working from a traditional, “frozen” requirements statement and similarly frozen design. Many developers are uneasy with the notion of growing software over time by changing code that is “finished.” And this is not to say that developers without these qualities cannot use .NET. The point is merely that some advantages of using .NET will be lost if the developers building on the platform don’t use it to build resilient software.

What’s more, developers who have great customer relationships, harbor a service orientation, and are comfortable with change, will nonetheless need some wholly new technical skills to succeed with .NET. Object-oriented development platforms like .NET are much more powerful than technologies that have preceded them. While the power in these platforms allows us to handle more complexity and tackle bigger problems, it also presents a danger in that using it incorrectly or carelessly can produce software that is worse, not better. Clearly, haphazard usage of the syntactic features and design capabilities of OO environments can easily result in software that is an entangled, impenetrable mess.

The following are some of the most important techniques and concepts for .NET developers to master. Experience with OO over the past decade or so has taught us that these things are not easy to learn or to teach—they really have to be experienced, thought about, questioned, and internalized by practicing them over time. However, simple awareness of the concepts can get an organization pointed in the right direction, by establishing some broad goals and expectations for the tacit acquisition of skills over time.

- It's not possible to master the .NET platform while hiding behind tools like code generators, wizards, GUI-builders, and the like. There is absolutely no substitute for learning and *internalizing* the platform languages and API, and the best practices of OO design and development. This is not to say there aren't situations where tools can provide a significant productivity boost. But use of tools should be a conscious decision made by a team fully aware of the tradeoffs and consequences, and should never be a substitute for actually understanding what is going on. A team where the tools are smarter than the people is a team that won't go very far!
- Producing resilient software and growing a codebase require more discipline than some developers are used to. Successful OO development teams learn to evolve software via three coordinated practices: *refactoring*, *automated unit testing*, and *continuous integration*. Refactoring is the practice of improving the design of existing code without changing its behavior. It is typically used to prepare existing code for the addition of new functionality, or to make the code easier to understand (and therefore easier to maintain.) Refactoring is impossible without automated unit testing, which consists of the development of a suite of programmatic tests that exercise the functionality of business components. As new functionality is added to the codebase, tests for the functionality are added at the same time, and before new code can be integrated, all existing tests must pass completely. Unit testing gives the development team the confidence to refactor while knowing that they are not breaking existing functionality. Finally, continuous integration mandates that the codebase always be in a state in which executable software could be released. Developers are compelled to integrate code several times a day by committing changes to a source control system, after making sure the tests pass. Refactoring and continuous integration are essential for keeping software flexible and inexpensive to change, and unit testing is essential for refactoring and continuous integration to be done safely.
- Producing resilient software requires that developers maintain a balance between two goals—orthogonality and simplicity—that are sometimes complimentary and

sometimes at odds with one another. Orthogonality requires that, in general, code should be factored so things that change together impact each other, and vice versa. There can be no duplicate code in resilient software, and covariant functional areas should be “near” each other conceptually. Simplicity requires that code be expressive but without excessive ornamentation, tricky algorithms, obscure constructs, and the like. Good OO teams constantly battle excessive architecture and extra features by focusing on keeping things as simple as possible.

- Being effective at OO development means having a solid understanding of design patterns and pattern languages. Much of the communal body of knowledge about how to co-optimize simplicity and orthogonality is documented in a few dozen design patterns. Most patterns are aimed at implementing functionality in such a way that minimizes the impact of changes on the software.

One of the lessons to be learned from the Smalltalk and Java communities is that perhaps the most effective strategy for building good OO practices is to use the services of a qualified mentor over an extended period of time. The best mentors are people who have not only internalized the skills, techniques, and practices themselves, but who have also helped others do so as well. In addition, a mentor should help focus on the wider organizational context and cultural impact of OO development, so that developers and business people alike are aware of the risks and great opportunities made possible by the technology.

Whether a mentor is retained or not, by far the best way to learn .NET in particular and OO best practices in general is by *using them on projects*. Organizations new to OO should plan for project teams to be slower than they think they’ll be, and slower than the organization would like them to be. Over time, confidence and experience will result in better estimates of resources required to deliver software, and fewer false starts and mistakes. Organizations should start small, with well-defined, focused projects that have low criticality and visibility. With experience, teams will be able to take on the more

complex, riskier problems whose solutions will really benefit from the power of .NET and OO.

Conclusion

In this paper, we've seen that a major advantage to adopting .NET as a development platform is that it enables the creation of software that is easier and cheaper to change over time. We've described the object-oriented features of .NET—like encapsulation, abstraction, design-by-contract—and the rich .NET core API, and showed how they provide the tools for creating resilient software. We examined some of the implications of a world in which software is inexpensive to change over its lifetime, first by contrasting with a world where the opposite is true, and then by viewing software development as a portfolio management exercise. We concluded that organizations should seek to optimize the total lifetime return on every dollar invested in software development, by allowing the business to steer development with small course corrections—powered by resilient software that is inexpensive to change. Finally, we noted some of the technical skills that software professionals will need to take full advantage of this power inherent in the .NET platform.

The theme of Microsoft's marketing strategy—namely, that the modern, agile, adaptive enterprise can only succeed with a renewed attitude towards software development—harbors a kernel of truth. Survival in the modern economy requires that software development organizations be able to respond to inevitable change quickly and effectively. The key to this ability is to recognize the power in platforms like .NET to create software that is inexpensive to change. However, it is not enough to buy the tools and declare a commitment to the platform. As software developers, we must adopt an attitude that centers on servicing our customers, measuring our success by the business value that we deliver, and ultimately viewing inevitable change as an opportunity rather than a risk.