Journey of Enlightenment: The Evolution of Development at Microsoft

Eric Brechner, Ph.D.
Director, Microsoft Development Excellence
One Microsoft Way
Redmond, WA 98052
ericbrec@microsoft.com

ABSTRACT
Like many software companies, Microsoft has been doing distributed application development for many years. However, recent changes in the market have altered the rules, both in terms of customer expectations and programming models for ubiquitous interconnected smart devices. These changes have provoked two dramatic shifts in the way we develop software. The first is the creation and use of the .NET Framework as a simple, secure, and robust platform for device-independent software development, data manipulation, and communications. The second is an agile yet highly disciplined approach to designing, testing, implementing, and verifying our software which presumes all bugs are unacceptable and must be found and fixed early before they impact internal groups, external partners, and eventually our customers. This paper discusses the nature and impact of these two dramatic shifts to the development practices at Microsoft.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *life cycle; productivity; programming teams; software process models; software quality assurance.*

General Terms
Management, documentation, design, reliability, security, human factors, standardization, languages, verification.

Keywords
Agile, design, scrum, test driven development, lean, refactoring, Microsoft, .NET, quality, XML, SOAP, WSDL, versioning, interfaces, coordination, collaboration, contracts, security, reliability.

1. INTRODUCTION
Like many software companies, Microsoft has been doing distributed application development for years. However, changes in the market over the past five or six years have altered the rules, both in terms of customer expectations and programming models. Key market changes include smarter devices; wider variation in devices; ubiquitous connectivity; proliferation of malicious computer viruses, worms, and spyware; and the broad reliance of businesses and consumers on software for their livelihoods and well-being.

These changes have provoked two dramatic shifts in the way we develop software. The first is the creation and use of the .NET Framework to provide a simple, secure, and robust platform for device-independent software development, data manipulation, and communications that supports a wide variety of programming languages. The second is an agile yet highly disciplined approach to designing, testing, implementing, and verifying our software which presumes no bugs are acceptable so all must be found and fixed at the earliest possible time before code gets checked in and impacts internal groups, external partners, and eventually our customers.

These two changes to how we develop software can be thought of in terms of tools and process. Naturally, they meet together with the people who use the tools and follow the process. The people aspects of the problem are the most critical for successful projects, and the most interesting in their impact on the way the tools and processes are used in practice.

In addition to discussing the .NET Framework and changing Microsoft development practices in more detail, this paper will focus on the subtle parallels between tools and processes that were driven by human factors to make distributed development of distributed systems more practical.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

2. THE EMERGENCE OF .NET
The .NET Framework was driven by changes in the market, so an overview of the design decisions is bound to sound like marketing. This I regret and will try to stick to the technical side of decisions.

It is a common misconception that .NET is purely focused on device-independent development to compete with a language like Java. To support a wide variety of smart devices, you do need a device-independent way of controlling them and describing a user interface which adapts its presentation and interaction to the device. But this is not nearly enough to support ubiquitous interconnected smart devices. You still need to address ubiquity and interconnectivity.

2.1 Interconnectivity

Interconnectivity in today's world presents significant challenges:

Data manipulation: You need a standard means of describing and serializing data. XML was a clear choice for .NET. It is a well-established standard with broad Internet adoption, can represent complex data types, and is easily serialized in a processor-independent manner.

Communications: You need standard protocols for messaging. SOAP was another clear choice for .NET. It is based on XML with enough flexibility and adaptability to implement simple request/reply protocols like HTTP as well as arbitrary exchange patterns found in enterprise integration scenarios.

Security: You need the devices, their communications, and their data to be secure. Once those devices are interconnected, they become vulnerable to attack and their private data may be compromised. The .NET Framework protects applications and data from the most common threats, by managing memory to prevent buffer overflows, using verifiable type safety to prevent data corruption, driving least privilege throughout the system to shield assets, and integrating encryption support for easier protection of data integrity and privacy.

Reliability: You need the devices and the services they depend on to be robust and reliable so the devices stay connected and data is preserved. The .NET Framework uses managed code and exceptions throughout to trap and prevent the most common failures while providing a standard lossless mechanism to capture error conditions. Exceptions are only as effective as their proper use. But choosing a uniform mechanism prevents the disasters of conflicting error handling, and exceptions make it easy to pass information far up the stack without data loss or special handling.

2.2 Ubiquity

Ubiquity requires adoption. Adoption requires simplicity of transition from old models, simplicity of development and deployment, and simplicity and consistency of interface. Modern methods make simplicity and consistency of user interface tractable, though still difficult.

Simple transition from old models is a far more serious problem. Naturally, the user interface, communication, and data models require a shared library of objects and interfaces. However, making access to those services simple, reliable, and secure requires language support.

But which language? C++ is notoriously insecure. Adding features to Java has proven challenging for non-technical reasons. Visual Basic has never quite gotten the respect it arguably deserves. Even if one of those languages was a good choice, you'd still have adoption problems for all the others.

So the .NET Framework chose all the above. By creating a common language runtime (CLR), the .NET Framework can support the constructs of almost any language (over 20 languages are available). A new language, C#, was designed to construct the library of objects and interfaces in the most simple, secure, and robust way possible. But any CLR-supported language can leverage those libraries without crossing an insecure boundary.

In addition, the CLR adds attributing to all the languages, which simplifies messaging and serialization, and the Visual Studio .NET environment provides a common build, deployment, and debugging platform, which vastly simplifies development.

3. AGILITY WITH DISCIPLINE

Having a simple, secure, and robust framework for writing interconnected applications allows developers to quickly construct oodles of powerful new code. That's a big problem, especially for large development teams, like the ones at Microsoft. Customers can easily get lost in a sea of features, assuming they first don't drown in unchecked defects. Some controls are necessary.

3.1 Old school Microsoft

When Microsoft was very small, developers did everything. They switched from one project to the next sometimes on the same day. As the company grew larger, devs formed into focused small teams.

Two new roles were created: program managers (PMs) who coordinate with marketing, user research, and packaging to specify what features are in the product and when certain milestones need to be hit; and testers who test the software, verify it matches the specification, and greenlight release.

A mix of perhaps 30–60 PMs, devs, and testers form an individual product unit responsible for a product offering or significant component. Today, Microsoft has over 400 product groups, each independently managed with significant autonomy.

Traditionally, each product group determined and followed its own practices and processes. In the past, most have followed a modified waterfall model with daily or weekly integrated builds and six to twelve week design-implement-stabilize cycles. Use of unit tests, code reviews, and design inspections varied widely across groups.

However, over the past seven years our teams have become increasingly randomized. Products have matured, so new feature requirements aren't clear and change rapidly. Expectations of secure and reliable products have risen along with the threats of viruses, worms, and spyware, so our teams are frequently correcting past errors or redesigning old components.

You could say we are victims of our own success or victims of our own carelessness or lack of foresight. Regardless, our product groups are struggling to take back control in a sea of constant change. This has led us to a number of adjustments in how we work, many grounded in agile methods.

3.2 Agile influence

The agile software movement has gained a great deal of momentum, and for good reason. You focus on the customer, respond to change, and deliver value instead of documentation, plans, and wrangling. Developers can target their rapid development on satisfying customer needs with constant feedback and an uninterrupted value stream.

Unfortunately, when you are serving a wide variety of customers and working with a large number of partners, it's tough to get them all in a room once a year, let alone once a week or month.

This problem is inherent with integrated services. While a single service or application may have a manageable number of primary customers and partners, integrated services provide higher value

at the cost of a wider variety of customers and a larger number of partners. This is especially true for platform development.

Integrated services can be built up bit by bit over time, to provide better focus. But when large numbers of teams are working simultaneously on an integrated offering, everything gets dysfunctional quickly. Microsoft's approach has been to add just enough process and documentation to keep trust boundaries in sync and the value stream flowing. To be clear, this is still a work in progress, but many teams are seeing positive results.

3.3 Just enough process

With over one hundred product groups working on Windows alone, your customers and partners are both outside and inside the company. They are both down the hall and across the globe. This can make daily builds anywhere from exciting to impossible.

The first step we've taken has been to isolate components as much as possible. This doesn't mean the system can function without all the components; it just means that each component can be built independently—that is, as long as you don't change the interface between components. I'll get back to that central problem in a moment.

The second step we've taken is to institute standard build systems for our major product lines and a significant quality bar for submitting code into the build. Before we had these standards, anyone could and would frequently break the build. Sometimes builds would take weeks, crippling continuous integration and the value stream.

Now, product lines have minimum standards before check-in for unit test coverage, code and security reviews, advanced code analysis (targeting security holes and other defects), and passage of all automated tests. These gates provide just enough to ensure a functional and clean build, without being too burdensome.

The idea is to enable teams to have the latest builds available quickly so that they can validate and adjust direction frequently with customers and partners. Many teams are employing scrum [5], lean development [4], test-driven development (TDD) [3], and refactoring [2] to make themselves more responsive to change, drive better design, minimize work in progress, and provide the required level of code quality.

3.4 Just enough documentation

Of course, integrated systems are only as good as their points of integration. This is as true of an integrated operating system as it is of integrated distributed services across devices. In both cases, the system can't stop for any one component; it must always be running.

Because an interface change can be so disruptive, extra controls must be put in place to minimize the impact:

- The interface must be designed well upfront.
- The interface requirements and interaction must be well understood and documented with buyoff from all key customers and partners.
- The interface must be carefully version-controlled so that changes do not adversely impact currently running components.

With such a diverse and broad population of developers, just about every design methodology is in use somewhere at Microsoft. While strong upfront design is essential for coordination across large projects, many projects have failed by doing too much upfront design instead of just enough. Design continues to be a journey of discovery, not a destination.

Versioning of interfaces is just as important as versioning of source code and binaries. Versions must be well-managed and maintained to keep the entire enterprise up and running without interruption or failure. This means integrating versioning into the depths of our serialization and messaging engines. The problem is a continuing source of interest and research for the company [2].

3.5 More than a handshake

While designing and versioning interfaces present many difficulties, the broadest challenge is documenting the requirements and interactions of interfaces with buyoff from all key customers and partners. What's particularly interesting about this problem is that it happens on two levels: between groups of people and between groups of services.

Between groups of services, the .NET Framework is relying on the Web Services Description Language (WSDL, http://www.w3.org/TR/wsdl). This XML description is made available by service providers through a simple protocol. The WSDL specifies expectations around data and communication, and forms a rudimentary contract between the interacting services.

Like a contract between people, the WSDL contract lies outside of the interactions themselves. It merely guides them when needed or desired. Intermediaries can be inserted anywhere along the message route to validate and enforce the contract at the appropriate level. This flexibility allows the server and client to use only as much control as they need over time without forcing the validation code to be shared, versioned, or originate from one source.

Without a contract in place, interfaces could change or produce unexpected behaviors, causing illegitimate results or catastrophic failures. With a contract in place, each side can know what to expect, reject inappropriate requests, and when properly constructed can robustly recover to a stable state if the contract is not kept.

As it happens, this is precisely the behavior you want between groups of people who rely on each other. At Microsoft, informal contracts exist between many teams with dependencies. Again, the "just enough" rule applies.

The contract states the expectations and requirements of each team (delivery dates, languages, platforms); the ownership and responsibilities of each team (source code, testing, bug databases, build, setup, drop points, localization, service operations); and the fallback positions for both sides if the contract is broken. Teams also often share automated build verification tests to ensure interfaces continue to match expectations.

The fallback position is the most interesting. It is what makes the interaction between groups possible, be they services or people. Without a fallback position, the system is not stable and will likely fail. With a fallback position, the system keeps running and resolves issues.

For people, the fallback position simply diffuses tension because everyone knows there is a way out so they can completely focus

on the problem at hand. That doesn't always make it easy or pretty, but it does remove uncertainty and allows participants to compare the current situation and solutions against the fallback. The result is clearer thinking, better prioritization, and less apprehension.

4. CONCLUSION

Agility and change don't come easily for large companies of any kind. In the past, Microsoft has leveraged its corporate structure of hundreds of small independent product units and open network communications to quickly adapt to changes in the marketplace.

Now the market for ubiquitous interconnected devices and increased expectations of "always on, always working" have driven Microsoft to make dramatic shifts in its platform, development environment, and development process. We created the .NET Framework, standardized build systems and quality gates before check-in, and are devising schemes to manage our interfaces between services and the teams that build them.

All these efforts together foster an environment in which each small product unit can adopt agile development strategies which minimize work in progress and deliver continuous customer value throughout the development cycle. A key aspect of this strategy is "just enough" documentation and process to enable continuous integration across groups.

Even with the new framework and standards in place, the large collection of product units still have wide latitude to try innovative ideas in both their products and process. Some groups embrace change more aggressively than others, and some groups fail to adapt. Some groups are making great progress, and some groups are dealing with pain and dysfunction. But autonomy brings out originality, and many radical ideas from the past are now common practice at the company. Just enough structure coupled with a flexible approach keeps Microsoft agile and customer-focused.

5. ACKNOWLEDGEMENTS

The author would like to thank Wolfgang Emmerich for inviting this paper, Don Box for enlightening me about our latest efforts in Web services, Linda Caputo for her editing assistance, and Clemens Szyperski for once again harassing me to do it.

6. REFERENCES

[1] Fowler, Martin, et al. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Pub Co, 1999.

[2] Meijer, Erik, and Szyperski, Clemens. *Overcoming Independent Extensibility Challenges.* Communications of the ACM. Vol. 45, No. 10, pp. 41–44, October 2002.

[3] Newkirk, James, and Vorontsov, Alexei. *Test-driven development in Microsoft .NET*. Microsoft Press, 2004.

[4] Poppendieck, Mary, and Poppendieck, Tom. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley Professional, 2003.

[5] Schwaber, Ken. *Agile Project Management with Scrum.* Microsoft Press, 2004.