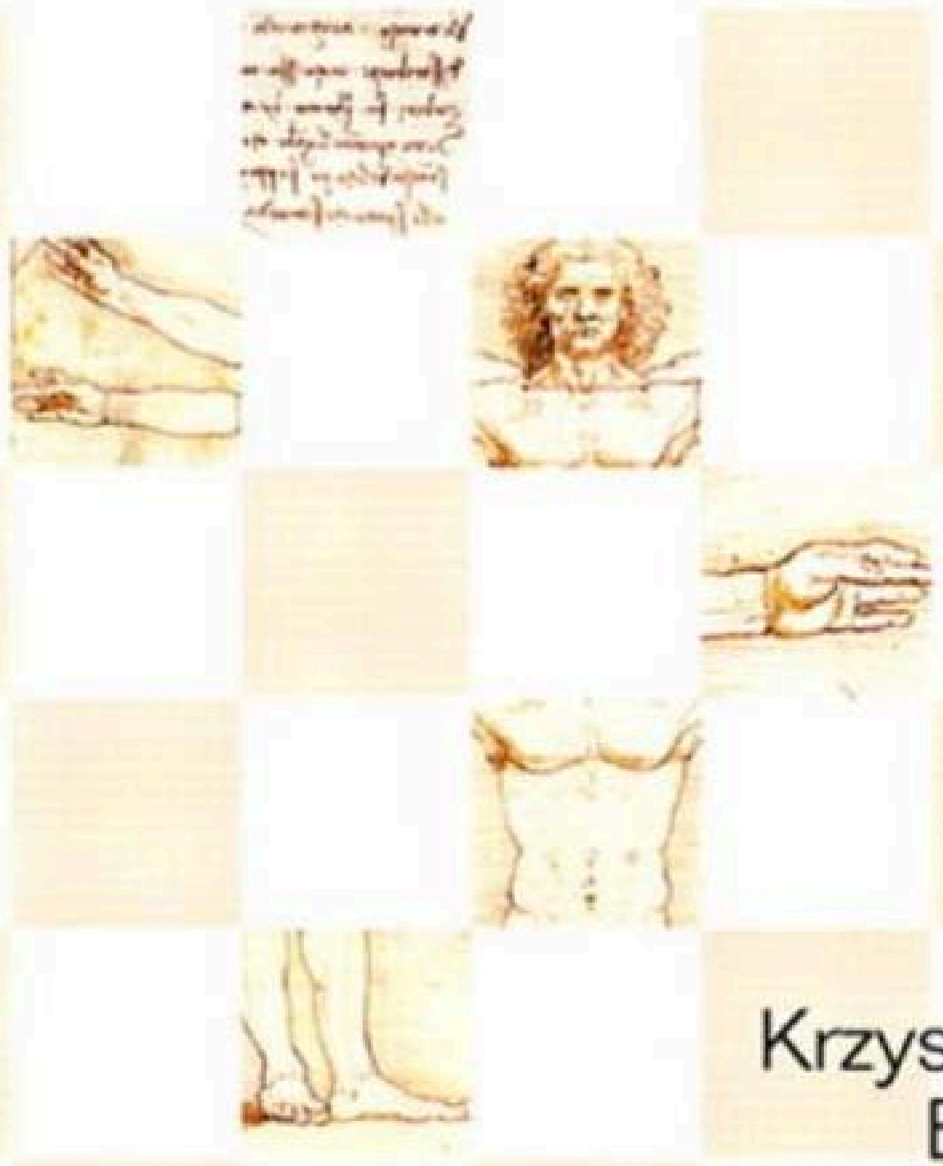
Foreword by Anders Hejlsberg
Distinguished Engineer, Microsoft Corporation

Framework Design Guidelines

Conventions, Idioms, and Patterns for Reusable .NET Libraries

Microsoft .net Development Series



Krzysztof Cwalina
Brad Abrams

1 Introduction

IF YOU COULD STAND over the shoulder of every developer who is using your framework to write code and explain how it is supposed to be used, guidelines would not be necessary. These guidelines give you, as the framework author, a palette of tools that allow you to form a common language between framework authors and the developers who will use the frameworks. For example, exposing an operation as a property instead of exposing it as a method conveys vital information about how that operation is to be used.

In the early days of the PC era, the main tools for developing applications were a programming language compiler, a very small set of standard libraries, and the raw operating system application programming interfaces (APIs)—a very basic set of low-level programming tools.

Even as developers were building applications using such basic tools, they were discovering more and more code that was repetitive and could be abstracted away through higher level APIs. Operating system vendors noticed that by providing such higher level APIs they could make it cheaper for developers to create applications for their systems. This increased the number of applications running on the system, which, in turn, would make the system more appealing to end users who demanded a variety of applications. Also, independent tool and component vendors quickly recognized the business opportunities in raising the API abstraction level.

In parallel, the industry started, slowly, to accept object-oriented design with its emphasis on extensibility and reusability.[1] When reusable library vendors adopted object-oriented programming (OOP) for the development of their high-level APIs, the concept of what we consider a *framework* was born. Application developers were no longer expected to write most of the application from scratch. The framework would provide most of the needed pieces that would then be customized and connected[2] to form applications.

As more vendors started to provide components, which were to be reused by stitching them together into a single application, developers noticed that some of the components did not fit together well. Their applications looked and worked like a house built by different contractors who never talked to each other. Likewise, as a larger percentage of application source code became constructed of API calls rather than standard language constructs, developers started to complain that they now had to read and write multiple languages: one programming language and several "languages" of the components they wanted to reuse. This had significant negative impact on developer productivity, and productivity is one of the main factors in the success of a framework. It became clear that there was a need for common rules that would ensure consistency and seamless integration of reusable components.

Today, most application development platforms spell out some kind of design conventions to be used when designing frameworks for the platform. Frameworks that do not follow such conventions, and so do not integrate well with the platform, are either a source of a constant frustration to

1. Object-oriented languages are not the only languages well suited for developing extensible and reusable libraries, but they played a key role in popularizing the concepts of reusability and extensibility. Extensibility and reusability are a large part of the philosophy of object-oriented programming (OOP), and the adoption of OOP contributed to increased awareness of their benefits.
2. Recently there has been a great deal of criticism of object-oriented (OO) design, claiming that the promise of reusability never materialized. OO is not a guarantee of reusability (especially without testing), but we are not sure that it was ever promised. On the other hand, OO provides natural constructs to express units of reusability (types), to communicate and control extensibility points (virtual members), and to facilitate decoupling (abstractions).

those trying to use them, or are at competitive disadvantage, and ultimately fail in the marketplace. The ones that succeed are often described as self-consistent, making sense, and finally well-designed.

1.1 Qualities of a Well-Designed Framework

The question is, then, what defines a well-designed framework, and how do you get there? There are many factors, such as performance, reliability, security, and so on, that affect software quality. Frameworks, of course, must adhere to these same quality standards. The difference between frameworks and other kinds of software is that frameworks are made up of reusable APIs, which presents a set of special considerations in designing quality frameworks.

1.1.1 Well-Designed Frameworks Are Simple

Although a good framework must also be powerful, most frameworks do not lack power because it can fairly easily be measured by stacking it up against the core functional requirements. On the other hand, simplicity often gets sacrificed when schedule pressure, feature creep, or the desire to satisfy every little corner-case scenario takes over the development process. However, simplicity is a must-have feature of every framework. If you have any second thoughts about the complexity of a design, it is almost always much better to cut the feature from the current release and spend more time to get the design right for the next release. If the design does not feel right, and you ship it anyway, you are likely to regret having done so.

Many of the guidelines described in this book are motivated by the desire to strike the right balance between power and simplicity. In particular, Chapter 2 talks extensively about some basic techniques used by the most successful framework designers to design the right level of simplicity and power.

1.1.2 Well-Designed Frameworks Are Expensive to Design

Good framework design does not happen magically. It is hard work that consumes lots of time and resources. If you are not willing to invest real money in the design, you should not expect to create a well-designed framework.

Framework design should be an explicit and distinct part of the development process[3]; explicit because it needs to be appropriately planned, staffed, and executed, and distinct because it cannot just be a side effect of the implementation process. Too often, we see cases where the framework is whatever types and members happen to remain public after the implementation process ends.

The best framework designs are either done by people whose explicit responsibility is framework design, or by people who can put the framework designer's hat on at the right time in the development process. Mixing the responsibilities is a mistake and leads to designs that expose implementation details, which should not be visible to the end user of the framework.[4]

1.1.3 Well-Designed Frameworks Are Full of Trade-Offs

There is no such thing as the perfect design. Design is all about making trade-offs, and to make the right decisions, you need to understand the options, their benefits, and their drawbacks. If you find yourself thinking you have a design without trade-offs, chances are you are missing something big as opposed to really finding the silver bullet.

The practices described in this book are presented as guidelines, rather than rules, exactly because framework design requires managing trade-offs. Some of the guidelines discuss the trade-offs involved and even

3. Do not misunderstand this as an endorsement of heavy up-front design processes. In fact heavy API design processes lead to waste as APIs often need to be tweaked after they are implemented. However the API design process has to be separate from the implementation process and has to be incorporated in every part of the product cycle: the planning phase (what are the APIs our customers need?), the design process (what are the functionality trade-offs we are willing to make to get the right framework APIs?), the development process (have we allocated time to try to use the framework to see how the end result feels?), the beta process (have we allocated time for the costly API redesign), and maintenance (are we decreasing the design quality as we evolve the framework?).
4. Prototyping is one of the most important parts of the framework design process, but prototyping is very different from implementation.

provide alternatives that need to be considered given the specifics of the situation.

1.1.4 Well-Designed Frameworks Borrow from the Past

Most successful frameworks borrow and build on top of existing proven designs. It is possible—and actually desirable—to introduce completely novel solutions into framework design, but it should be done with the utmost caution. As the number of new concepts increases, the probability that the overall design will be right goes down.

The guidelines contained in this book are based on the experiences we gained while designing the .NET Framework; they encourage borrowing from things that worked and withstood the test of time, and warn about ones that did not. We encourage you to use these good practices as a starting point, and to improve on them. Chapter 9 talks extensively about common design approaches that worked.

1.1.5 Well-Designed Frameworks Are Designed to Evolve

Thinking about how to evolve your framework in the future is a double-edged sword. It can lead to additional complexity in the name of "just in case," but it can also save you from shipping something that will degrade over time, or, even worse, something that will not be able to preserve backward compatibility.[5] As a general rule it is better to move a complete feature to the next release rather than half-doing it in the current release.

Whenever making a design trade-off, you should answer the question of how the decision will affect your ability to evolve the framework in the future. The guidelines presented in this book take this important concern into account.

1.1.6 Well-Designed Frameworks Are Integrated

Modern frameworks need to be designed to integrate well with a large ecosystem of different development tools, programming languages, application models, and so on. Distributed computing means the time of frame-

5. Backward compatibility is not discussed in detail in this book, but it should be considered one of the basics of framework design, together with reliability, security, and performance.

works designed for specific application models is over. This is also true of frameworks designed without thinking about proper tool support or integration with programming languages used by the developer community.

1.1.7 Well-Designed Frameworks Are Consistent

Consistency is the key characteristic of a well-designed framework. It is one of the most important factors affecting productivity. A consistent framework allows for transfer of knowledge between the parts of the framework that a developer knows to parts that the developer is trying to learn. Consistency also helps developers to quickly recognize which parts of the design are truly unique to the particular feature area and so require special attention, and which are just the same old common design patterns and idioms.

Consistency is probably the main theme of this book. Almost every single guideline is partially motivated by consistency, but Chapters 3 to 5 are probably the most important ones in describing the core consistency guidelines.

We offer these guidelines to help you make your framework successful. The next chapter presents guidelines for general library design.