Design Considerations for Application Framework Extensibility

John F. Gnazzo

Introduction

An application framework is a software library that provides a fundamental structure to support the development of applications for a specific environment. An application framework acts as the skeletal support to build an application. The intention of designing application frameworks is to lessen the general issues faced during the development of applications. This is achieved through the use of code that can be shared across different modules of an application. Application frameworks are used not only in the graphical user interface (GUI) development, but also in other areas like web-based applications. (1)

One important aspect of designing a framework is making sure the extensibility of the framework has been carefully considered. This requires that one understands the costs and benefits associated with various extensibility mechanisms. This blog helps one decide which of the extensibility mechanisms—subclassing, events, virtual members, callbacks, and so on, that can best meet the requirements of the framework.

There are many ways to allow extensibility in frameworks. They range from less powerful but less costly to very powerful but expensive. For any given extensibility requirement, one should choose the least costly extensibility mechanism that meets the requirements. Keep in mind that it's usually possible to add more extensibility later, but you can never take it away without introducing breaking changes. (2)

This blog will discuss extensibility mechanisms including

- Unsealed Classes
- Protected Members
- Events and Callbacks
- Virtual Members
- Abstractions
- Base Classes
- Sealing

Unsealed Classes

Sealed classes cannot be inherited from, and they prevent extensibility. In contrast, classes that can be inherited from are called **unsealed classes**.

Recommendation: Use unsealed classes with no added virtual or protected members as a great way to provide inexpensive yet much appreciated extensibility to a framework.

Protected Members

Protected members by themselves do not provide any extensibility, but they can make extensibility through subclassing more powerful. They can be used to expose advanced customization options without unnecessarily complicating the main public interface.

Framework designers need to be careful with protected members because the name "protected" can give a false sense of security. Anyone is able to subclass an unsealed class and access protected members, and so all the same defensive coding practices used for public members apply to protected members.

Recommendation: Use protected members for advanced customization. Do treat protected members in unsealed classes as public for the purpose of security, documentation, and compatibility analysis.

Events and Callbacks

Callbacks are extensibility points that allow a framework to call back into user code through a delegate. These delegates are usually passed to the framework through a parameter of a method.

Events are a special case of callbacks that supports convenient and consistent syntax for supplying the delegate (an event handler). In addition, Visual Studio's statement completion and designers provide help in using event-based APIs.

Virtual Members

Virtual members can be overridden, thus changing the behavior of the subclass. They are quite similar to callbacks in terms of the extensibility they provide, but they are better in terms of execution performance and memory consumption. Also, virtual members feel more natural in scenarios that require creating a special kind of an existing type (specialization).

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

The main disadvantage of virtual members is that the behavior of a virtual member can only be modified at the time of compilation. The behavior of a callback can be modified at runtime.

Virtual members, like callbacks (and maybe more than callbacks), are costly to design, test, and maintain because any call to a virtual member can be overridden in unpredictable ways and can execute arbitrary code. Also, much more effort is usually required to clearly define the contract of virtual members, so the cost of designing and documenting them is higher.

Abstractions

An abstraction is a type that describes a contract but does not provide a full implementation of the contract. Abstractions are usually implemented as abstract classes or interfaces, and they come with a well-defined set of reference documentation describing the required semantics of the types implementing the contract. Some of the most important abstractions in the .NET Framework include Stream, IEnumerable<T>, and Object.

You can extend frameworks by implementing a concrete type that supports the contract of an abstraction and using this concrete type with framework APIs consuming (operating on) the abstraction.

A meaningful and useful abstraction that is able to withstand the test of time is very difficult to design. The main difficulty is getting the right set of members, no more and no fewer. If an abstraction has too many members, it becomes difficult or even impossible to implement. If it has too few members for the promised functionality, it becomes useless in many interesting scenarios.

Too many abstractions in a framework also negatively affect usability of the framework. It is often quite difficult to understand an abstraction without understanding how it fits into the larger picture of the concrete implementations and the APIs operating on the abstraction. Also, names of abstractions and their members are necessarily abstract, which often makes them cryptic and unapproachable without first understanding the broader context of their usage.

However, abstractions provide extremely powerful extensibility that the other extensibility mechanisms cannot often match. They are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on. They are also extremely important for testability of frameworks. Good abstractions make it possible to stub out heavy dependencies for the purpose of unit testing. In summary, abstractions are responsible for the sought-after richness of the modern object-oriented frameworks.

Base Classes

Strictly speaking, a class becomes a base class when another class is derived from it. For the purpose of this section, however, a base class is a class designed mainly to provide a common abstraction or for other classes to reuse some default implementation though inheritance. Base classes usually sit in the middle of inheritance hierarchies, between an abstraction at the root of a hierarchy and several custom implementations at the bottom.

They serve as implementation helpers for implementing abstractions. For example, one of the Framework's abstractions for ordered collections of items is the IList<T> interface. Implementing IList<T> is not trivial, and therefore the Framework provides several base classes, such as Collection<T> and KeyedCollection<TKey, TItem>, which serve as helpers for implementing custom collections.

Base classes are usually not suited to serve as abstractions by themselves, because they tend to contain too much implementation. For example, the Collection<T> base class contains lots of implementation related to the fact that it implements the nongeneric IList interface (to integrate better with nongeneric collections) and to the fact that it is a collection of items stored in memory in one of its fields.

As previously discussed, base classes can provide invaluable help for users who need to implement abstractions, but at the same time they can be a significant liability. They add surface area and increase the depth of inheritance hierarchies and so conceptually complicate the framework. Therefore, base classes should be used only if they provide significant value to the users of the framework. They should be avoided if they provide value only to the implementers of the framework, in which case delegation to an internal implementation instead of inheritance from a base class should be strongly considered.

Sealing

One of the features of object-oriented frameworks is that developers can extend and customize them in ways unanticipated by the framework designers. This is both the power and danger of extensible design.

When you design your framework, it is, therefore, very important to carefully design for extensibility when it is desired, and to limit extensibility when it is dangerous.
A powerful mechanism that prevents extensibility is sealing. You can seal either the class or individual members. Sealing a class prevents users from inheriting from the class. Sealing a member prevents users from overriding a particular member.

Bibliography

1. *Application Framework.* **Jasnssen, Cory.** 2014, Techopedia.

2. *Designing for Extensibility.* **Microsoft.** s.l. : Microsoft Corporation, 2009.