Singularity Design Note 21

Extensibility Features for Sing#

INFORMAL DESIGN AND RATIONALE
VERSION 0.6
MARCH 8, 2006

This document motivates and describes the design for type extension features in Sing#. Most of this design is now implemented.

This document is a work in progress.

> *NOTE: Sing# code containing type extensions must be compiled with the `/allowTypeExtensions` flag passed to sgc.exe, and the resulting .exe MSIL file must be run by Bartok. Code compiled by sgc with the `/allowTypeExtensions` flag will not run, and will probably not even verify, on a stock .NET VM.*
>
> *NOTE: Currently, the `Bartok.SystemExtension.dll` file, which contains the `Microsoft.Bartok.Options.MixinAttribute` and `...MixinExtendAttribute` classes, should be in the current directory when running sgc.exe.*

1. Type Extensions

Consider this scenario:

```
class C1 {
      // some core members etc.
      ...

      // some members related to feature F1
      ...

      // some members related to feature F2
      ...
}
class C2 {
      // some core members etc.
      ...

      // some members related to feature F2
      ...

      // some members related to feature F3
      ...
}
```

We want to be able to write the code specific to features F1, F2, and F3 separately from the underlying classes and each other. Because:

- we may wish to build configurations of the system w/ or w/o a given feature
- we may wish to have alternative implementations of some feature that we can select among
- we may wish to organize our code better, to yield a narrower, simpler default interface to the base classes, to allow clients of a class to explcitly augment its interface only with those features they require, and to allow specifying the interfaces between classes and their features (e.g. F1 and F2 in C1 may not need to access each other)

Below is an alternative way to express the code. An `extend` declaration adds new members to an existing class, struct, or interface. The new members of the extension are visible only where the scope containing the extend declaration is visible. To manage this visibility, `extend` declarations can be put into a namespace. An `extend` declaration is visible to clients in the same scope, to clients in a lexically nested scope, and, if the `extend` declaration is in a namespace, to clients that "import" the namespace via a `using` declaration. [Currently, there is no way to distinguish using a namespace to gain access to its types under a shorter name, and using a namespace to gain access to its extensions. Hopefully, they will not conflict in practice. It would be easy to add support for distinguishing these two kinds of using, e.g. through an additional keyword on the using declaration.] Multiple classes can be extended in a single scope, e.g., if the feature spans classes. Of course, a single class can be extended by several extensions, and each extension can be put in a different scope. (Unlike partial classes, where all parts of a class must be declared in the same scope, and in fact must be compiled together from source., and clients either see all parts of a class or none.)

```
// Core classes and operations
namespace Core {
 class C1 {
      // some core members etc.
      ...

      // (no feature-specific operations)
 }
 class C2 {
      // some core members etc.
      ...
 }
}

// Feature F1
namespace F1 {
 using Core;
 extend class C1 {
      // some members related to feature F1
      ...
 }
}

// Feature F2
namespace F2 {
 using Core;
 extend class C1 {
      // some members related to feature F2
      ...
 }
 extend class C2 {
      // some members related to feature F2
      ...
 }
 // maybe also some helper classes specific to feature F2, which
 // C1 and C2 no longer need to know about
}

// Alternative version of Feature F2
namespace F2b {
 using Core;
 extend class C1 {
      // some members related to feature F2
      ...
 }
 extend class C2 {
      // some members related to feature F2
      ...
 }
}

// Feature F3
namespace F3 {
```

```
 using Core;
 extend class C2 {
      // some members related to feature F3
      ...
 }
}

// A client of C1 that uses only feature F2
namespace Client {
  using Core;
  using F2;
  ... new C1() ... calls of C1.F2 operations ...
}
```

It is possible for two different extensions of the same core class to add members with the same names. Such members are semantically distinct, but with overloaded names. This is completely legal. (Unlike partial classes, where multiple members with the same name (and other characteristics such as argument types) cannot be declared, even across class parts.) If a client sees both extensions (e.g., by using the namespaces containing the extensions), and tries to access the overloaded member name, and there is no other way to resolve the overloading (e.g. distinct numbers or static types of arguments, for method members), then the client will receive an "ambiguous overloaded member access" error. (There currently is no mechanism to resolve such ambiguous accesses, but one could be added.)

An extension to a class, struct, or interface may also specify additional "superinterfaces", which makes the extended type also implement the given interfaces. This is known as "retroactive abstraction". Previously it has been viewed as an important facility in adapting preexisting classes to new contexts (in the absence of structural subtyping). But in the context of extensions, it allows the fact that a type implements some interface via a set of operations to be moved out into a separate extension; the extension specifies the operations along with the fact that the extended class implements the interface. [While technically possible in some cases, I don't advocate allowing a superclass to be added via extension.]

One extension can access new members introduced by another extension, simply by making the other extension visible to the first extension.

In general, a class, struct, or interface declaration can be preceded by an `extend` keyword, which means that any superinterfaces and members declared in the extend declaration are *logically* added to the underlying class, struct, or interface, which should be previously declared.

- The limitations on the kinds of members allowed in the extendee (e.g. those for structs and interfaces) are applied to the extender.
- Visible redeclarations (e.g. between the extendee and the extender, or between an extender and another extension visible to the extender) that wouldn't be allowed in the original class, e.g. redeclaring a member with the same signature, are disallowed in extensions. (But see the support for extensions to existing function members described in the next section.)
- An abstract method or property may be added to an abstract receiver class, but a warning is issued. All concrete subclasses of the abstract class must also be extended with overriding non-abstract methods or properties, no later than link-time.
- An extension cannot add a new destructor or static constructor, since classes can only have one of them. (But see the support for extensions to existing function members described in the next section.)
- To extend a class nested in another class, the outer class is extended with a body that contains an extension of the nested class. [Alternatively, one could allow a qualified name as the extendee class or interface.]

(Access privileges for extensions is discussed in a later section.)

NOTE: In the current implementation, type extensions must be compiled from source at the same time that all clients using the augmented interface are compiled; there is no way to compile a type extension separately from clients that make use of members added by the extension. The extendee types can *be compiled separately, however.*

NOTE: In the current implementation, structs cannot be extended with new instance fields. (I need to develop a reasonable approach to struct field initialization that works in the face of extensions; the standard semantics in C# doesn't.)

NOTE: In the current implementation, nested types cannot be added to another type via extension. (This is a Bartok limitation.)

NOTE: In the current implementation, interface inheritance cannot be added to another type via extension. (This is a Bartok limitation.)

2. Function Member Extensions

Here's another scenario:

```
// A base class:
class C1 {
  // fields
  ...

  // static constructor
  static C1() { ... }

  // instance constructors
  C1(int x, int y) { ... }
  ...

  // destructor
  ~C1() { ... }

  // methods
  virtual int m1(int a, int b) { ... }
  ...

  // a property
  int p1 {
    get { ... }
    set { ... }
  }
  // an indexer
  int this[int i, string s] {
    get { ... }
    set { ... }
  }
  // events
  event DD e1;
  event DD e2 {
    add { ... }
    remove { ... }
  }

  // operators
  public static int operator !(C1 c) { ... }
  public static int operator +(C1 c, int i) { ... }
  public static implicit operator C1(int x) { ... }
  public static explicit operator int(C1 c) { ... }
}

// An extension that adds some static & instance fields to the
// base, and also extends some constructors and methods to deal
// with the new fields
namespace F4 {

 extend class C1 {
  // add some fields to C1, and give them default initial values
  static int f1 = ...;
  int f2 = ...;
```

```
  // augment the static constructor (implicitly extend)
  static C1() { ... }

  // augment some instance constructors
  extend C1(int x, int y) { ... }
  ...

  // augment the destructor
  extend ~C1() { ... }

  // augment some methods
  extend int m1(int a, int b) {
    ... int a2, b2; ... int res = extendee(a2, b2); ... return ...; }
  ...

  // augment a property
  extend int p1 {
    get { ... int res = extendee(); ... return ...; }
    set { ... extendee(value2); ... }
  }
  // augment an indexer
  extend int this[int i, string s] {
    get { ... int res = extendee(i2, s2); ... return ...; }
    set { ... extendee(i2, s2, value2); ...  }
  }
  // augment some events
  extend event DD e1 {
    add { ... extendee(value2); ... }
    remove { ... extendee(value2); ... }
  }
  extend event DD e2 {
    add { ... extendee(value2); ... }
    remove { ... extendee(value2); ... }
  }

  // augment some operators
  extend public static int operator !(C1 c) {
    return ... extendee(c2) ...; }
  extend public static int operator +(C1 c, int i) {
    return ... extendee(c2,i2) ...; }
  extend public static implicit operator C1(int x) {
    return ... extendee(x2) ...; }
  extend public static explicit operator int(C1 c) {
    return ... extendee(c2) ...; }

 }

}
```

This example illustrates that existing function-like members of a class (instance and static constructors, destructors, instance, virtual, and static methods, operators, indexers, and property and event accessor methods) can be augmented by an extension, i.e., that new code can be "injected" into them. (Unlike partial classes, which do not support function-like member extension.) A function-like member extension must be of

the same kind and have have the same argument types, result type, and modifiers as the extended member. Whenever the original code would have been invoked, the extended code is run instead. For a method, operator, indexer, accessor, or destructor, the extending code can choose when and whether to call the original function-like member, using an extendee call. Thus, an extension of a function-like member is akin to an "around" method, which can easily simulate either a "before" or an "after" method. An extendee call has no explicit receiver, but takes the same number and types of arguments as the function-like member being invoked, and returns the same type of result as the invoked member. Extendee calls can be used even in property, indexer, and event accessors, operators, and destructors. For static and instance constructors, the extending code automatically begins with an extendee call; constructor extensions cannot include an explicit extendee call, nor can they include this or base initializers. A static constructor in a class or struct extension is automatically an extension of the (implicit or explicit) static constructor of its extendee class or struct. [Currently, there are no limits on the number of extendee calls that can be included in an extension, either statically or dynamically. Just as there are no limits on the number of base calls in a method.]

An extension to a function member is always executed, even if the extension is not visible to the invoking client. Visibility of an extension is only relevant when determining which member names are visible to a client, not what code is run when a visible function-like member is invoked.

In general, inside an extend-type declaration, function-like members (i.e. methods, operators, indexers, properties, property-style events, instance constructors, static constructors, and destructors) can be preceded by an extend keyword, which means that their bodies extend the function member with the same signature of the extended class. There should be a unique previous declaration of the extendee function member with the same signature and modifiers declared by the extended class, or by some visible extension of the extended class. For an extension of a read-write property, the extension can but doesn't need to extend both the get and set accessors. [It is unclear what the rules ought to be for extending just the add or remove accessor of an event, or just the == but not != operator, or just the optrue but not the opfalse operator, etc. Probably should continue the requirement that all related operators must be extended if any are.]

3. Order of Extension

Several separate extensions can extend the same core type. If more than one extension extends the same function-like member, then the order in which the function-like member extensions are invoked becomes visible to the programmer. We require that overlapping extensions (extensions that extend the same function-like member with new code) be totally ordered, with one extension extending the core type, a second extension extending the first, a third extension extending the second, and so on. Then whenever the function-like member is invoked (either by an outside client or via a base call from a subclass) the code for the function-like member in the most-extending extension is run first. If this code makes an `extendee` call, then the code in the next-most-extending extension is run, and so on.

Currently, one extension extends another if the second extension is visible to the first, but not vice versa. Mutually visible extensions are unordered, and extensions that do not see each other (i.e., are independent) are unordered. Unordered extensions should have disjoint sets of function-like member extensions. This disjointness of unordered extensions is checked when a group of extensions is compiled, and also checked when independently compiled assemblies are linked.

> *NOTE: Currently, all extensions of a type must be compiled at the same time, so there is no need for a link-time check.*

Static constructors are not considered to overlap; multiple independent extensions can define static constructors, and the constructors will be run in some unspecified order. Likewise, field initializers in unordered extensions are evaluated in unspecified order when an instance is created.

To illustrate, here is a second extension of C1:

```
namespace F5 {
 using F4;   // this extension builds on the F4 extension

 extend class C1 {
  // add some fields to C1, and give them default initial values
  static int f3 = ...;
  int f4 = ...;

  // augment the static constructor (implicitly extend)
  static C1() { ... }

  // augment some instance constructors
  extend C1(int x, int y) { ... }
  ...

  // augment the destructor
  extend ~C1() { ... }

  // augment some methods
  extend int m1(int a, int b) {
    ... int a2, b2; ... int res = extendee(a2, b2); ... return ...; }
  ...
  // other extensions
  ...
 }

}
```

This F5 extension uses the F4 extension, and so the extension in F5 extends the extension in in F4, which in turn extends the core C1 class. Invocations of the m1 method on a C1 instance first calls the code in F5. If and when this code executes the extendee call, the code in F4's m1 method will be run. If and when that code executes the extendee call, the code in C1's original m1 method will be run. As another example, when creating a new instance of C1, first the instance constructor for C1 in F5 is run. This constructor implicitly first executes an extendee call, which invokes the constructor in F4. That constructor implicitly first executes an extendee call to invoke the original constructor in C1. When that constructor returns, F4's constructor will resume. When it completes, F5's constructor will resume.

4. Access Privileges Granted to Extensions

Another important issue with extensions is the access privileges (i.e., public, internal, protected, protected internal, or private) granted to the extension. The most conservative answer is to grant only public (or internal) access to the extended type, which is all that any other regular outside client would have access to. But this limits the scenarios in which a single large class could be broken up into multiple separate extensions; this would be desirable to allow constructing subsets of the total functionality or to choose among alternative implementations of some feature, but oftentimes these extensions may need more privileged access to the internals of the underlying class.

To address this, the extension specifies how much access privilege it needs to members of the extended class or struct, e.g. by specifying an access modifier after the 'extends' keyword:

```
extend private class C1 {
  ... access to C1's private members granted here ...
}
```

Since interface members have no access privileges (they're all public), no access privilege is specified when extending an interface:

```
extend interface I {
  ...
}
```

A related question is what access one extension is granted to another extension's additions. First, the second extension must have the first extension in scope. Then, the same access modifier applies to all the members of the original class and all its in-scope extensions.

NOTE: The current implementation always grants private access privilege to extensions.

5. Extensibility Restrictions

A class may specify how it may be extended, and what access privileges to its members are granted to extensions. At one extreme, any extensions are allowed. At the other extreme, no extensions are allowed (this might be true of a class that also disallows subclassing). In between, a specific set of extensions or extenders are allowed. A large number of possible designs exist for specifying various coarser- or finer-grained constraints on potential extenders. Given this wide range of possibilities, perhaps the right solution is to define a set of attributes for expressing these constraints. For instance:

- On a class or interface:
  - `[Extensible(`*`Access`*`)]`: the type can be extended with new members or extensions of existing members by any extensions, with the given access privilege granted.
  - `[ExtensibleBy("`*`F1`*`", `*`Access`*`),...]`: the type can be extended by extensions in the named module, and those extensions are granted the given access privilege. One could also imagine a mechanism to organize modules into "families", e.g. the GC family, and then grant extensibility privileges to all modules in that family, without explicitly listing them all.
  - `[NotExtensible]`: the type can't be extended. akin to `sealed`.

A class or interface can specify the `NotExtensible` attribute, or one or more `Extensible` and `ExtensibleBy` attributes, or no extensibility attributes. DESIGN CHOICE: can require an extensibility attribute, or default to `Extensible(ProtectedInternal)`, unless otherwise specified e.g. by compiler command-line arguments or other system configuration specifications.

- On a member:
  - `[ExtenderAccess(`*`Access`*`)]`: the member grants no more than Access privilege to any otherwise legal extenders.
  - `[ExtenderAccess("`*`F1`*`", `*`Access`*`)]`: the member grants no more than Access privilege to any otherwise legal extenders in namespace F1.
  - `[NoExtenderAccess]`: the member is not accessible to any extenders.

These attributes allow additional access restrictions to be placed on members beyond those allowed generically to extensions.

- On function members:
  - `[Extensible]`: the function member can be extended by any extensions.

- `[ExtensibleBy("F1"),...]`: the function member can be extended by extensions in the named namespace.
- `[NotExtensible]`: the function member can't be extended.

A function member can specify the `NotExtensible` attribute, the `Extensible` attribute, or one or more `ExtensibleBy` attributes, or no extensibility attributes. If the latter, then the function member's extensibility defaults to `Extensible`, unless otherwise specified e.g. by compiler command-line arguments or other system configuration specifications.

NOTE: Extensibility restrictions are not implemented. All types are extensible, and all members of all types are extensible.

6. Implementation Strategies

Multiple implementation strategies are possible, with differing trade-offs between expressiveness and performance:

- completely separate compilation into bog-standard MSIL
  - limits what kinds of extensions can be supported to adding new members to existing classes, with only public accessibility granted, and limits the efficiency of some of these extensions, particularly dispatching of virtual function members and storage of instance fields
  - cannot augment existing function members
  - cannot get more privileged access
- anticipated separate compilation, where the extendee class and/or function member is compiled in such a way to allow extensions to be added later
  - can support all kinds of extensions, including function member extensions, except for adding new superinterfaces to existing classes
  - can support more efficient instance fields (but still not as efficient as original fields)
  - may be able to support more efficient virtual function members
  - could get more privileged access, at a loss of strict protection at the MSIL level
- merged compilation where the class and (some of) its extensions are compiled together (a la partial classes)
  - allows all extensions, as efficiently as original members
  - allows all access privileges, without sacrificing encapsulation at the MSIL level
  - can still be extended later, under any of the compilation strategies
  - in general, a compilation takes a set of classes/modules and a set of other assemblies and produces a new assembly. this compilation is allowed to perform as much (re)optimization of the extensions in its view as it likes
  - whole-program compilation yields "perfect" results
- can compile a single source class/module in multiple ways, each specialized to some extensibility context

  - later compilations choose which specialization to build on

- dynamic patching, dynamic compilation
  - allow more flexibility & efficiency for code compiled separately

A given system can mix and match implementation strategies for different parts of the system.

> *NOTE: The current implementation has the Sing# front-end (sgc) generate regular MSIL + a few Bartok-defined attributes, and then lets Bartok do whole-program weaving of the generated files. The MSIL can be compiled by Bartok, but it doesn't completely follow regular MSIL type-verification rules, so it cannot be verified by a standard MSIL verifier.*

7. Discussion

[To be fleshed out later]

vs. aspects
vs. subclass mixins & global name rebinding
vs. traits & ordered MI
module alternatives

8. Syntax

This syntax extends the official C# grammar (ellipses refer to either all the old alternatives for some non-terminals, or that the new version is just like the previous version after this point):

```
class-declaration:
  ...
  attributesopt "extend" extension-modifiersopt "class" ...

extension-modifiers:
  extension-modifier
  extension-modifiers extension-modifier

extension-modifier:
  "public"
  "protected"
  "internal"
  "private"

class-member-declaration:
  ...
  import-declaration

struct-declaration:
  ...
  attributesopt "extend" extension-modifiersopt "struct" ...

struct-member-declaration:
  ...
  import-declaration

interface-declaration:
  ...
  attributesopt "extend" "interface" ...

interface-member-declaration:
  ...
  import-declaration

method-header:
  ...
  attributesopt "extend" method-modifiersopt return-type ...

property-declaration:
  ...
  attributesopt "extend" property-modifiersopt type ...

event-declaration:
  ...
  attributesopt "extend" event-modifiersopt "event" type member-name
    "{" event-accessor-declarations "}"

indexer-declaration:
  ...
  attributesopt "extend" indexer-modifiersopt indexer-declarator ...

operator-declaration:
  ...
```

```
  attributesopt "extend" operator-modifiersopt operator-declarator ...

constructor-declaration:
  ...
  attributesopt "extend" constructor-modifiersopt constructor-declarator ...

static-constructor-declaration:
  ...
  attributesopt "extend" static-constructor-modifiers identifier ...

destructor-declaration:
  ...
  attributesopt "extend" "~" identifier ...

declaration-statement:
  ...
  import-declaration

primary-no-array-creation-expression:
  ...
  “extendee” “(“ argument-listopt “)”
```