Detection of Breaking Changes

This document outlines the kinds of breaking changes we will detect – source level breaking changes – which will cause customer source code compiled against an older version of our assemblies to not compile anymore. This document is not concerned with binary breaking changes – those that will allow code to be recompiled, but will not allow a compiled assembly referencing an older version of our assembly to automatically start using the newer version of our assembly.

* Definitions
  1. **Assembly** – A compiled DLL or EXE
  2. **Externally visible member** – A public member of an externally visible type or a protected or protect internal member of an unsealed, externally visible class. These do not include destructors
  3. **Externally visible type** – A public type at the root level of a namespace or a public, protected, or protected internal type that is nested within an externally visible type
  4. **Member** – A constant, field, method, property, indexer, event, operator, constructor, destructor, or nested type within a class, struct, or interface
  5. **Type** – A class, struct, interface, enum, or delegate declaration
* Types of breaking changes we will detect
  1. Assembly-level breaking changes:
     1. Removed an assembly with at least one externally visible type
  2. Type-level breaking changes:
     1. Changing the accessibility of an externally visible type or member from public to protected or protected internal
     2. Removing an externally visible type *(this includes changing the visibility to internal or private, because it will cause the type no longer be externally visible)*
     3. Changing an externally visible type’s name *(this will be detected by 2.b because a changed name will be seen as the old type being removed and a new type being added)*
     4. Changing an externally visible class with a public constructor to be abstract
     5. Changing an externally visible class with a public constructor or an externally visible unsealed class with a protected constructor to be static
     6. Adding or removing generic type parameters on an externally visible type *(this will be detected by 2.b because changed generic parameters will be seen as the old type being removed and a new type being added)*
     7. Changing the base class of an externally visible type to something which is not derived from the previous base type
     8. Adding an interface with members to the base interface list of an externally visible interface
     9. Removing an externally visible interface from the list of implemented interfaces on an externally visible type *(Note that if an interface name is removed from the list in code, the interface could still be implemented by the type if another interface being implemented by that type inherits from the removed interface – this situation is not a breaking change)*
     10. Changing an externally visible class with a public or protected constructor from unsealed to sealed
  3. Generic type parameter-level breaking changes
     1. Changing the constraints on a generic type parameter of an externally visible type
     2. Changing the variance of a contravariant or covariant *(in or out variance modifier, respectively)* generic type parameter on an externally visible interface or delegate
  4. Member-level breaking changes
     1. Adding a member to an externally visible interface
     2. Removing a member from an externally visible type *(Note: we will make some exceptions to this rule if an “equivalent” member is added which allows code accessing the old member to access the new member. These kinds of exceptions are when:*
        + *A constant is removed and a static read-only field is added with the same name, type, and a greater than or equal accessibility*
        + *A constant is removed and a static property is added with the same name, type, and a greater than or equal get accessor accessibility*
        + *A static read-only field is removed and a constant is added with the same name, type, and a greater than or equal accessibility*
        + *A read-only field is removed and a property is added with the same name, type, static or instance type, and a greater than or equal get accessor accessibility (Note: read-write field cannot be replaced because the field could have been passed as a ref or out parameter to a method and a property cannot)*
        + *A static, read-only, non-virtual property is removed and a constant is added with the same name, type, and a greater than or equal accessibility*
        + *A non-virtual property is removed and a field is added with the same name, type, access restrictions, and static or instance type, and a greater than or equal accessibility*
     3. Refactoring members of an externally visible interface by moving them to a base interface *(explicit implementations of the original interface would be using the wrong interface name specifier and therefore would break; this will be detected by both 4.a and 4.b because it will be removed from the derived interface and added to the base)*
     4. Changing an externally visible member’s name *(this will be detected by 4.b because a changed name will be seen as the old member being removed and a new member being added)*
     5. Changing an implicit interface member implementation into an explicit one on an externally visible type where the interface is also externally visible *(this will be detected by 4.b because the missing implicit implementation will be viewed as a removed member)*
     6. Adding or removing parameters to externally visible delegates, methods, indexers, or constructors *(Note: adding optional or params parameters does not cause a breaking change because the code using the original method will still compile)*
     7. Changing the (return) type of an externally visible delegate, constant, field, method, property, indexer, event, or operator to something that is not assignable to the previous entity’s type *(Note: changes from void to something else are allowed; read/write fields, properties, and indexers cannot change type at all; for write-only properties and indexers, this assignability check needs to be reversed: the type of the old member needs to be assignable to the type of the new member)*
     8. Changing a static member on an externally visible class or struct to be an instance member, or vice versa
     9. Changing an externally visible read/write field to read-only
     10. Changing an externally visible property from (externally) read/write to (externally) read-only or (externally) write-only or changing a public get/set accessor to protected or protected internal.
     11. Adding an abstract event, method, property, or indexer to an externally visible abstract class with a public or protected constructor
     12. Changing an externally visible event, method, property, or indexer in an unsealed class with a public or protected constructor to be abstract
     13. Changing an externally visible virtual event, method, property, or indexer to be non-virtual on a class with a public or protected constructor
     14. Sealing (adding a sealed override or changing an existing override to sealed) an externally visible event, method, property, or indexer to an unsealed class with a public or protected constructor
     15. Removing an override of an externally visible abstract event, method, property, or indexer from an abstract class with a public or protected constructor
  5. Parameter-level breaking changes
     1. Removing a default value from a parameter of an externally visible delegate, method, indexer, or constructor *(Note: this will be detected by 4.f because this will be equivalent to adding a required parameter)*
     2. Removing the params modifier on a parameter of an externally visible delegate, method, indexer, or constructor *(Note: this will be detected by 4.f because this will be equivalent to adding a required parameter)*
     3. Changing the type of a parameter to an externally visible delegate, method, indexer, or constructor in such a way that the call site could have an error. For non-modified parameters, this means changing the type to anything not assignable to the original type. For ref or out parameters, this means changing to type to anything other than the original type
     4. Renaming parameters of externally visible delegates, methods, indexers, or constructors *(call sites using named parameters will not compile)*
     5. Changing the ref/out modifier on a parameter of an externally visible delegate, method, indexer, or constructor
     6. Removing the “this” modifier on the first parameter of an externally visible extension method
* Additional Support
  1. The following attributes can be defined in the assembly being checked (or a referenced assembly) and used via duck typing by the detector. All attributes must be defined in the namespace “Infragistics”:
     1. VersionComparisonNameAttribute – An assembly-level attribute which should have the property “Name” returning a name which will not change across different versions of equivalent assemblies. For example, Infragitsics4.Documents.Excel.v13.2.dll should have a VersionComparisonNameAttribute with a Name of “Infragitsics4.Documents.Excel”. This will allow the detector to automatically match up assemblies form different versions.
     2. NamespaceRenamedAttribute – An assembly-level attribute which should have properties “NewName” and “OldName”. This allows types in a renamed namespace to be matched up and compared against associated types in the previous versions. The namespace rename will get reported as a breaking change, but the changes to the full names of the contained types will not.
     3. TypeRenamedAttribute – A type-level attribute which should have the property “OldName”. This allows a renamed type to be matched up and compared against the associated type in the previous versions. The type rename will get reported as a breaking change. *Note: for generic types, the old type name should not contain a generic type parameter list. In addition, it is not possible to use this attribute for a type which has had generic type parameters added or removed.*
* Types of breaking changes we WILL NOT detect
  1. Adding a new namespace in which an externally visible type resides *(the new namespace could have the same full name as a type defined by the customer, causing a build error)*
  2. Adding an externally visible type *(the new type could have the same full name as a type defined by the customer, causing a build error)*
  3. Adding a new implicit conversion operator to an externally visible type *(this could cause overload resolution to find multiple acceptable candidates instead of just one, which would cause a compilation error)*
  4. Adding an externally visible instance method to a type *(there could have been an extension method defined in customer code for the type with the same name and parameters – after upgrading, their code will bind to the new instance method and not the extension method, and if the return types differ, there could be a build error [not to mention the semantic error of a different method being called unintentionally])*
  5. Adding an externally visible method, indexer, or constructor overload *(this could cause overload resolution to find multiple acceptable candidates instead of just one, which would cause a compilation error)*
* Types of non-breaking changes we will detect
  1. Changing the default value of a method parameter
* Other questions/considerations
  1. Should we have an attribute for a “known” breaking change that should not be reported?
  2. Should we do a pass on the breaking changes and remove certain redundant errors? Examples:
     1. An interface is removed and we have an error for that and another saying that a type which implemented it no longer does
     2. A type is changed to abstract and had its public constructor changed to protected
     3. If a parameter was changed completely, the type and name will most likely change. We should probably only warn about the type change.
* TODO:
  1. Properties and indexers that are read-only are covariant, write-only are contravariant, read-write are invariant
  2. We can probably be a bit smarter about the constraint changes on type parameters. Changing an in-only parameter to a contravariant type or an out-only parameter to a covariant type should be allowed.