FDO and Related Infrastructure Systems in Re-architected FieldWorks[[1]](#footnote-1)

Table of Contents

[1 General issues and attitudes 5](#_Toc238450049)

[1.1 Code quality should improve 5](#_Toc238450050)

[1.2 Follow the ‘DRY’ principle 5](#_Toc238450051)

[1.3 Unit tests 5](#_Toc238450052)

[1.4 Null object pattern 5](#_Toc238450053)

[1.5 General New Skills 5](#_Toc238450054)

[1.6 Old code still around 6](#_Toc238450055)

[1.7 Build problems 6](#_Toc238450056)

[1.8 Installing Firebird Server 6](#_Toc238450057)

[2 FDO 6](#_Toc238450058)

[2.1 Interfaces 7](#_Toc238450059)

[2.1.1 CmObject Interfaces 7](#_Toc238450060)

[2.1.2 Factory Interfaces 7](#_Toc238450061)

[2.1.3 Repository Interfaces 8](#_Toc238450062)

[2.1.4 FDO Vector interfaces 8](#_Toc238450063)

[2.1.5 Alternative string and unicode interfaces 8](#_Toc238450064)

[2.1.6 Internal interfaces 9](#_Toc238450065)

[2.2 Implementations 9](#_Toc238450066)

[2.2.1 CmObject Implementations 9](#_Toc238450067)

[2.2.2 Factory Implementations 9](#_Toc238450068)

[2.2.3 Repository Implementations 9](#_Toc238450069)

[2.2.4 FDO Vector implementations 9](#_Toc238450070)

[2.2.5 FDO MultiString & MultiUnicode classes 9](#_Toc238450071)

[2.3 Statics on CmObject Implementation classes 9](#_Toc238450072)

[2.4 FDO & UOW interaction 10](#_Toc238450073)

[2.5 FDO & Persistence Code Interaction 10](#_Toc238450074)

[2.6 FdoCache 10](#_Toc238450075)

[2.7 Service Locator and Dependency Injection 10](#_Toc238450076)

[2.8 HVOs 11](#_Toc238450077)

[2.9 Back references 11](#_Toc238450078)

[2.10 FDO Namespaces 12](#_Toc238450079)

[2.10.1 SIL.FieldWorks.FDO 12](#_Toc238450080)

[2.10.2 SIL.FieldWorks.FDO.Application 13](#_Toc238450081)

[2.10.3 SIL.FieldWorks.FDO.Application.Impl 13](#_Toc238450082)

[2.10.4 SIL.FieldWorks.FDO.DomainImpl 13](#_Toc238450083)

[2.10.5 SIL.FieldWorks.FDO.DomainServices 13](#_Toc238450084)

[2.10.6 SIL.FieldWorks.FDO.Infrastructure 14](#_Toc238450085)

[2.10.7 SIL.FieldWorks.FDO.Infrastructure. Impl 14](#_Toc238450086)

[3 Supporting Infrastructure 14](#_Toc238450087)

[3.1 Persistence 14](#_Toc238450088)

[3.1.1 Creating a new BEP 15](#_Toc238450089)

[3.1.2 Unit Testing a new BEP 17](#_Toc238450090)

[3.2 Unit Of Work 19](#_Toc238450091)

[3.3 IdentityMap 21](#_Toc238450092)

[3.4 Undo/Redo 21](#_Toc238450093)

[3.5 Metadata Cache (IFwMetaDataCacheManaged : IFwMetaDataCache) 21](#_Toc238450094)

[3.6 DomainDataByFlid (ISILDataAccessManaged : ISILDataAccess) 21](#_Toc238450095)

[3.7 Decorators (Metadata cache & DomainDataByFlid) 22](#_Toc238450096)

[3.8 Repositories 23](#_Toc238450097)

[4 Upgrading Issues 23](#_Toc238450098)

[4.1 Applications 23](#_Toc238450099)

[4.1.1 Flex 24](#_Toc238450100)

[4.1.2 TE 24](#_Toc238450101)

[4.1.3 Data Notebook 24](#_Toc238450102)

[4.1.4 List Editor 24](#_Toc238450103)

[4.2 Common Upgrade issues 24](#_Toc238450104)

[4.2.1 ISILDataAccess vs. CmObject use 24](#_Toc238450105)

[4.3 FDO code 25](#_Toc238450106)

[4.3.1 Repositories 25](#_Toc238450107)

[4.3.2 Factories 25](#_Toc238450108)

[4.3.3 Obsolete namespaces in FDO 25](#_Toc238450109)

[4.3.4 FdoCache 25](#_Toc238450110)

[4.3.5 FDO Vector classes 25](#_Toc238450111)

[4.3.6 FDO MultiString & MultiUnicode classes 26](#_Toc238450112)

[4.3.7 Virtual Property Handler class & related interface 26](#_Toc238450113)

[4.3.8 Obsolete methods 28](#_Toc238450114)

[4.3.9 Instance methods not found on some CmObject class 28](#_Toc238450115)

[4.3.10 IVwCacheDa interface 28](#_Toc238450116)

[4.3.11 ScrFdo 28](#_Toc238450117)

[4.3.12 Porting assemblies 28](#_Toc238450118)

[4.3.13 Unit tests 29](#_Toc238450119)

[4.3.14 What won’t work yet 29](#_Toc238450120)

[4.4 A Worked Example 29](#_Toc238450121)

[4.4.1 Fix FDO ‘using’ declarations 29](#_Toc238450122)

[4.4.2 Make everything use interfaces and Tags 29](#_Toc238450123)

[4.4.3 Undotasks/UnitOfWork 30](#_Toc238450124)

[4.4.4 Object Creation 30](#_Toc238450125)

[4.4.5 Get rid of calls to PropChanged 31](#_Toc238450126)

[4.4.6 Dealing with HVOs 31](#_Toc238450127)

[4.4.7 Implementations of PropChanged 31](#_Toc238450128)

[4.4.8 Writing systems and multistrings 32](#_Toc238450129)

[4.4.9 Stuff on the implementation class 32](#_Toc238450130)

[4.4.10 Valid objects, dummy objects, and the like 32](#_Toc238450131)

[4.4.11 Working with flids 32](#_Toc238450132)

[4.4.12 Vanished methods 33](#_Toc238450133)

[4.4.13 Wrapping up 33](#_Toc238450134)

[5 Unit Test Framework 33](#_Toc238450135)

[5.1 FdoTestBase (abstract) 33](#_Toc238450136)

[5.2 PersistingBackendProviderTestBase (abstract) 33](#_Toc238450137)

[5.3 MemoryOnlyBackendProviderTestBase (abstract) 34](#_Toc238450138)

[6 Starting Flex-The window Frame 34](#_Toc238450139)

# General issues and attitudes

## Code quality should improve

It might seem as if this should go without saying, but I want to emphasize it anyway: the primary goal of the re-architecture is to improve the code: to make it easier to maintain, to enhance, and to keep free of bugs. If you find yourself tempted to do something you think is ugly and prone to cause bugs later because something is awkward to fix properly for the new architecture, resist the temptation! If you can’t think of a clean, maintainable way to get something working again, get help. If there’s a problem with the new architecture, Randy and I want to know and fix it, not have people do bad things to work around it.

## Follow the ‘DRY’ principle

That is, “Don’t repeat yourself.” We want to be removing code duplication wherever possible, and certainly not introducing it. If you find yourself writing or modifying essentially the same algorithm in two places, figure out a way to put the common parts in just one place. The clean code, design patterns, and agile principles books are good places to learn techniques for doing this. If you can’t think of a way to avoid duplication, ask at least one other person for ideas.

One possible exception is algorithms that are needed in both C++ code and C# code. If there is only a moderate amount of duplication, it may not be worth the trouble to wrap everything necessary in COM interfaces. In such cases (and anywhere else you can’t avoid duplication), comment in both places conspicuously indicating that the duplicate needs to be kept in sync.

## Unit tests

It's probably not feasible to get to a point where everything is tested in the new architecture, but try to ensure that you have unit tests for whatever you need to change. The book on working with legacy code is full of ideas about how to get just the part you need to work on under test.

A key idea in that book is that a unit test should run in less than a tenth of a second. Unit tests should not access databases or files. The new architecture should allow all tests (except of the BEPs themselves) to run using a memory-only backend provider. Slow tests are probably really integration tests (involving real instances of dependencies rather than some kind of stub) and may need to be reworked if they are needed as unit tests. (It may be worth keeping the integration test, too.)

## Null object pattern

Current best practice is to avoid the necessity for constant checking for nulls. One of the best ways to do this is to have methods that would otherwise return null return instead a “null object” that implements the expected interface(s) in such a way that a client can achieve the right result by calling the null object method in the same way as the method of the regular object. I’d like people to be aware of this pattern and look for ways it could help us.

## General New Skills

Refer to the document “New skills for a new architecture” for stuff I hope everyone learns to apply.

## Old code still around

Be aware that in many DLLs, especially FDO, there is a lot of code around that is not actually built. Some of it is retained to make integrations easier, some because we expect to finish porting it eventually. Code that we definitely expect to need in some form at some point is indicated using #ifdef WANTPORT; please continue this convention if you want to mark off some part of a new DLL which you choose not to port immediately. Search such code if you find yourself wanting a function that doesn’t exist and that you think might once have had.

There are also complete files around that have their compile action set to “None”, that is, they are in the project but not really live code. This is misleading, but the files are useful both for integration, and possibly because some of their functionality may eventually be needed. As I find such files, I’m trying to make the situation more obvious by inserting #ifdef FILE\_GETS\_BUILT around all the code.

## Build problems

Mostly you should be able to do “nant remakefw”, “nant mkall”, or “nant build all” in the WW branch. Once that succeeds, you can run the FLEx skeleton (FLEx.exe, the target of LexTextExe). We don’t yet have a skeleton of any of the other applications.

There are a few mysteries currently unsolved:

1.       The views tests sometimes report problems that are not repeatable on running them again (especially when run as part of remakefw).

2.       On Vista, I’ve also had test failures that are apparently spurious in mklg and mkgenlib.

3.       To pass all the FDO unit tests you need to install the Firebird server. (See section 2.3 immediately below.) Even then, I’m seeing a mysterious thing in the Resharper test runner where all the firebird tests are crossed out with a red line, though it says all the tests pass.

## Installing Firebird Server

Instructions for installing Firebird client/server is in [FW branch]\Doc\FieldWorks Setup\FirebirdClientServerSetup.html.

# FDO

The main difference between FDO in the new architecture and FDO in the old is that in the new FDO CmObjects hold all of their own state, while in the old system, the ISILDataAccess cache held it all. (The old style CmObjects did hold their HVOs, but all other data came by accessing the cache, or by accessing SqlServer for some cases.) The ISILDataAccess implementation now is on the client side of FDO and is a Façade for clients which cannot access FDO directly, either because they were coded to the old interface historically or because they need the flexibility of operating on configurable properties based on flids. The Views code is the main client that should continue to use it.

## Interfaces

### CmObject Interfaces

All CmObjects have generated interfaces for the main model-defined properties. As with the FW 6.0 system, these generated interfaces are ‘partial’ interfaces, which can be augmented by hand-written code. These interfaces are how all code outside of the FDO assembly is to interact with the implementations of the domain objects (CmObjects). It doesn’t work like this yet, of course, but that is where we want to external code to end up. Even within the FDO assembly, the only legitimate use of the implementations of the interfaces belongs in the new FDO.DomainImpl namespace. Code in the FDO assembly that is outside this namespace is to only use the interfaces. This also is the goal, but not yet the reality.

### Factory Interfaces

The new FDO has basic, generated support for a Factory interface for each concrete CmObject class in the model. These generated interfaces are ‘partial’, so they can be augmented by hand-written code for Create methods that cannot be easily generated. The basic, generated Factory interface supports one Create method with no parameters. This basic method is the replacement for the ‘new Foo()’ constructor.

Hand-written additions to factories need to be added in two places. The first place is to add a new method to the interface. This would be done in the “Foo\Src\FDO\FdoInterfaceDeclarations.cs” file, as is done:

/// <summary>

/// Additional interface methods.

/// </summary>

public partial interface ILgWritingSystemFactoryFdo

{

/// <summary>

/// Create an ILgWritingSystem.

/// </summary>

/// <param name="icuLocale">The ICU Locale</param>

/// <param name="locale">The regular locale</param>

/// <param name="rightToLeft">true for a right to left writing system, otherwise false</param>

/// <param name="defaultSerif">Name of the default Serif font</param>

/// <param name="defaultSansSerif">Name of the default sans serif font</param>

/// <returns></returns>

ILgWritingSystem Create(string icuLocale, int locale, bool rightToLeft, string defaultSerif, string defaultSansSerif);

}

The implementation additions for factories would go in the “Foo\Src\FDO\DomainImpl\FdoFactoryAdditions.cs” file.

### Repository Interfaces

The new FDO has basic, generated, support for a Repository interface for each CmObject class (concrete and abstract) in the model. These generated interfaces are ‘partial’, so they can be augmented by hand-written code. The basic Repository interface supports object access by HVO or Guid, as well as access to all instances of the class of the Repository, **plus** all of the subclasses of the class. A call to the ICmObjectRepository’s AllInstances() method will return every object in the data store. A count of all instances is included in the basic Repository interface.

In general you may use the following regular expression for converting anything using CmObject. CreateFromDBObject() type interfaces to a repository

by finding

“= {[^.]\*}.CreateFromDBObject\({[^.]\*}, ”

and replacing with

“= \2\.ServiceLocator\.GetInstance\<I\1Repository\>()\.GetObject(”

Hand-written additions to repositories need to be added in two places. The first place is to add a new method to the interface. This would be done in the “Foo\Src\FDO\FdoRepositoryInterfaceAdditions.cs”. Hand-written implementation for repository additions will go in the “Foo\Src\FDO\DomainImpl\ FdoRepositoryAdditions.cs”. (The hand-written and generated Repository implementation code probably ought to move to FDO.Interface namespace, since Repository implementations are technically Infrastructure, and the files moved, but it hasn’t been done yet.)

### FDO Vector interfaces

The new system primarily uses standard .Net  collection/list interfaces, such as IList, or ICollection, but there are additional interfaces, as well. There is a new IFdoVector interface, which adds access to the entire array of the HVOs of the contained objects (much as is done with the ToArray() method), as well as the array of object Guids in the vector. This interface is public, and any code can use it. There is an internal interface that is used to support for Undo/Redo, and persistence. The expectation is that this access will stay internal and only be used internally by ‘authorized’ code, even internally.

There are several other interfaces defined to support vectors, some public, but others internal. These use generics, which allow for type safe vectors. A main division of vectors in these interfaces support collections vs. sequences of objects. Some vectors allow duplicate objects (i.e., reference sequence properties), but many do not (i.e., owning vectors and collection properties).

One consequence of going to the standard interfaces is that ‘Add’ no longer returns the thing added. Combined with the move to factories, this means that code like this needs to be changed:

LexSense newSense = myEntry.Senses.Add(new LexSense());

Todo: finalize decision about what to replace that with and document.

### Alternative string and unicode interfaces

The string and Unicode alternative classes are supported by interfaces. The base interface is ITsMultiString, but it is the base for other interfaces, which contain the public accessors that are normally on the alternative classes.

### Internal interfaces

There are numerous internal interface additions included in the FDO assembly. For instance, there is an ICmObjectInternal interface (derives from ICmObject interface)) that adds a bunch of methods to CmObject that are only used inside the FDO assembly. There are currently a collection of internal interface additions to various factories, the methods of which are also internal to the FDO assembly. The interface additions and implementations can be found in the files mentioned, above.

## Implementations

All generated interfaces of the various types have generated partial implementation classes. The implementations are normally in the FDO.DomainImpl namespace. (Even the repository implementations are there, but they really belong in the FDO.Infrastructure namespace and will likely be moved there at some point.)

### CmObject Implementations

All of these domain object interfaces also have generated partial implementation classes. These classes can be augmented by using partial classes, as is done in the FW 6.0 code. The generated classes are in the FDO.DomainImpl namespace, which ought not be used by code that is outside of that namespace.

### Factory Implementations

Eventually (hopefully before the year is over), all CmObject creation will be handled by factories (which already exist). At some point during the year, the code generator can change all public constructors to internal, which will kind of force the issue. :-)

### Repository Implementations

The repository implementations are currently in the FDO.DomainImpl namespace, but they will likely be relocated to their correct ‘home’ in the FDO-Infrastructure namespace.

The generated interfaces and the implementations can be augmented with hand-written code in the same way the domain objects and factories can be augmented.

### FDO Vector implementations

The implementation classes for the vector classes are in the FDO.DomainImpl namespace. The expectation is that code will use the interfaces, rather than the implementation classes outside of the FDO assembly, and that a good bit of code inside it will do the same.

### FDO MultiString & MultiUnicode classes

The two main multi-foo accessor classes implement the ITsMultiString C++ interface, as well as some additional C# interfaces. The base ITsMultiString interface provides for basic ‘get/set’ operations. Additional interfaces (which derive from ITsMultiString) provide for more methods that code is used to seeing on the older multi-foo implementations.

## Statics on CmObject Implementation classes

It is not desirable for any CmObject implementation class to have any public and static properties, methods, or constants. Such code prevents the long-term goal of splitting up FDO into Bounded Contexts, since that type of code completely binds client code to the implementation classes, which is not  good.

To the end of decoupling code that should not be coupled, the new FDO system has provided new ways to access the current constants. There are new FooTag classes (also partial classes) that mirror the CmObject class hierarchy and which hold all constants. The old stuff is no longer generated.

It would also be good that by the end of the re-architecting effort that there be no public static properties or methods on any CmObject implementation class. Internal, protected, or private, yes, but not public. Eventually the whole implementation class will be internal.

## FDO & UOW interaction

The CmObjects register all data changes with the UOW. There is no need for any other code to do this. The CmObjects are not strict POCOs, I guess, since they do this registration.

There are ways to remove this to make CmObjects true POCOs, but the price could be lower performance at Commit time. The Commit code would have to be able to tell which objects in the IdentityMap were new, deleted, or modified. It can be done, of course, but it may not be cheap.

A probable way to make CmObjects true POCOs would have the Factories add the new kid to the UOW. Repositories could be pressed into service for deletions with a Remove method. The really hard part would be the modified objects. There would need to be a snapshot of some sort at load time, which is compared to the current state at save time, and those with differences would be saved as modified. The Undo/Redo system would have to be revised quite a bit to support this kind of world, as well.

## FDO & Persistence Code Interaction

There is no access between the domain objects and the persistence code. Nor, should there ever be such a need.

## FdoCache

I believe that the FdoCache class will be removed at some point. In the 6.0 system, it served many disconnected functions. For instance, it provided C#-friendly access to the metadata cache as well as the actual cache. These C# methods can easily be added to the base C++ interface, so are not needed on FdoCache. (Indeed, many of them have already been removed.)

## Service Locator and Dependency Injection

The FdoCache provides for access to a connection-specific Service Locator instance, which, in turn, provides for access to interfaces for factories, repositories, metadata cache, and the ISILDataAccess interface, among other things. To the extent that we are able to use Dependency Injection within FDO itself, then use of the Service Locator will diminish.

We’ll gradually build up a list of what services one can count on finding in the service locator. I think we will probably gradually give it specialized methods for common ones. For test code, we will not want to pay the penalty of creating all such objects in advance, but on the other hand, we don’t want tests to break because some method that the test is only using coincidentally needs a service that it previously did not. I think the solution may be a default test-only service locator which knows how to create a minimal default instance of any of the standard interfaces on demand.

One primary idea of DI is that if a class depends on some other class, it should be at least possible for client code to provide (“inject”) the dependency, especially in order to use a test stub. One way to do this is with additional constructors which take the dependency objects as arguments, while the default constructors just create the usual dependency object. This gets tricky over lots of layers, which is why we have the service locator for things that are very widely used.

## HVOs

The HVOs in the new system are honest handles in the new system. They are only valid for that one session. One should never store them in a configuration file and expect them to be valid the next session. The object’s Guid is always unique and can be used form one session to the next to get at the same object.

The HVOs are not stored in any of the data stores, nor is there any reason to do so in the future, given that they are now honest, temporary, handles.

## Back references

Back references were declared last fall to not automatically be a natural part of the model of the objects referred to; rather, the primary way to access them is through the Repository for the referring class. That is, instead of, say, ICmPossibility having a “property” RefsFromStatusOfLexSense, the LexSenseRepository has (read: “will have”) a method SensesWithStatus(ICmPossibility status).

There are cases where such associations are bi-directional, and these associations need to be supported within FDO when found. By saying something is “bidirectional,” we don’t mean that there is (as yet) a way to just set a flag and make both directions available; rather, a “bidirectional” relationship is something which in model terms makes sense as a property of objects at both ends.

For example, the segments of a text paragraph is a very natural property of a paragraph, though the actual link in the model is the InstanceOf property of a CmBaseAnnotation. (Actually it’s such a natural property that we might even change the model to make it the primary one, but that’s another story.) There is no automatically-generated way to get the segments of a paragraph, but the repository for CmBaseAnnotation can provide the annotations which have that paragraph as their InstanceOf, and from these the ones with the right annotation type can be selected and ordered by their BeginOffset. Code to do this should be written as a property of IStTxtPara, and an attribute set (as described below) to indicate that this method should be used as a “virtual property” visible to views. Ideally code should also be written to keep this virtual property up to date and notify views of changes as relevant properties of CmBaseAnnotations change.

On the other hand, while it makes perfectly good sense to retrieve from the LexSenseRepository the senses which have a particular status, “Senses which have this status” does not make sense as a property of CmPossibility. (It’s a fundamental concept in our model that paragraphs have segments, relevant to almost every paragraph, but not that list items have “senses which have this status”; indeed, the latter is utterly meaningless for any list item that isn’t in the Status possibility list.)

Occasionally, a particular view may want to display a back-reference property that doesn’t “make sense” as a virtual property of the object. For example, we might want a view that shows the various items in the Status list and (for each one) the senses that have that status. We do not want to make an unnatural virtual property in our main domain model (and maintain it for every CmPossibility) just to handle this. FDO exists to support the domain model, and not such ‘extras’. The suggested way to provide support for such one-off relationships is to use decorators for the MDC and the SDA, thus giving CmPossibility a ‘property’ that exists for the one view but not generally as part of the domain model. This would leave FDO out of the loop, which makes sense for properties which are not a natural part the domain model.

Between cases where the back reference property is such a natural part of the model that it might well have been the primary property and cases where it would make no sense at all there are probably a lot of cases where the answer is much less obvious. We’ll have to consider these as we come to them. A good clue is whether you can think of a straightforward name (like “Segments”) for the backref property; if “RefsFromStatusOfSense” is the best that can be done, it’s probably not a natural property!

## FDO Namespaces

The main FDO namespace is SIL.FieldWorks.FDO (shortened to FDO for purposes of discussion here). Files in the root folder should all be in this namespace (although not all files that are currently there will survive for long there). The basic folder structure seen in the FDO project file maps to additional namespaces. (Resharper helps encourage this, as you may have observed.) The lower level namespaces are:

FDO.Application

FDO.Application.Impl

FDO.DomainImpl

FDO.DomainServices

FDO.Infrastructure

FDO.Infrastructure. Impl

(There are other folders in Perforce under the FDO folder, but those are not part of the FDO assembly.)

### SIL.FieldWorks.FDO

This namespace holds the interface declarations for the domain objects, repositories, and factories, as well as a lot of old (and in some cases obsolete and not compiled code files) legacy code, which may not really belong in this namespace. For instance, FdoCache is probably not really needed at this point, but it is hard to get rid of right now. Other code files were easier to leave in this namespace, as long as significant integrations were being made. As the integrations dwindle, it would be good to review what is left in this namespace to make sure it really belongs there.

This namespace should be the most widely used of all the FDO namespaces, outside of the FDO assembly.

### SIL.FieldWorks.FDO.Application

This namespace holds the ISILDataAccessManaged interface, which extends the regular ISILDataAccess interface. It is a Façade because it provides an alternative API for clients such as the views code that can’t use FDO objects directly. Clients that find it easier to not use FDO directly, because they are more general in nature are also good candidates for using the Facade.

The name of the implementation class was selected some time back to be more helpful in knowing what the class did, than the interface name is. (One can argue how well the new name achieves that goal, I suppose.)

This namespace also holds a decorator base class of the SDA implementation. Clients who need to create a decorator will need to use this namespace to get at that abstract base class, but I’m not sure many other external (to the FDO assembly) clients will need to use it.

### SIL.FieldWorks.FDO.Application.Impl

This namespace holds the implementation of the ISILDataAccessManaged interface. Right now, nothing else is in it, but other classes can be added here, as needed.

[NB: Unless you are working with FDO’s IOC code, you should never include this namespace in a ‘using’ statement.]

### SIL.FieldWorks.FDO.DomainImpl

This namespace contains the implementations of the main domain object interfaces, as well as some implementations of other interfaces, such as those that support the vector collections and multistring classes. There are a few directly implemented classes with no interfaces, some being public. The idea is that no external FDO client code is to use this namespace or anything in it, but we aren’t fully there yet. No external client code can access the domain object implementations classes, or create them via Constructors, since that is all internal now. The Repository implementations have now been moved into their proper home in the “SIL.FieldWorks.FDO.Infrastructure. Impl” namespace.

[NB: Unless you are working with FDO’s IOC code, you should never include this namespace in a ‘using’ statement.]

### SIL.FieldWorks.FDO.DomainServices

This namespace is fairly new. The idea for it is that code that provides domain-aware services goes into this namespace. Right now, it holds all manner of static code that once lived on CmObject implementation classes, which had to move elsewhere to make those classes internal. Some of that code may well belong elsewhere, but for now it resides here to get it off the domain classes.

### SIL.FieldWorks.FDO.Infrastructure

This namespace holds interfaces and code that are outside of the domain layer, but not in the higher layers, such as application or UI. Code in this namespace would be the UOW, IdentityMap, Undo/Redo, and the interfaces for the BEPs, along with other code.

The expectation is that FDO will make use of interfaces or code in this namespace, as will external clients, where such access if appropriate. (Some code is internal, so is not available to external clients.)

### SIL.FieldWorks.FDO.Infrastructure. Impl

This namespace holds the implementations of the various BEPs, as well as implementations of several other interfaces. It also holds some classes that have no interface, such as the IdentityMap. These might need an interface in order to really hide the implementations, even in the FDO assembly.

Once a BEP is selected for use in FW 7.0, the unused ones can be ‘mothballed’. This could be taken clear out of the system, or simply disabled. Project references to database dlls that are not shipped can also be removed from the FDO project.

[NB: Unless you are working with FDO’s IOC code, you should never include this namespace in a ‘using’ statement.]

# Supporting Infrastructure

This section covers selected infrastructure code, not all of it. The parts that are covered are particularly interesting, since they relate a good deal to the new FDO domain layer.

## Persistence

A key component in the new architecture is the persistence mechanism. In the past, access to SqlServer was completely open to any code anywhere. In the new architecture, this is not the case. Appropriate public access is allowed, but to a very limited extent. Three new interfaces (and implementations of them) in the new architecture manage this access. One of the new interfaces is public and is available to all client code. The other two interfaces are internal and not available to code outside of the FDO assembly. All three of the interfaces are implemented by one C# class in the FDO assembly.

1.       IDataSetup interface

This interface is available to any C# code. The idea is that this interface will expose behaviors, such as Open/Close/Backup/Restore/Delete/etc. on the user’s data. The interface also supports importing and exporting from one supported data store system to another. One can import the converted XML from FW 6.0 into any of the supported storage systems. One can also export all of the data from one supported system to another.

There are several methods defined by this interface, but the expectation is that more will be added, as needed, to support the end-user’s tasks.

There is one method on this interface that allows client code to ‘eager load’ whatever objects it wants. The design allows for any number of definitions of bundles of objects to be reconstituted.

2.       IDataReader interface

This interface is ‘internal’, which means that code outside of the FDO assembly cannot get to it to use its methods. The expectation is that no more methods need to be added to this interface, than are already defined on it.

This interface provides for the “R” in the basic CRUD database operations.

This interface essentially is a Repository, since it operates on Aggregate Roots, and has direct access to the data store implementation. It is essentially an implementation of the Strategy pattern, in that the implementation is injected (Dependency Injection) into the publically accessible Repositories of each CmObject class.

No code other than the main Repositories in the FDO assembly is authorized to use the IDataReader interface. There is no way to prevent its use, but it isn’t authorized, so should not be done.

3.       IDataStorer interface

This interface is ‘internal’, which means that code outside of the FDO assembly cannot get to it to use its methods. There is one ‘Commit” method defined on this interface, and no other ones need be added.

This interface provides for the “C”, “U”, and “D” in the basic CRUD database operations.

This interface provides a way for the UOW management system to persist the actual data changes (create/delete/update).

No code other than the UOW management system in the FDO assembly is authorized to use the IDataStorer interface. Even for the UOW system, only the “Commit” method is authorized to use this interface There is no way to prevent its use, but it isn’t authorized, so should not be done.

One internal, abstract, base class (FDOBackendProvider) implements all three of these interfaces. There is one internal, sealed, concrete subclass of FDOBackendProvider for each of the supported storage systems, as well as one subclass that is used for memory-only ‘data store’ usages, such as unit tests. Currently, there are four supported data store systems, as well as the memory-only system (Berkeley DB, Db4o, Firebird, and XML).

All of these systems work with the XML representation of each CmObject instance. Other data storage systems can be added, as needed. If these new systems read/write these XML packets, they can also derive from FDOBackendProvider. If they wish to use some other storage system, such as an NHibernate+regular RDBMS, then there will need to be some refactoring of FDOBackendProvider to support that kind of persistence mechanism.

### Creating a new BEP

There could be two kinds of new BEPs developed. One would continue the xml key/value type BEP. The other could be anything else, such as an NHibernate-based RDBMS BEP.

#### Common implementation steps

1.       Add basic support for the new underlying data store. For example, I added the Db4o dll to the DistFiles folder, and added a reference to it in from the FDO assembly. For say, Firebird client-server, this would be an installation for Firebird. (Firebird embedded is not installed, and the required files are just lying around in DistFiles ready to be used.)

2.       Add a new item to the FDOBackendProviderType enum for that BEP. (Do not commandeer any current BEP code or enums. Make new ones.)

3.       Add one or more C# files to the FDO project to access the new data store. The file(s) go in the Src\FDO\Infrastructure\PersistenceImpl folder.

4.       Add a new case statement to "switch (m\_backendProviderType)" in the FDO\DomainImpl\FdoServiceLocatorFactory.cs file to create the new BEP when selected for use.

5.       Add a new unit test class. (See 3.1.1.1.1 Unit testing BEPs for details.)

You can look over the various BEP implementation classes in the FDO\Infrastructure\PersistenceImpl folder to get an idea of what needs to be put into the new BEP file to get it to pass the BEP tests.

I've probably left something out, but the above stuff ought to get you a good ways down the road to adding a new BEP.

#### XML-based key/value BEPs

This kind of BEP would be the easiest, since it can inherit a lot of common behavior from the FDOBackendProvider base class. Subclasses *must* implement two protected abstract methods and really should override the Commit method (for systems that actually store data). They may override the Dispose method, if they have anything to close down. There may be other code to add, but that would be private to that BEP.

##### Firebird sample

Using the current embedded Firebird BEP as an example (since there may be other BEPs added, such as MySql or Postgres), we see the following.

The CreateInternal method implementation would include one or two queries that create the database with one table in it. The firebird BEP has no explicit DB creation command (as a string), as creation of the file in the right way is all that is done. The one table is created with this query string:

cmd.CommandText = "CREATE TABLE MAIN (ID INTEGER NOT NULL PRIMARY KEY,

XMLDATA BLOB SUB\_TYPE TEXT)";

The StartupInternal implementation has one query in it, along with some other C# code. That query is:

cmd.CommandText = "SELECT \* FROM MAIN";

All rows are processed to get the xml, which is then used to create CmObjectSurrogate instances for each row. The cost to fetch data for an out-of-process client-server system is mitigated, since there is only one round trip across the process boundary. (Indeed, the now removed SqlServer BEP ran circles around everyone else with this one query, even though it was cross-process, and everyone else was not.)

The Commit method would have only three queries in it: Insert, Update, and Delete. The data is fed to the Commit method in such a way that the implementation knows right off, which kind of operation is needed for each object in gets.

#### Non-xml-based key/value BEPs

This could be something like a regular RDBMS, with its various tables and columns, such as the current FW 6.0 SqlServer system. There could be other types, as well, such as an xml-based DB. There are none of these at the moment, but if one were to be added, it would require refactoring the FDOBackendProvider base class to support it. The common xml key/value code would then probably move to a new subclass of FDOBackendProvider, and the current BEPs would derive from that new class. Truly common code could stay on FDOBackendProvider. If there were no such common code, then the new BEP would have to implement all three of the interfaces (IDataSetup , IDataReader,and IDataStorer), and see that it was injected properly.

### Unit Testing a new BEP

It should be fairly easy to do unit tests on a new BEP, since there are numerous unit tests already defined, and which are used for testing all current BEPs.

#### Basic BEP Testing

Basically, a new test class is created for the new BEP, and it derives from one of two base classes to inherit its unit tests. The current class structure is:

PersistingBackendProviderTestBase : FdoTestBase (Defines common unit tests for all BEPs.)

                DB4oTests : PersistingBackendProviderTestBase (Embedded DB4o)

                FirebirdEmbeddedTests : PersistingBackendProviderTestBase (Embedded Firebird)

                BerkeleyDBTests : PersistingBackendProviderTestBase (Berkeley DB)

                XMLTests : PersistingBackendProviderTestBase (XML)

                MemoryOnlyTests : PersistingBackendProviderTestBase (In-memory DB)

                PersistingClientServerBackendProviderTestBase : PersistingBackendProviderTestBase  
                                (Defines additional unit test for client-server BEPS.)

                        FirebirdClientServerTests : PersistingClientServerBackendProviderTestBase  
                                    (Client-Server Firebird)

All Client-Server BEPs ***must*** derive from PersistingClientServerBackendProviderTestBase and override one method, which is in the form:

protected override FdoCache CreateCache()  
{  
                const string filename = "TestLangProj.fdb";  
                if (!m\_internalRestart && File.Exists(filename))  
                                File.Delete(filename);  
                return BootstrapSystem(FDOBackendProviderType.kFirebird, new object[] { filename },   
                                                m\_loadType);  
}  
  
Everything else is handled by the base test class.

[NB: the filename must be (read: it is imperative that) unique to all other BEP DB names. The test suite deletes the files when the test fixture is setup, and creates them *ex-nihilo* before the tests are run. To distinguish the embedded vs. client-server Firebird files, I prepend ‘cs’ to the file extension for the client-server Firebird file.]

#### BEP ‘Port’ Testing

Besides the above basic testing, the BEP test framework also provides a way to ‘port’ data from each BEP to all other BEPs, including the BEP of the same type. These tests are included in the “BEPPortTests” class (derives from the .Net Object class). There are only two test methods, and those are the only ones expected on this class, as they test the two ways one can port from one BEP to another (i.e., source BEP previously opened, and source BEP previously closed).

All new BEPs *must* participate in this port test framework, and it is very easy to do so. One just needs to add two new instances of a test helper class (BackendStartupParameter) to two lists, and ensure each BackendStartupParameter is set to work for the respective BEP. Samples of this for the two Firebird BEPs are:

                new BackendStartupParameter(FDOBackendProviderType.kFirebirdEmbedded, BackendBulkLoadDomain.All,

                           new object[] { "TLP.fdb" }),

                new BackendStartupParameter(FDOBackendProviderType.kFirebirdClientServer, BackendBulkLoadDomain.All,

                           new object[] { Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().CodeBase.Substring(8)),

                                "TLP.csfdb") }),

and

                new BackendStartupParameter(FDOBackendProviderType.kFirebirdEmbedded, BackendBulkLoadDomain.All,

                           new object[] { "TLP\_New.fdb" }),

                new BackendStartupParameter(FDOBackendProviderType.kFirebirdClientServer, BackendBulkLoadDomain.All,

                           new object[] { Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().CodeBase.Substring(8)),

                                "TLP\_New.csfdb") }),

[NB: the filename must be (read: it is imperative that) unique to all other BEP DB names. The test suite deletes the files when the test fixture is setup, and creates them *ex-nihilo* before the tests are run. To distinguish the embedded vs. client-server Firebird files, I prepend ‘cs’ to the file extension for the client-server Firebird file.]

## Unit Of Work

Any code that initiates data changes must (read: it is imperative that) start a UOW and end the UOW, when the change is complete. Failure to do this will result in an exception being thrown by the UOW management system. The way to start a UOW is to call either of these two methods on the ISilDataAccess interface implementation: BeginUndoTask or BeginNonUndoableTask. The UOW is ended by calling the corresponding method: EndUndoTask or EndNonUndoableTask.

The UOW system is set up as a Finite State Machine (FSM), which means that certain activities are supported when the FSM is in certain states, but not when it is other states. For instance, data changes may be not be done, before starting a UOW, as the ‘waiting’ state does not allow such changes. Once the UOW has been started, the state is changed to allow for one or more data changes (create/delete/modify) for one or more CmObject instances. During this state, no BeginUndoTask class can be made (i.e., currently, there is no support for nested UOWs). Likewise, there can be no calls to Commit the UOW, nor can there be attempts to broadcast the data changes to say the display code.

When the EndUndoTask or the EndNonUndoableTask methods are called, the state again changes to a state where the data changes are broadcast to clients such as the display. This is done by calling the PropChanged method on all registered IVwNotifyChange. No additional data changes are allowed in this state, and any attempt to do so, will result in an exception being thrown.

I don’t yet have a distinct state set up for the Commit phase of the UOW’s life, but I’m thinking about adding one. For now, a commit cannot be done, except after the UOW is complete (i.e., after the EndUndoTask method is called. Commit is automatically done at the end of the EndNonUndoableTask.

There is no access to an actual UOW object other than by the UOW management class. The CmObjects themselves do not have this access, but they work through the manager class to register their comings, goings, and modifications. When such registration is done, one of several special IUndoAction classes is created and automatically added to the UOW’s collection of IUndoActions.

‘Outsiders’ can add their own IUndoActions, if they wish, but those should relate to issues that are not actual data changes. For instance, one could add a special IUndoaction class to mange undo/redo of a selection, or some other application level action.

There is no formal public access to the UOW manager class, even though it currently is exposed via the IActionHandler interface it implements. Casting that manager class to its C# class cannot be prevented, but the UOW support methods are internal, so there is no reason to do the cast. Indeed, the current support methods are going to change, once the design is resolved on the ‘events/side effects’ matter.

There are two UOW helper classes that are designed to aide client code in ensuring a UOW is started and ended properly: UndoableUnitOfWorkHelper and NonUndoableUnitOfWorkHelper. These helper classes will make the pair of calls to the respective Begin and End task methods. Both of these are disposable, so the preferred way to use them is in ‘using’ code. By default, both of these classes call Rollback on the UOW, if there is an unhandled exception of any kind, during the UOW processing. (This ensures the “A” in the ACID property all transactions are to support.) The last thing in the ‘using’ block should be:

Helper.RollBack = false;

The Dispose method checks that Rollback Boolean. If it is ‘true’ (default value), then a Rollback is done. If it is ‘false’, then the respective End task method is called.

At the moment, the data is not persisted at the end of a UOW. The design for that is still pending.

Implications:

Code wrapped in “using new UndoTaskHelper” at least needs to change to use UndoableUnitOfWorkHelper and to set Helper.Rollback = false at the end of all success paths.

If such code is also called as part of a larger UOW, it needs to be split into two methods: the core task (code inside the “using” block), which can be shared with the larger task (and independently tested!), and the original method which implements a stand-alone task, and typically does nothing but wrap the core method in a UOW.

If such code is creating a custom UndoAction to more efficiently record the changes, that behavior should go away. Hopefully the new code is enough more efficient to make it unnecessary. If that proves not to be the case, we may have to re-introduce this capability in selected cases.

Likewise, any direct calls to PropChanged should go away. These are common after creating objects, since the old code did not do this automatically. (The reason for this was to prevent a PropChanged inserting the new object before it is initialized. The new UOW will delay the automatic propchanged until the UOW is complete, and also suppress PropChanged calls for properties of new objects, since they can’t be previously displayed anywhere.)

If a data change is being made in response to PropChanged, this should now instead be done by implementing or modifying  XChangedSideEffects methods for the relevant classes.

## IdentityMap

The IdentityMap pattern is used in the new architecture to ensure that only one instance of any given CmObject is in memory for any given DB connection. The implementation class of the pattern is called IdentityMap, of course. This class is internal and sealed. Since it is sealed, there are to be no subclasses. Since it is internal, no code outside of the FDO assembly has access to it. Most authorized users are in the persistence/repository systems and the UOW system. There are a couple of other users, but they could be refactored to not use it.

## Undo/Redo

Undo & Redo are pretty much normal. About the only difference is that when a non-undoable UOW is started, then all prior UOWs are persisted, and the undo & redo stacks are cleared. A non-undoable UOW is then persisted automatically at its end. The user cannot undo it, of course.

I covered it in the UOW section, but it bears restating here, as well. Client code can add IUndoAction instances, and these will be included in Undo and Redo calls. These IUndoActions should not be for actual data changes, since those are all handled by FDO. They could be for a case of a selection being undone/redone, for example.

## Metadata Cache (IFwMetaDataCacheManaged : IFwMetaDataCache)

The main metadata cache (MDC)) implements the new IFwMetaDataCacheManaged interface, which is a C# ‘extended’ version of the base C++ IFwMetaDataCache interface. This extended interface includes everything from the base interface, plus some .Net-friendly methods that avoid some of the more unpleasant C++ method calls. Take this extension method, for example:

uint[] GetFields(uint clid, bool includeSuperclasses, int fieldTypes)

This extension method avoids having to use the AttrayPtr and asking twice for the fields.

The extended interface is available publicly, but the C++ code that expects to get the original interface is also happy.

NB: The MDC currently uses uints, but FWR-32 calls for them to be changed to ints for both the flids and clsids by the RC1 version of the FW 7.0.

## DomainDataByFlid (ISILDataAccessManaged : ISILDataAccess)

This class is not an actual cache as is found in FW 6.0. Rather, it is a Façade over the domain model’s CmObjects. The primary client is the Views code, but other code can profitably use it as well. Any code that works with objects and only knows about the object’s HVOs, guids, or flids is a good candidate for using the interface.

A fair bit of the current client code that uses the ISILDataAccess interface could better use the CmObjects, and not continue to use the interface. The section goes into more details on how to convert form the interface to the FDO objects.

Please note that the ISILDataAccess interface has also been extended, as was done for the metadata cache interface. The reasons for extending this interface are the same as for the other one: to be more .Net friendly. A sample (actually, the only one, so far) extended method is:

int[] VecProp(int hvo, int tag);

## Decorators (Metadata cache & DomainDataByFlid)

The new system provides a way to extend the support for the model properties (model-defined or virtual) that is found on the standard implementations of the two interfaces: IFwMetaDataCacheManaged and ISILDataAccessManaged. There is an abstract decorator base class for each of these two interfaces, which has most methods implemented to simply pass through to the main FDO interface implementation classes. Classes derived from these abstract base classes can then intercept select method calls to work with classes and properties that are not supported by the domain layer and its model. The overrides can then pass on to calls the main implementations, if the ‘stuff’ is not of interest.

A data access decorator would be used by the views code, for example. A meta data cache decorator would then be used by the data access decorator, since the fake hvos and fluids they both use are closely related.

These decorators should be used in situations where in the FW 6.0 system one would add certain kinds of virtual properties to the FDO classes.

For instance, there is a checkbox on each row in some browse views in Flex. In FW 6.0 that was implemented as an add-on model property and smarts were added to support that. The downside of this is that the browse views were not independent, so all browse views showed the same items selected. By using decorators to intercept those UI-related properties, such views can independently show the same object as checked and unchecked. Whether some property is currently available to show in multiple windows, or not, these non-model properties are the best way to extend the model with properties that are only interesting to one client.

Note that, because such properties exist only in the decorator, they have meaning only in the one view (or just possibly a couple of views) that use that decorator as their DataAccess. Changes to them should not be broadcast to all views, and in fact (since the properties are not known to the UOW, which is now responsible for all such broadcasts) there is no way to do so. Therefore, changes to them should not be notified by calling PropChanged on the main ISilDataAccess façade; indeed, that method is now unsupported (throws NotSupportedException) and will eventually be removed. Instead, send the notification of a change to a decorator property directly to the views it serves. (Of course, if you want to build your own mechanism in the decorator to keep track of the views that it serves and notify all of them when a decorator property changes, that’s fine. Just don’t call PropChanged on the main ISilDataAccess you are decorating.)

The way this might work in practice is there would be a pair of decorators (MDC and SDA), although one might be able to get by without the MDC decorator, as long as nobody tried to ask the main MDC for information about the ‘private’ properties. The SDA client code (i.e., views code) would tell its SDA (the decorator SDA) about some data change. That SDA could actually cache its private property information, of calculate it, on the fly, whatever it deemed best to do. (Indeed, it could even use the IVwCacheDa code to stuff information into its own cache.) That SDA could then fire off the PropChanged calls to clients it knew needed to know about the changes to its private properties. The main SDA façade and the UOW need never know about it at all.

Another big use case for decorators is in filters. Assuming the view is showing some made up ‘property’ of filtered objects; the decorators can show different filtered sets of data in separate windows.

I think now that back references are not generally part of the object model, another big user of decorators will be for the back ref properties. The ISILDataAccess decorator can then fetch the desired objects from a Repository

## Repositories

Repositories are interesting in that the interfaces for them are defined within the domain layer. Their implementations, however, are considered to be in the infrastructure layer, since they connect with the data store and talk to it. The Repositories are not currently in the Infrastructure namespace, but that is where they really belong. I suppose I ought to fix that pretty soon, no? :-)

# Upgrading Issues

This section will cover issues developers will run into trying to use the new FDO system, as well as upgrading other assemblies, during the re-architecture effort.

## Applications

I suggest that each application have a bare-bones minimal version that shows nothing, perhaps not even menus or toolbars. I know Flex is already in this condition, and that some efforts have been made to get TE to start. When the minimalist feature set is defined for each application, that can be made to work in the new architecture, and the rest of the code left out of the build, or defined out using #if WANTPORT. After that is working, that near braindead version can be sent off to India and tested.

Then, another ‘version’ and feature set can be defined, and the code made to get it to work. When it works, it can also be made available for testing. This cycle can be repeated, until each application is working fully. The key idea is to get code in the various assemblies to work for that limited feature set, and block out other code, temporarily.

I also suggest that when considering what to include in the very early ‘versions’ of each application that potential blocker-type cases be done early. That way, one can bail out of the rearchitecture effort on some unforeseen fatal issue.

### Flex

The ‘early’ version of Flex will include these features: x, y, & z.

### TE

The ‘early’ version of TE will include these features: x, y, & z.

### Data Notebook

Since this program is all written in C++, it will have to redone in C#. It will be redone using Flex technology, and reside in Flex. Some features may be modified or even omitted, as long as data is not lost.

The ‘early’ version of Data Notebook will include these features: x, y, & z.

### List Editor

Since this program is all written in C++, it will have to redone in C#. It will be redone using Flex technology, and each application will have its own list editor area(s) to edit the lists it cares about. Most of this already exists, but there are a few lists and some list-editor behaviors, particularly related to creating new custom lists, that are not yet supported.

The ‘early’ version of List Editor will include these features: x, y, & z.

## Common Upgrade issues

### ISILDataAccess vs. CmObject use

Since the old data cache (ISILDataAccess, or SDA) is no longer a cache, but rather a Façade over FDO, code that currently accesses the data via the SDA needs thought about how to treat it. Code, such as the Views code, or FXT, that has no knowledge of the object and properties it is working with, and must, therefore, be very general, ought to continue to use the SDA interface to access the data. In cases where the original code does know what objects it has and the desired properties it is interested in, but has merely used SDA to not have to create a CmObject instance, probably ought to switch to use the CmObject. It will be faster to use the CmObject. If one only has the HVO of the interesting object, it can be fetched from the relevant Repository and used. Or, perhaps the code can be revised to pass the CmObject around, and not it’s HVO. Upgrading code to pass, and use, the CmObject will be better in the end, as client code need not fetch it from a Repository.

## FDO code

### Repositories

Client code that used the constructor with the FdoCache and an hvo int *must* now use another mechanism to get an extant CmObject. One good option is the Repository for the class of object that is sought. If one does not know what its class is, then use the ICmObjectRepository.

### Factories

Client code should be switched to use the new Factories, rather than continue to use constructors.

### Obsolete namespaces in FDO

The old namespaces that corresponded to the FDO modules have been removed. Files that still have them in ‘using’ will need to remove them.

### FdoCache

Many of the old methods on FdoCache have been removed, since they belonged elsewhere. One needs to look ‘elsewhere’ in the proper home for these methods. For instance, the various ‘helper’ methods for the metadata cache (MDC) have been removed. One can just use the MDC instead, which now has some C#-friendly methods.

To get at the MDC, LangProject, or other big items, one can continue toi use FdoCache to get at them. One can get them from the Service Locator, as well, which is what the FdoCache does now anyway.

### FDO Vector classes

One big difference between the FW 6.0 vector classes and the new classes is that the new one does not support code such as:

var newFoo = bar.collProp.Add(new Foo());

The code to do this in the new system would be:

var newFoo = fooFactory.Create();  
bar.collProp.Add(newFoo);

This is because the new system uses the standard .Net collection and list interfaces, which do not return what was added.

There is also no Append(foo) method, since that is not part of the .Net interfaces. Add(foo) should be used rather than Append(foo).

The old HvoArray property needs to be redone to use the ToHvoArray() method. This was done to keep the symmetry with the ToArray() method.

The ‘FirstItem’ property of the vectors is not implemented in the new system. One can use [0] to get the first object in a sequence, and collections have no ordering, so getting the ‘first’ one is meaningless. One can also use LINQ-Objects to do this with its ‘FirstOrDefault’ mechanism, but that still won’t give any meaning to the first object in an unordered collection.

### FDO MultiString & MultiUnicode classes

For upgrading extant code, one can remove calls to now obsolete methods or properties, such as “UnderlyingTsString” for the multi-unicode string collections. Both MultiString and MultiUnicode classes return only the underlying ITsString. In some cases that assumed the C# string was being returned, one needs to add “.Text” to get it from the returned ITsString.

### Virtual Property Handler class & related interface

The old FW 6.0 IVwVirtualhandler interface and all of its implementation classes are obsolete and not to be kept in the new system. The new system handles this differently.

For ‘virtual’ properties that need to be accessed by the ISILDataAccess Façade, one needs to add a special attribute to the property. This new attribute is of the class “VirtualPropertyAttribute”, which is a subclass of the .Net Attribute class. Properties marked with this attribute are added to the metadata cache. If one need not access a property with the SDA, then this attribute ought not be used. That is a simple property of the sort that can be on any .Net object.

Here are some typical steps (the example is the virtual property ILexSense.LexSenseOutline).

1. Add an appropriate method to the interface, e.g., in ILexSense partial class in FDO/FdoInterfaceAdditions.cs add

ITsString LexSenseOutline { get; }

1. Add an implementation, e.g., in class LexSense of FDO/DomainImpl/OverridesLing\_Lex.cs

public ITsString LexSenseOutline

{

get

{

string outline = m\_cache.GetOutlineNumber(this, LexSenseTags.kflidSenses, false, true);

return m\_cache.MakeUserTss(outline);

}

}

1. Mark it as a virtual property, e.g., right before the method put

[VirtualProperty(CellarModuleDefns.kcptString)]

(If it's an object property it should have as a second argument a signature, a string giving the unqualified class name, e.g., “LexEntry”.)

1. Implement or modify side effect methods to send notifications if something changes that would affect it,e.g.,

protected override void AddObjectSideEffectsInternal(AddObjectEventArgs e)

{

if (e.Flid == LexSenseTags.kflidSenses)

{

// The virtual property LexSenseOutline may be changed for the inserted

// sense and all its following senses and their subsenses.

SensesChangedPosition(e.Index);

}

base.AddObjectSideEffectsInternal(e);

}

After a few intermediates, this calls

internal void LexSenseOutlineChanged()

{

uint flid = m\_cache.MetaDataCache.GetFieldId2(LexSenseTags.kClassId,

"LexSenseOutline", false);

ITsString tssOutline = LexSenseOutline;

// We can't get a true old value, but a string the same length with different

// characters should cause the appropriate display updating. Pathologically,

// the old value might differ in length; if that causes a problem at some point,

// we'll have to deal with it.

ITsStrBldr bldr = tssOutline.GetBldr();

StringBuilder sb = new StringBuilder(bldr.Length);

sb.Append(' ', bldr.Length);

bldr.Replace(0, bldr.Length, sb.ToString(), null);

((FdoMediator)m\_cache.ServiceLocator.GetInstance<IActionHandler>()).

RegisterVirtualAsModified(this, flid, bldr.GetString(), tssOutline);

foreach (LexSense sense in SensesOS)

sense.LexSenseOutlineChanged();

}

Note the use of the MetaDataCache to get the appropriate flid, and the call to RegisterVirtualAsModified; this eventually causes the PropChanged.

With virtual properties it is often difficult to obtain an accurate old value. For string properties the old value is not critical. For object properties, if you are not entirely sure of the correct old value pass an empty collection; this may be less efficient but should always give the right display. (But will Undo?)

You may need to make new overloads of RegisterVirtualAsModified, only simple TsString properties are supported as yet. In general each RegisterObjectAsModified method (there is one for each type of property) may eventually need a corresponding overload of RegisterVirtualAsModified; this will typically create a new subclass of the class which RegisterObjectAsModified creates, and passing it to RegisterCommon. The subclass will override Undo and Redo to do nothing, and IsDataChange to answer false.

### Obsolete methods

There are a variety of obsolete methods scattered around in the FDO assembly. They should not be re-implemented, but rather the proper method should be used.

One method that need not be updated is the “IsRealObject” method of FdoCache. All CmObjects are real, even if they have been deleted, or haven’t been properly set up with valid Hvos yet. Calls to the old IsRealObject method will have to be removed, since the code won’t compile with it remaining.

Should some be changed to IsValidObject or something similar?

### Instance methods not found on some CmObject class

Some public methods (not static methods) have not yet been included in the interface, or perhaps they were in the interfaces, but have simply not yet been ported yet.

One ought to consider if the method actually belongs on that class, or if it might really belong elsewhere, before adding it to the interface. For instance, a lot of previous static methods related to getting writing systems are now living in the ILgWritingSystemRepository.

### IVwCacheDa interface

The IVwCacheDa interface is not supported in FDO. It was used in FW 6.0 to add data to the SDA cache, which avoided adding the data to the DB. FDO has no need of the interface, since it handles all legitimate domain data. Any data that might profitably go into this cache cannot be domain data by definition. Clients that really insist on having a place to stuff such data would be wise to use some C++ implementation that was disassociated from persistence altogether, rather than try and add the capability to the FDO system. Using the interface for that use case is perfectly fine.

### ScrFdo

This assembly has been brought back into the main FDO assembly, in theory. In practice, however, very little of the ScrFdo code has actually been moved back at this point. Until the integrations from the trunk die down to a trickle, it will be quite hard to ensure that the code in the old ScrFdo assembly has been properly integrated into the main FDO assembly. I recommend that when the ScrFdo code is finally moved into the FDO assembly, that the old code be deleted in the original file, when the port is finished. Then, it will be easy to spot what has yet to be ported.

### Porting assemblies

In general, I suggest assemblies be upgraded, to support the ‘early’ versions of the apps, but code not needed yet, be left ‘as is’ or if it fails to compile, then wrapped with #if WANTPORT #endif, until such time as it is really needed and can be dealt with properly. Likewise, unit tests can be disabled or wrapped the same way, until the code they test is needed.

### Unit tests

All unit tests that need FDO should derive from the base class “MemoryOnlyBackendProviderTestBase”. This base class is set to use the memory only ‘data store’. Besides being faster than using a real data store system, one need not fret about ensuring the data is correctly restored after the tests are run, as is required with the FW 6.0 TestLangProj data set. Since it is only in memory, it will be recycled nicely.

The memory only BEP is reset to its pristine state during the base method call to “FixtureSetup()”. This is defined on the base class for all FDO dependent tests “FdoTestBase”. You all will call that base method if you override the method, right? :-)

What can one assume is in the memory-only test database? Any writing systems? LangProject? Lexicon? Any possibility lists? Are there helper functions to create commonly-useful sets of objects?

Unit tests ought to test the System Under Test (SUT), and only the SUT. One should assume systems the SUT uses have been tested adequately. If they have not, then they should be. One can burn up a ton of cycles checking things not a part of the SUT.

### What won’t work yet

Paragraph-level text editing using Views won’t work yet (you will get assertions when PropChanged is called directly).

Todo: anything else worth mentioning?

## A Worked Example

I thought it might be interesting to include here the steps I (JohnT) went through in order to get the LexEdDll to build successfully. This is roughly in chronological order.

### Fix FDO ‘using’ declarations

Remove using declaration for obsolete FDO subdomains (SIL.FieldWorks.FDO.Ling, etc.). You may need to add SIL.FieldWorks.FDO.DomainImpl and/or SIL.FieldWorks .FDO.Infrastructure.

### Make everything use interfaces and Tags

Change all references to FDO implementation classes to interfaces, e.g., when it complains that LexEntry is undefined, change to ILexEntry. (Do not add a namespace reference to allow continued use of LexEntry; this won’t work much longer). Watch out for class names that match property names; ReversalIndex, for example. You don’t want to add the “I” to the property name.

Change sequences like ILexEntry.kclsidLexEntry to LexEntryTags.kClassId. Across the whole project, with regular expressions, replace “I{[^.]\*}\.kclsid\1” with “\1Tags.kClassId”.

Also, things like (int)LexEntry.LexEntryTags become just LexEntryTags; you can do this with a project-wide replace using regular expressions, “\(int\)[^.]\*\.{[^.]\*Tags}” is replaced with “\1”.

### Undotasks/UnitOfWork

Change things like:

using (new UndoRedoTaskHelper(m\_obj.Cache, sUndo, sRedo)) {...}

to

UndoableUnitOfWorkHelper.Do(sUndo, sRedo, m\_obj, () => {...});

That is, the old code block becomes a delegate implemented as a lambda expression.

Or, preferably, the contents of {...} should be a single call to a zero argument method (Undo handling is a separate responsibility from the task that can be Undone, and that separate task may be useful to include in other tasks, too, whereas you cannot now nest undoable tasks). If so,

using (new UndoRedoTaskHelper(m\_obj.Cache, sUndo, sRedo)) {DoTheTask();}

becomes

UndoableUnitOfWorkHelper.Do(sUndo, sRedo, m\_obj, DoTheTask);

Note that there no parens after DoTheTask; you are passing a delegate.

There’s also an approach using (var helper = new UndoableUnitOfWorkHelper(…)), but you have to remember to set helper.RollBack to false as the last step in your task; I think the Do approach is cleaner.

### Object Creation

Object creation. Change things like

                ILexEntryRef ler = new LexEntryRef();

                ent.EntryRefsOS.Append(ler);

or

                ILexEntryRef ler = ent.EntryRefsOS.Append(new LexEntryRef());

to

                var ler = ent.Services.GetInstance<ILexEntryRefFactory>().Create();

                ent.EntryRefsOS.Add(ler);

(You don't absolutely have to change the type to 'var', it's just nicer. You do have to make two lines of it; Add() does not return the thing added. Note the change from Append to Add, as well as the more obvious use of the factory.)

This is one of the things I don’t like much; there is a discussion on the Wiki of possible alternative approaches to object creation.

In case you’re not appending, you’ll also need to deal with the fact that InsertAt() has changed to Insert, and the arguments have swapped (index comes first). Slightly painful, but puts us in line with other .NET interfaces.

### Get rid of calls to PropChanged

Commonly these are to inform the Views of newly added objects; these can just be deleted, the new UOW is smart about sending them AFTER you have initialized the objects.

(If they are to update the display of virtual properties you have a harder problem. I didn’t come across one of these and haven’t entirely figured out how to handle them.)

### Dealing with HVOs

In many cases where you are passing HVOs to methods of CmObject property collections and sequences you will need to pass instead a CmObject. One way to get one (if you already have some other object) is with anObj.Services.GetObject(hvo). Or if you have a cache, cache.ServiceLocator.GetObject(hvo).

HvoArray (on our sequence property classes) becomes ToHvoArray() (or consider whether the algorithm could be changed to use objects).

Atomic properties ending in Hvo: insert dot, or (preferably) do something equivalent using objects, e.g., affAllo.MsEnvPartOfSpeechRAHvo != 0 becomes affAllo.MsEnvPartOfSpeechRA != null. (In this case affAllo.MsEnvPartOfSpeechRA.Hvo != 0 won't work, because it will produce a null reference exception).

### Implementations of PropChanged

Commonly these are kludges to produce side effects in the model. This should not be done! In fact it will no longer work; you can’t make model changes in the UOW phase where PropChanged is sent.

Instead, if certain things ought always to change when certain other things change, we now want to implement that domain logic in FDO as part of the setters of the primary properties.

That may be a project…I haven’t entirely figured out yet how to hook changes to object sequence properties. To get the thing to at least compile quickly, you may block out the whole PropChanged method with #if WANTPORT. You will typically also have to delete the code that says the class implements IVwNotifyChange, and that adds and removes it from collection of things that get such notifications.

### Writing systems and multistrings

Writing system collections are found on the writing systems repository, sometimes with somewhat different names. For example, cache.LangProj.InstalledWritingSystems becomes cache.ServiceLocator.WritingSystems.AllInstances().

Some methods have also changed signatures, returning WS objects or object collections rather than HVOs; add .Hvo or .ToHvoArray() or the like, as needed.

Some values retrieved from multistring accessors require .Text appended to convert from TsStrings to strings.

### Stuff on the implementation class

Since the implementation classes (like LexEntry) are no longer accessible, something must be done about their nested types and static methods.

Enums may need to be moved to the Tags class. I had to do this for LexRefTags.MappingTypes. This is stretching the responsibility of the Tags classes; if someone has a better idea, speak up.

I stretched a point further and put a method on LgWritingSystemTags that returns a fixed ITsTextProps. I like that even less well. Ideas welcome.

### Valid objects, dummy objects, and the like

Delete calls that check whether objects are real, or valid, or dummy, or that attempt to convert dummies to real. All objects in the current system are real. I’m not 100% sure that we will never need to check whether an object we still have a reference to represents something that has been deleted, but for now, there’s no way to do it.

### Working with flids

If something really has to be done using flids, you can retrieve the facade. For example, I changed this:

return  obj.GetObjectInAtomicField(flid) as IFsFeatStruc;

to this:

int hvoFs = obj.Services.GetInstance<ISilDataAccess>().get\_ObjectProp(obj.Hvo, flid);

if (hvoFs == 0)

                 return null;

return obj.Services.GetObject(hvoFs) as IFsFeatStruc;

(We could re-implement GetObjectInAtomicField, but we don't want to encourage flid-based programming.)

### Vanished methods

You may well find some methods being called which no longer exist. Often you can find the old implementation in FDO in a file that isn’t currently being built, or disabled by #if WANTPORT. If appropriate, you can fix the implementation as needed and reinstate the method (also in the interface, if you need it outside FDO).

### Wrapping up

The last thing I did was to change LexText.build.xml to reinstate LexEdDll into the dependencies of LexTextExe. I also removed MorphologyEditorDll from the dependencies of LexEdDll, since I’d already applied WANTPORT to the couple of things actually wanted from there. I noted this in the build file as a reminder to put it back eventually if needed.

# Unit Test Framework

The FDO unit test framework has been revised a good bit. The test class hierarchy is:

FdoTestBase (abstract)

PersistingBackendProviderTestBase (abstract)

(One class per embedded type BEP)

PersistingClientServerBackendProviderTestBase (abstract)

(One class per client-server type BEP)

MemoryOnlyBackendProviderTestBase (abstract)

(Many subclasses)

(Various classes that do not need FdoTestBase support)

## FdoTestBase (abstract)

I don’t see any more need at this point for more direct subclasses of FdoTestBase.

## PersistingBackendProviderTestBase (abstract)

The PersistingBackendProviderTestBase serves as the location where all tests are defined for testing the actual (de)serializing capabilities for each supported BEP. There is to be one subclass of PersistingBackendProviderTestBase for each supported BEP, but that subclass does not add any BEP specific tests. The reason for this is that most of the BEP’s API is internal so cannot easily be directly tested. The idea of placing all tests on PersistingBackendProviderTestBase is that one can then test the affects of the internal API by dogin some action, remembering the state, persisting the changes, restarting the BEP, and checking the persisted and reloaded results against the remembered starting state. This will test all of the internal API, but indirectly.

I see a need for one more new abstract subclass of PersistingBackendProviderTestBase. That class will add tests for concurrent access, and it will have one concrete subclass for each supported concurrent access-friendly BEP. The other BEPs do not support concurrent access, so they do not need those tests.

## MemoryOnlyBackendProviderTestBase (abstract)

The MemoryOnlyBackendProviderTestBase class is the main workhorse base class for unit tests, since it uses an in-memory BEP. The assumption is that all unit tests that need FDO will derive from this base class, both in the FDO test suite, as well as external test classes.

An alternative to external clients using MemoryOnlyBackendProviderTestBase is that we create mock FDO domain objects that have no BEP at all. Such a mock FDO framework would also not need the UOW support. It would just be barebones implementations of the domain model interfaces. The justification for doing this is that those external clients ought not be testing FDO, but their own code.

Those who do use MemoryOnlyBackendProviderTestBase can expect to have a fully populated MDC, which has information on all of the model-defined classes in FDO, as well as all model-defined properties and dev-defined “virtual” properties. The MDC is populated with this information by reflecting over the FDO assembly and adding the metadata to the cache, based on some new subclasses of the .Net Attribute class. Two of these custom attribute classes support the model-defined classes and properties (ModelClassAttribute and ModelPropertyAttribute). The third (VirtualPropertyAttribute) supports dev-defined virtual properties.

All BEPs that are created from scratch (rather than restarted) are populated with some basic objects, among which are one LangProject and a few writing systems, among other basic objects. One can see the various objects created by looking at the FDOBackendProvider::BootstrapNewSystem() method and the methods it calls.

There is one important extant subclass of MemoryOnlyBackendProviderTestBase, which is ScrInMemoryFdoTestBase. It is important because it serves as a sample of a class that adds lots of methods used to support testing some sub-domain of FDO, in this case Scripture tests. Other classes can be created to do similar things for other areas, such as the lexicon, interlinear texts, Data Notebook, etc.

# Starting Flex-The window Frame

Until such time as a regular mechanism is provided to start Flex, you will need to use a command line option (shortcut or VS startup). It will be like this: -db Memory

Currently supported options are:

1. Firebird,
2. DB4o,
3. BerkeleyDB,
4. XML, and
5. Memory

Flex does have a working File-Open menu, which uses the simple .Net File-open dialog window. That dialog window is fed the supported file extensions, so we can open a data store on any supported, persisted, BEP.

1. I (RandyR) expect this document to not be exhaustively complete, but that it will require updating as questions come up. [↑](#footnote-ref-1)