Interim Progress Report

Date Submitted

March 3, 2017

Prepared By

Joshua Zelin
Patrick Clouse
Christopher Lupo
Nathan Faulkner

Table of Contents

Executive Summary

Fujitsu is a large multi-national company. As a result, they have several projects that undergo refactoring. Some of these projects are Software Development Kits (SDK). This creates a problem for client code, also known as solutions, that is dependent on an older version of a refactored SDK. These solutions will eventually need to adopt the new SDKs, which in turn means that the solutions will also need to be refactored. Usually this process is tedious and ties up human resources. This will slow down the adoption of the new SDK, and prevents programmers from working on other projects. The answer to this dilemma is automation, specifically the automation of refactoring client code. In order to accomplish this, our project will meet the following goals, which have been split up into two stages.

1. Take old and new SDKs as inputs and create a mapping from old to new.
2. Use those mappings to change client code from using the old SDK to using the new SDK.

Between versions of SDKs, namespace names and class names might have changed. Functionally these namespaces and classes are the same between versions, but the problem still remains that it is hard to tell which namespace name or class name in the old SDK matches with which namespace name or class name in the new SDK. This can be solved by providing every class name and namespace with a globally unique identifier, which does not change between versions. Fujitsu has already made an attribute for their classes that does this called `ModelIdentifer`. This `ModelIdentifier` attribute will allow us to know which name in the old SDK maps to which name in the new SDK. Our first stage deals with reading the old and new SDKs and using these `ModelIdentifers` to create a mapping from old to new. As of this report, this stage has been accomplished. We can read in an old and new SDK and programmatically create the mappings.

Once the mappings have been created, they need to be stored so that they can persist between runs. Saving the mappings in a database accomplishes this goal. The database will also link the two stages together, so that any mappings created in stage one can also be used by stage two to change client code. Creating the database and saving the mappings has been accomplished.

Stage two involves using the mappings generated in stage one, which are retrieved from the database, to refactor the client code. Anywhere that clients have using statements that reference refactored namespaces from the old SDK need to be updated to use the new namespaces. Also anywhere that old classes are used also needs to be updated so that the client code can adopt the new SDK. Roslyn, a .NET compiler, allows us to analyze and change client code. For a particular class name in client code, we can see what that class name is, grab the new class name from the mapping, and use Roslyn to replace the old name with the new name and update client code. Currently, the ability to change using statements from old namespaces to new namespaces has been accomplished. Also changing class names in some instances has also been done.

Throughout all of this, testing will be done on all stages. Black box testing and unit testing will be needed. Creating the black box tests is an ongoing process, and there are already many tests written.

Project Description

Sponsor Background

Fujitsu America is the child company of Fujitsu, which is a global leader in information systems and technology. The branch our sponsor is working out of is in Durham, NC. It is one of the top three suppliers of retail systems and services worldwide. Using Microsoft’s .NET development platform, these systems offer a high-performance yet open platform that retailers as diverse as Nordstrom, RadioShack and Dressbarn are able to customize.

Contact Information

Terry Crawford Terry.Crawford@us.fujitsu.com
Dave Killian Dave.Killian@us.fujitsu.com,
Pat Majerus Pat.Majerus@us.fujitsu.com
George Havens George.Havens@us.fujitsu.com
Jeff Canady Jeff.Canady@us.fujitsu.com

Problem Statement

Fujitsu develops SDKs that clients use in their own projects. In old versions of the SDKs there are a set critical artifacts of namespaces, class names, and assembly (dll) file path references. However, in the newer versions of these SDKs some of these artifacts have changed, primarily the namespaces and dll filenames. If the client tries to use the new SDK the client project will not compile because these critical artifacts changed. If the client code that uses the old artifacts was changed to use the new artifacts the client project would compile. The complexity of this transformation largely rules out simple text editor automation because the migration is likely to affect both the consumer source code and the project structures that build that source code. Therefore the solution to this problem is to automate changing client projects to use the new artifacts instead of the old ones.

All projects are going to require refactoring at some point, no matter how big or small they are. Refactoring a SDK can lead to issues with already existing code in previous solutions. The result of this is spending an unnecessary amount of man-hours trying to fix all of the errors that the refactoring created. If this process were to be automated, then there would be no need for human intervention when it comes to refactoring projects.

Project Goals & Benefits

The goal for this project is to be able to run an automated tool that will transform the client code from using the old SDK to using the new SDK. Client projects are written in C# or Visual Basic The majority of the changes in the SDK involve namespaces, fully qualified names and member variable declarations. There are some edge cases that we are not required to cover that will prevent the entire project from compiling. As such, having the entire project compile is a stretch goal. However, with the three types of major changes mentioned above, we should be able to cover at least 90% of the changes between the SDKs, the majority being namespace changes.

The last goal for this project is for Fujitsu to run the refactoring tool against an actual customer project, however this is a stretch goal and is intended only if the team feels confident. The major benefit of this project is that it will enable Fujitsu to quickly refactor their large code bases on any project, without having to spend human effort tracing down namespace conflicts.

Development Methodology

As the executive summary mentions, our project is split into two parts, which we call `CreateMappings` and `TransformClient`. Since there are two main parts, our team has decided to split the project into multiple iterations:

1. Iteration 1 - Create mappings between the old and new SDK and test that the mappings are correct.
2. Iteration 2 - Transform the client project so it uses the artifacts in the new SDK instead of the old SDK and test the the project was changed correctly.
3. Iteration 3 - Test our code on a real client project provided by Fujitsu and make necessary changes to ensure that the three types of major changes mentioned in the Project Goals and Benefits Section.
4. Iteration 4 - Add additional functionality to handle edge cases to bring the client project closer to compiling as time permits.

The deliverables for our project consist of the following:

1. A zip file of the executable files
2. A detailed instruction guide on how to run our code
3. A demo of our code executing

Team Roles

Patrick Clouse	Development Leader
Nathan Faulkner	Testing Leader
Christopher Lupo	Tracer Development Manager
Josh Zelin	Team Leader/Database Manager

Challenges

The hardest challenge is to transform the client so that *everything* compiles. Most of the changes are of the namespaces, fully qualified names, and member variable declarations. There are some edge cases we might not cover due to time constraints that will prevent the entire project from compiling. Therefore compilation of the entire project is a stretch goal.

A second challenge will be using our system in a memory efficient way. There is already a tool called Resharper that does the refactoring that we are tasked with. However, if Resharper is used on an entire project, the machine running it will run out of memory before it finishes the job.

Resources Needed

1. Visual Studio, IDE to develop the refactoring project in
2. C#, language our project is written in
3. Roslyn, .NET compiler platform used to read and change source code
4. Database, to persist the mapping from old to new SDK versions. There's too much data to store in a machine's main memory
5. Windows OS, the program will not be written to run on any other operating system and will not be tested on other operating systems.

Requirements

Overall View

The senior developers at Fujitsu America will be the main users of this system. The big picture of this system is to transform a client project to use new SDK artifacts instead of old SDK artifacts. This will allow Fujitsu America to automate the refactoring of their SDK for their clients. This project is a little different than most as there is only one requirement. However, this feature has multiple sub-requirements. This is the case because features should be testable on their own and each sub-requirement builds upon the last and therefore cannot be tested independently as a full feature.

Functional Requirements

1) The system shall use dll files in order to create mappings between an old and a new SDK and then transform client code referencing the old SDK to reference the new SDK
    a) The system shall take two folder paths, a path to a folder containing the original dll files and a path to a folder containing the new dll files, and a .csproj file to be updated to contain references to the updated client code.
    b) The system shall have a unique identifier, known as a `ModelIdentifier`, for all classes contained within the client code. The `ModelIdentifier` shall be the same in the old and new SDK relative to the class, even if the class name or the namespace changes.
    c) The system shall create a mapping between the old and new SDK using the unique identifiers. This mapping shall contain the `ModelIdentifier`, old class name, new class name, old namespace, new namespace, old dll file path, new dll file path.
    d) The system shall transform the client project by leveraging the mappings created between the old and new SDKs.
        i) The system shall update the old assembly file references in the client project.
        ii) The system shall update the using statements in the client code.
        iii) The system shall update the fully qualified names in the client code.
        iv) The system shall update the class types of member variable declarations in the client code

Non-Functional Requirements

1. Users shall access the system via command line interface
2. The system will be able to transform client code written in C# and VB.net
3. The system should persist the mappings. They should be stored after the system has finished running

Constraints

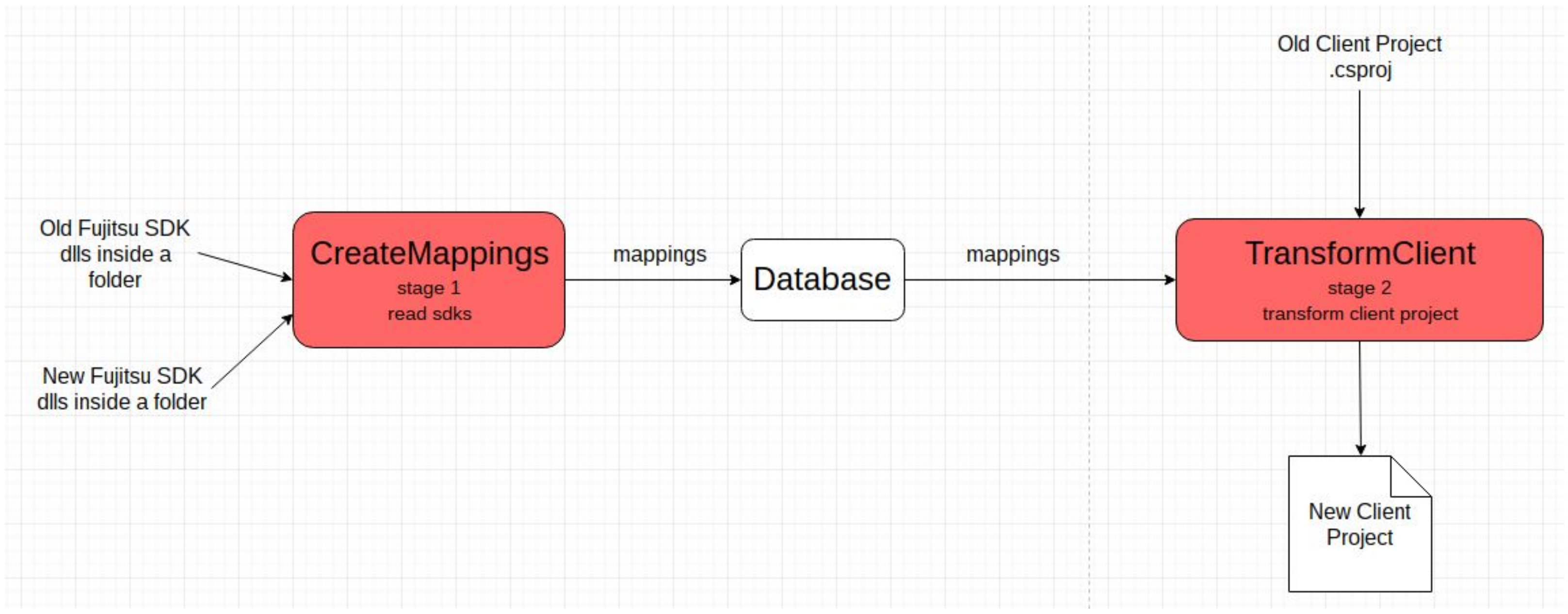
1. The implementation language must be C#
2. Development should be done in Visual Studio 2015
3. The system will be written and tested on Windows OS and is not guaranteed to run on any other platform.
4. The system will follow C# coding conventions defined by Microsoft that can be found here

Assumptions

1. Client code will *not* share a namespace with SDK code
2. The namespace containing the `ModelIdentifier` will *not* change between versions
3. The new SDK will be installed in the same relative path as the original
4. The dll files belonging to the new SDK will be installed at the time of conversion
5. The old and new SDK will *not* exist at the same time in the same binary tree at runtime while using Roslyn
6. The client project will be converted to use the new SDK before given to us
7. The Copy Local function of Visual Studio is turned off

Design

As previously mentioned, our design can be split up into two steps, which will end up being two different stages that run one after another. We designed this in stages primarily because we anticipate the user to run each stage in batch format. For example, the user will create large amount of mappings over different Fujitsu America systems then fix large amount of client projects. If everything was in only one stage then the user would need to create new mappings every time they wanted to transform a client project. The following is our design at a high level:



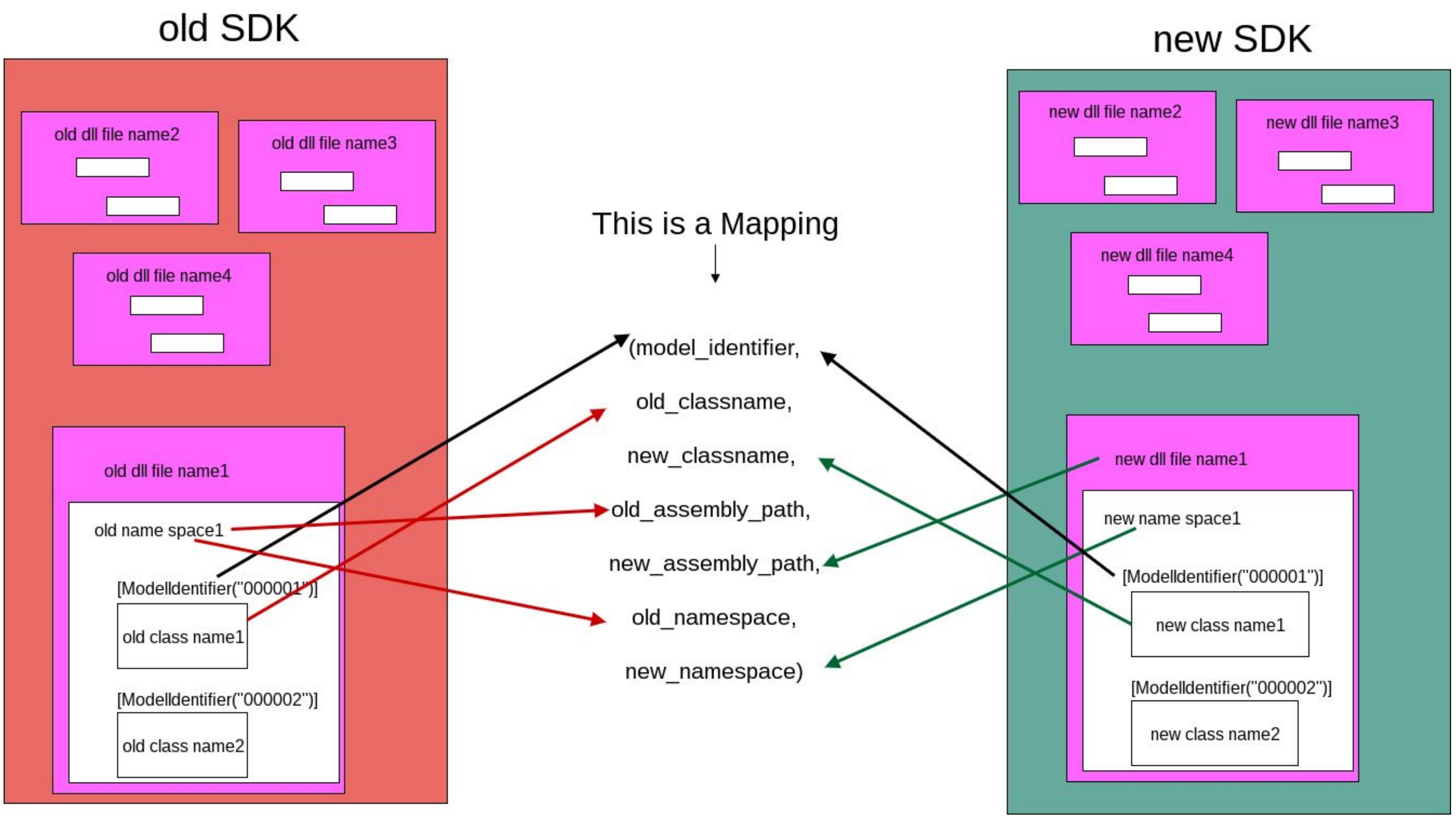
Stage 1: `CreateMappings`

1. Input:
    a. Old SDK dll folder path.
    b. New SDK dll folder path
    c. The classes in the dll files will be tagged with a `ModelIdentifier` custom attribute
2. Process:
    a. Find the symbols tagged with `ModelIdentifier`
    b. Create the mappings between the old and new artifacts
3. Output:
    a. Mappings stored in SQL database

This stage has two .cs classes:

1. `ReadProject.cs`. This is the starting point and prepares the dll files to be processed by `ReadFile.cs`. This also manages saving the mappings when dll files have been read.
2. `ReadFile.cs`. This reads individual dll files to find the custom attribute that identifies code artifacts. This also creates the mapping between the old SDK artifacts and the new SDK artifacts

Mapping Example



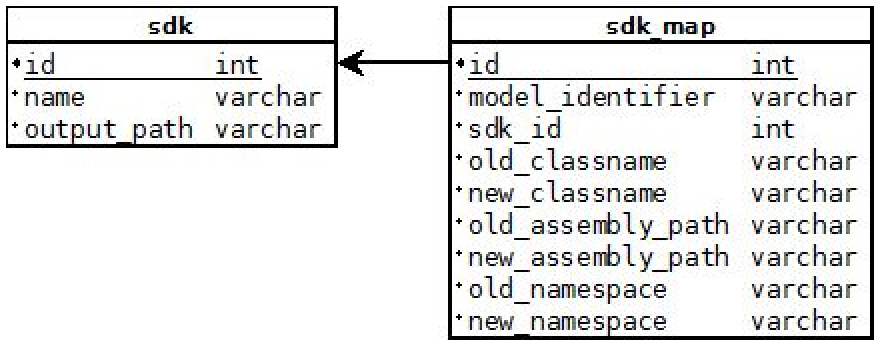
Stage 2: `TransformClient`

1. Input:
    a. Old client project (.csproj)
    b. Database mappings created from `CreateMappings`
2. Process:
    a. Transform the old SDK artifacts to the new SDK artifacts
3. Output:
    a. New client project with updated SDK artifacts

This stage has two .cs classes:

3. `ProjectTransform.cs`. This is the starting point and manages iterating over client files and changing the .csproj file
4. `FileTransform.cs`. This class transforms the source code

Database



The database design has two tables. The `sdk` table keeps track of each execution of `CreateMappings`. The `sdk_map` table stores the mapping data between the old and new `sdks`. `CreateMappings` takes in a name as a command line argument, which must be unique to this execution. The `sdk` table also has an `output_path` which is the folder path that the executable for the client project is located. The `sdk_map` table has a unique index on `model_identifier` and `sdk_id`, meaning that `model_identifier` is guaranteed to be unique per run of `CreateMappings`. The use of this unique index along with the `sdk` table allows for the persistence of mapping data on subsequent executions of `CreateMappings`.

When `sdk_maps` are created, they are assigned the following:

1. The unique `model_identifier` belonging to the class being mapped
2. The `sdk_id` the map belongs to
3. The class name from the old SDK, and the class name from the new SDK (these may or may not be different)
4. The assembly path that points to the old SDK, and the assembly path that points to the updated version of the SDK
5. The namespace from the old SDK and the namespace from the new SDK (these may or may not be different)

User Interface

The user interface for both stages will be a command line. It is run as follows:

```
$ C:/path/to/CreateMappings.exe <old_folder_path> <new_folder_path>
<sdk_name>
```

```
$ C:/path/to/TransformClient.exe <project’s .csproj file> <sdk_name>
```

Console feedback will be given during execution to inform the user(s) of current progress

Implementation

Any file paths referenced are examples and solely for explanation purposes.

Stage 1: `CreateMappings`

`CreateMappings` is a C# project which contains two .cs classes:

1. `ReadProject.cs`
    a. This is where `main` is. This class reads in the dll folder path and iterates over each file in the folder then pushes data to the database once all of the mapping data has been created by ReadFile.cs. If the file is a dll then it is passed to ReadFile. This class also keeps a dictionary which temporarily stores the mapping data before saving to the database. Because these SDK's may be large and the dictionary is stored in main memory we are filling the dictionary with the old data and saving in the database. Then we are clearing the dictionary and duplicating the process for the new data
2. `ReadFile.cs`
    a. This class reads an individual dll file. It iterates over all the objects in the dll file that have custom attributes. If the object has `ModelIdentifier` as a custom attribute then it reads the properties we need to track and adds them to the dictionary in ReadProject. .NET reflection is what is used to read the dll file.

Stage 2: `TransformClient`

`TransformClient` is a C# project which contains two .cs classes:

1. `ProjectTransform.cs`. This is where `main` is
    a. This class reads in the .csproj and iterates over each file
    b. After all the files are processed the .csproj xml file is edited as follows:
        i. Remove all the `Reference` tags that have a dll path in the old SDK.
        ii. Add new `Reference` tags with new dll file paths. These new paths are inside a `HintPath` tag. The new paths will be a relative path in relation to the output path contained in the `OutputPath` tag.
        iii. The project output path is changed to refer to the folder containing the new SDK dll files. The new output path will have the same parent folder as the folder that contains the old SDK dlls.
        iv. The following is an example of a `Reference` tag that needs changing:

```
<ItemGroup>
  <Reference Include="SDK, Version=1.0.0.0, Culture=neutral, processorArchitecture=MSIL">
    <SpecificVersion>False</SpecificVersion>
    <HintPath>..\..\bin\SDK.dll</HintPath>
    <Private>False</Private>
  </Reference>
```

2. `FileTransform.cs`. This class uses Roslyn to read the source code of a single file then changes that file to use the artifacts used in the new SDK
    a. This uses a compiled version of the syntax tree to get a semantic model. The semantic model is needed for the more complex transformations. If you don't have a compiled version of the code then the syntax symbols are unavailable.
    b. The following are the specific changes that are made to the source code:
        i. Using statements
        ii. Fully qualified names
        iii. Member variable declarations
    c. A `DocumentEditor` is used to change the source code. It allows for writing to a syntax tree while also reading the tree. This is needed because the syntax tree is being read and written to at the same time.

Database

The database is integrated with Visual Studio using the LINQ library. This library auto-generates a `DataContext` class which contains a wrapper class for each table in the database. Each wrapper class stores the attributes for an entry into that table, as well as providing a means to add, edit, and delete table entries. However, the wrapper classes inside of `DataContext` only provide basic functionality. In order to make access to the database more usable, we implemented a singleton pattern.

First, we separated the database access into its own unique project (`DBConnector`), and added a dependency to the other projects so they could reference it. Next, we created connector classes for the database tables mentioned in the design section of this document. The examples given below will use one of the connector classes (`SDKMapSQLConnector`). These connector classes have private constructors and are initialized at runtime. They provide a static function called `GetInstance()` which returns the single instance that was created at runtime, thus completing the singleton pattern. Finally, we added more robust functionality to the connector classes in order to access the database in different ways:

1. The ability to save a single object or a list of objects
2. The ability to pass in a where clause and receive a single object, list of objects, or a dictionary of objects. The following two examples actually use this generic where clause functionality, which enhances code reuse.
3. The ability to get objects by their attributes. For example, `GetByOldNamespace(String oldNamespace, int sdkId)` in

`SDKMapSQLConnector`will return a list of `sdk_maps` where each `sdk_map` will have the namespace and SDK specified in the parameters

4. The ability to get a dictionary of objects, making it easier see which original value maps to which new value. For example, `GetClassnameMap(String ns, int sdkId)` will return a dictionary of values that belong to a certain namespace and SDK. The values returned will map an old classname to a new classname. This kind of functionality simplifies updating client code because we can do a simple dictionary lookup for the new value of a value we want to change.

Test Plan & Results

Black Box Tests

1. alias: Client code uses alias to reference the SDK.
2. assemblyChange: The name of the output assembly (dll file) is changed between the SDK versions. This tests that the project can modify references.
3. assemblyMerge: The old SDK uses multiple assemblies (dll files), but those are merged into a single assembly (dll file) in the new SDK. This tests that the project can remove refernces.
4. assemblySplit: The old SDK uses a single assembly (dll file), but that's split into two assemblies (dll files) in the new SDK. This tests that the project can add references.
5. basic: Class name changes between versions. This tests that the project can handle class name changes.
6. basicNamespace: Root namespace changes between versions. This tests that the project can handle namespace changes.
7. casting: The client code casts to a class in the SDK. This tests that the project can modify casting statements.
8. classInClass: The SDK has a class defined inside of a class. This tests that the project can handle inner classes.
9. extendsSDKClass: The client code extends an SDK class. This tests that the project can modify extension statements.
10. extraLibrary: The client code uses a library that's not part of the SDK. This tests that the project can handle non-SDK (and thus ModelIdentifier-less) code.
11. fullyQualified: Client uses fully qualified name for SDK class (rather than using a 'using' statement). This tests that the project can modify fully qualified names, not just class names.
12. fullyQualifiedModelIdentifier: The ModelIdentifier in the new and old SDK uses the full path of the ModelIdentifier, rather than relying on using/namespace sharing. This tests that the project can handle the SDK source code using ModelIfentifier with a fully qualified name.
13. generics: The client code uses the SDK class as the type qualifier for a generic class. This tests that the project can modify generic qualifier statements.
14. instantiatesSDKClass: The client creates a new instance of an SDK class. This tests that the project can modify instantiation/new statements.
15. multiAssembly: The SDK uses multiple assemblies (dll files). This tests that the project can handle more than one assembly (dll file).

16. multiBasic: Multiple namespaces, files, and classes per file in SDK. Multiple files and classes per file in client. This tests that the project can handle these things all at the same time.
17. namespaceInNamespace: The SDK has a namespace directive inside of a namespace directive, creating an implicit nesting. This tests that the project can handle the SDK source code not providing the full namespace name in the namespace statement.
18. newConflicts: Client code has class name matching a class name in the new SDK. This tests that project can eliminate name conflicts that might arise from a class name change in the SDK.
19. nonRootUsing: The client's "using SDK" statement is inside of the namespace block. This test only applies to the C# client, the VB client only exists for consistency. This tests that the project handles using statements that aren't at the root of the syntax tree.
20. nothing: There is no difference between the SDK versions. This tests solely that the project is updating the client's references to point to the new assemblies (dll files)
21. oldConflicts: Client code has a class name matching a class name in the old SDK. This tests that the project won't mistakenly think that a reference to a client class is a reference to an SDK class.
22. typeof: The client uses the typeof expression with an SDK class as the parameter. This tests that the project can modify typeof statements.
23. unused: Client code does not use the SDK. Tests that we're not making assumptions about a first element existing. This tests that the project can handle files that don't make any references to the SDK.

Running the Black Box Tests

1. Copy the folder matching the test id (so that that we don't modify the original)
2. Compile the old SDK and the new SDK
    a. Open the old SDK's solution file (.sln) in visual studio
    b. Click on Build > Build Solution
    c. Repeat for the new SDK
3. Compile and run the client and verify the output (to make sure everything's working properly before we run our code)
    a. Open the clientC# solution file (.sln) in visual studio
    b. Click on Build > Build Solution
    c. Click on Debug > Start Without Debugging
    d. Make sure the output matches the expected output
    e. Repeat for the clientVB solution file
4. Run the program to make the mappings
    a. Execute the create mappings program with the following arguments
        i. The path to the bin1 folder
        ii. The path the the bin2 folder

        iii. The test id
5. Run the transformation program on the client project
    a. Execute the transform client program with the following arguments
        i. The path to the .csproject file (this is in the ClientC#/Client folder)
        ii. The test id
    b. Repeat part a, using the .vbproject file instead (this is in the ClientVB/Client folder)
6. Compile and run the client.
    a. See step 3
7. Make sure the output matches the expected output

Finding the expected output:

Inside of the folder matching the test id, there will either be a file called “expectedOut.txt” or both a file called “expectedOutOld.txt” and a file called “expecetedOutNew.txt”. If the file “expectedOut.txt” exists, this file contains the expected client output (steps 3 and 7). Otherwise, “expectedOutOld.txt” contains the expected client output for step 3, and “expectedOutNew.txt” contains the expected client output for step 7.

An easier, automated, execution:

Open the test explorer in Visual Studio. Run the TestMapping* and the TestPostTransform* tests.

Unit Tests

These have not yet been implemented. This is in part because, due to the linear nature of stage 1, the unit tests would have been no more useful than the black box tests. Stage 2 is still in progress, but it seems like it will also be linear, limiting the usefulness of unit tests. However, the database classes will be important to cover, as they are used from several different places. Also, the unit tests will be necessary to meet the code coverage requirement, as the black box tests are unable to measure code coverage.
When these tests are written, they will use the unit test and code coverage built into Visual Studio Enterprise.

Task Plan

Task Plan

Item	**Owner(s)**	**Due Date**	**Status**
Sponsor meeting	All	January 20	Complete
	All	January 26	Complete
	All	February 2	Complete
IPR draft	All	February 3	Complete
Iteration 1 - Create Mappings	All	February 6	Complete
Duplicate the `ModelIdentifier` class that Fujitsu uses to uniquely identify their classes	cblupo		Complete
Create functionality to use reflection to find unique identifiers on classes and add data to a list that will later be added to the database	cblupo		Complete
Test the above	nafaulkn		Complete
Create a wrapper class to hold mapping information that needs to be saved	cblupo		Complete
Create database and association tables to store mapping information in	jtzelin, pbclouse		Complete
Use LINQ to integrate the database with C#	jtzelin, pbclouse		Complete
Test that the database is able to save information	nafaulkn		Complete
Create new project to handle all interactions with the database	jtzelin, pbclouse		Complete
Test that the mapping creation integrates properly with the database	nafaulkn		Complete
Allow command line arguments for two different folders, one for the old assembly information and one for the new assembly information	cblupo		Complete
OPR 1	cblupo	February 7	
Sponsor meeting	All	February 9	Complete
	All	February 16	Complete
	All	February 16	Complete
	All	March 2	Complete
OPR 2	nafaulkn	March 2	Complete
IPR final	All	March 3	Complete

Iteration 2 - Transform Client Using Roslyn	All		In Progress
Create test methods to use Roslyn and create a proof of concept	cblupo		Complete
Iterate only .cs files, folders, and subfolders	cblupo		Complete
Create ability to read the saved mappings from the database	jtzelin, pbclouse		Complete
Change the test methods to use real saved data instead of hard-coded test values	cblupo		Complete
Test that we receive the same mappings we saved	nafaulkn		Complete
Reliably write to source code using a document editor	All		Complete
Allow ability to change using directives in client code	All		Complete
Test that we can change using directives	nafaulkn	March 3	In Progress
Allow ability to change member declarations in client code	All		In Progress
Test that we can change member declarations	nafaulkn		In Progress
Allow ability to change fully qualified names in client code	All		In Progress
Test that we can change fully qualified names	nafaulkn		In Progress
Find a way to add/remove references from the .xml build file in visual studio using LINQ to XML	cblupo		In Progress
Add test references to create a proof of concept	cblupo		In Progress
Integrate references with database and save real data instead of test data	All		In Progress
Test that we can edit the build files and that this connects properly with the database	nafaulkn		In Progress
	All	March 2	Not Started
Sponsor meeting	All	March 16	Not Started
	All	March 23	Not Started
OPR 3	jtzelin, pbclouse	March 30	Not Started
Sponsor meeting	All	March 30	Not Started
Iteration 3 - Deployment	All		Not Started
End to end test of system using our test code	All		Not Started
Use client code and SDK provided from Fujitsu in order to test the system	All	March 31	Not Started
Provide instructions for easy deployment	All		Not Started

Sponsor meeting	All	April 6	Not Started
	All	April 13	Not Started
	All	April 20	Not Started
	All	April 27	Not Started
P&P	All	May 3	Not Started
Iteration 4 - Stretch Goals	All	TBD	Not Started
Add unrequired source code edge cases as time permits	All		Not Started
Change client projects that use Visual Basic	All	TBD	Not Started

Team Contact Information

Patrick Clouse	pbclouse@ncsu.edu
Nathan Faulkner	nafaulkn@ncsu.edu
Christopher Lupo	cblupo@ncsu.edu
Joshua Zelin	jtzelin@ncsu.edu

Suggestions for Future Teams

1. Start researching the required technologies early.
2. Roslyn documentation is scarce.
3. Be careful when editing .csproj files. These are what allow your project to compile. If they are changed incorrectly, Visual Studio will not longer recognize your project. Make a copy of your code before you run your tests
4. Renaming folders in Visual Studio will *NOT* rename them in your actual file system. Once you rename them in Visual Studio, you must manually go and rename them on your file system. Do *NOT* rename them on your file system first.