WHITE PAPER

MIGRATING NATURAL TO C#

TABLE OF CONTENTS

Migrating Natural to C#

How can businesses move safely and cost-effectively from SAG Natural to C#? This document explores migration through automated code conversion, a mature approach being perfected by the software company Astadia, as an answer to this question.

The document explores the key business drivers for making the step from Natural to C#, and how Astadia tools make this transition possible. It will explain the added value Astadia migration offers over other approaches and demystify the migration process with a step-by-step look at the way Natural can be transformed into working, maintainable C# code.

1. Why Migrate from Natural to C#?

1.1. Reasons to Go to C#

There are many good reasons to make the move from Natural to C#, but the following are the largest concerns for businesses:

- High (and continuously increasing) maintenance and runtime fees for the existing Natural products.
- Shrinking availability of Natural developers and lack of interest in Natural from young developers.
- Lack of an easy transition path from the mainframe version of Natural to its Linux, Unix, and Windows platform versions.

C# offers answers to all of the above concerns:

- Maintenance fees are negligible when compared to Natural (or even nonexistent depending on the choices made).
- Even though there is no one universally accepted barometer to measure the popularity of the various programming languages, there is no denying that C# is considered to be one of the most widely used programming languages today[1]. In addition, the design of C# aims for programmer portability, for the vast range of developers familiar with C, C++, and Java.
- Even though C# has been developed by Microsoft, the product and its documentation are publicly available on the Internet (the first versions were even ISO/IEC and ECMA standards).

Next to that, moving to C# also means:

- Enabling the use of a state-of-the-art IDE, with extensive debugging, refactoring, profiling and (unit)testing support.
- Enabling the use of thousands of (third-party) libraries, covering almost all imaginable computing needs: database interaction, mail/ftp/http/… communication, parsing, xml processing, …
- Enabling the use of modern application architectures including the use of an application server, web front-ends, SOA, cloud deployment, …

[1] References: https://pypl.github.io/PYPL.html, http://trendyskills.com/

1.2. Approaches: Replacement, Rewrite or Automated Migration

Once the need to move away from Natural has been established, the next big question is: how to turn a large, mature Natural code base into the C# equivalent?

Most organizations take one of three approaches:

1. Replace by third party COTS (Commercial Off-The-Shelf) software,
2. Rewrite, or
3. Migrate automatically.

Replacement by COTS software is the most cost-effective solution, but only practical if one can actually find 3rd party software that offers the same functionality as the existing Natural application (and provides a migration path for the existing data).

Rewriting on the other hand too often has led to huge costs and ultimately to project failures. According to a research note from Gartner[2] the cost for rewriting is between \$6 and \$26 per Line of Code (LOC), and accomplished at a rate of 160 LOC per day and per developer. Even extrapolated to a code base of only 1M LOC, it is obvious that these will become very expensive, lengthy, and risky projects.

Astadia advocates automated migration as the only realistic approach for large, legacy applications:

- It offers consistency: since all source code is translated by software, there can be no differences in quality and all existing functionality is kept as-is.
- It offers speed and continuous improvement possibilities: a complete code base can be converted in a couple of hours, meaning that this process can be repeated as often as needed or wanted.
- It offers a very large degree of testability: when keeping the converted application's functionality and user interface unchanged, the original application's behavior can effectively be used as a regression test (and running this regression test, too, can be automated).
- It offers minimal interruption: because of the speed of the conversion, any "code freeze" period before taking the migrated application into production can be kept to a minimum. The existing team can keep working on its daily tasks, including the ongoing maintenance of the Natural code, during most of the migration project.
- It gives Astadia the confidence to offer projects with fixed duration and price, equivalent functional behavior, and equivalent performance.

Like most IT projects, a migration project is a complex undertaking, one that deserves the right amount of expertise and dedication. Astadia has developed project and product methodologies, and we are happy to provide you with more information regarding these.

[2] Gartner Research Note "Forecasting the Worldwide IT Services Industry: 1999,1"

2. Natural to C#: Architecting a Migration

This section focuses on the underlying design principles that any migration from Natural to C# should consider, and how Astadia has incorporated these in its set of flexible tools.

2.1. Keys to a Successful Migration

What are the keys to a successful migration project? A primary concern in any migration is how to validate the functional correctness of the new programs. In Natural to C# migrations, this can be hard, especially if the Natural programs are driving business-critical processes and they are being adapted to evolving user requirements while a migration is running at the same time.

Under these circumstances, migrations should target:

Equivalent Program Behavior and Performance
For programs to be functionally correct, business users have to accept them. For large Natural applications that are being actively maintained, users and developers have an existing and well-tried process to specify changes, develop, test, and bring new releases into production. This familiar process should not be disturbed by migration efforts. An important best practice in migrations is targeting functional and performance equivalence with a production release and avoid whenever possible the introduction of functional changes that don't follow the standard development process. This approach makes it possible to leverage automated testing tools to reduce project costs, improve accuracy and move the project forward faster.

Automated, Iterative Processes
To manage risk, the generally accepted process in migrations is no different from that of conventional software engineering: using agile, iterative processes that can periodically align the migration project to the latest versions of the Natural programs undergoing the migration. Consistency and speed are also critical to a parallel iterative process. Using tools to automate the migration imposes rigorous consistency of transformation and brings the end goal within reach of stakeholders.

Developer Confidence
There are many differences between Natural (a procedural, business-oriented language with a very rich runtime and non-procedural statements) and Java (an object-oriented, managed, multi-purpose language). Natural developers that need to maintain the new Java programs will need to be confident in their ability to recognize the business rules and correctly implement and test the changes they make after the migration is completed.

2.2. Key Traits of a Professional Software Conversion Tool

The key hallmarks of a professional language translation tool are the way it strikes a balance between three spectra of interest.

Astadia's Natural-to-C# conversion tools offer a solution to each of these areas:

Customization and Consistency

Anyone who has written software in a team before knows any program can be written in a variety of ways. A professional software conversion tool will provide the means to apply a configuration of customization options that suit the requirements of the customer.

These options can be simple things like naming conventions and the formatting of comments, or they can be more sophisticated, like the efforts the conversion tool will spend to detect and optimize structural or object oriented patterns that are inherent to the Natural code. At the same time, the tool should make it possible to manage such a configuration of customization options for a consistent application in an iterative process. These management facilities should also include configuration of other aspects of the migration like the translation of scripts, screens, or databases.

Maintainability by Natural and C# Developers

Consistent translation improves the maintainability of the generated C#, but for the C# code to be easy to work with for the Natural developers, it also needs to be based around simple design principles. This means the migration should generate code that provides (as much as possible) a 1:1 relationship between the number of lines of Natural code and the number of lines of C#, and keeps identifiers (variable names, program names, …) as close as possible to the original ones.

This will facilitate the recognition of business entities and rules in C# by the Natural developer. At the same time, C# developers with limited exposure to Natural should be able to pick up the programs and be optimally productive in the shortest possible timeframe. Simple design principles improve the understandability of the converted programs also by C# developers. Typical Natural constructs that are unknown in C# (like `REDEFINES`, or `ESCAPE` handling, or `BREAK` handling) are transformed into their closest C#-style equivalents.

Functional Equivalence with Natural and Full Support for the Target Platform

The basic requirement of a conversion tool is that the programs it produces are 100% functionally equivalent (including side effects encountered at runtime) with the original Natural programs. At the same time, the code that is produced should enable full use of the richness of the target platform.

This means some Natural language syntax will be replaced with calls to .NET APIs. It also means that the code should be fully usable in Visual Studio and support execution in debug mode. And also, the converted programs should easily be integrated with newly written .NET programs and vice versa.

2.3. Flexible Migration Tools

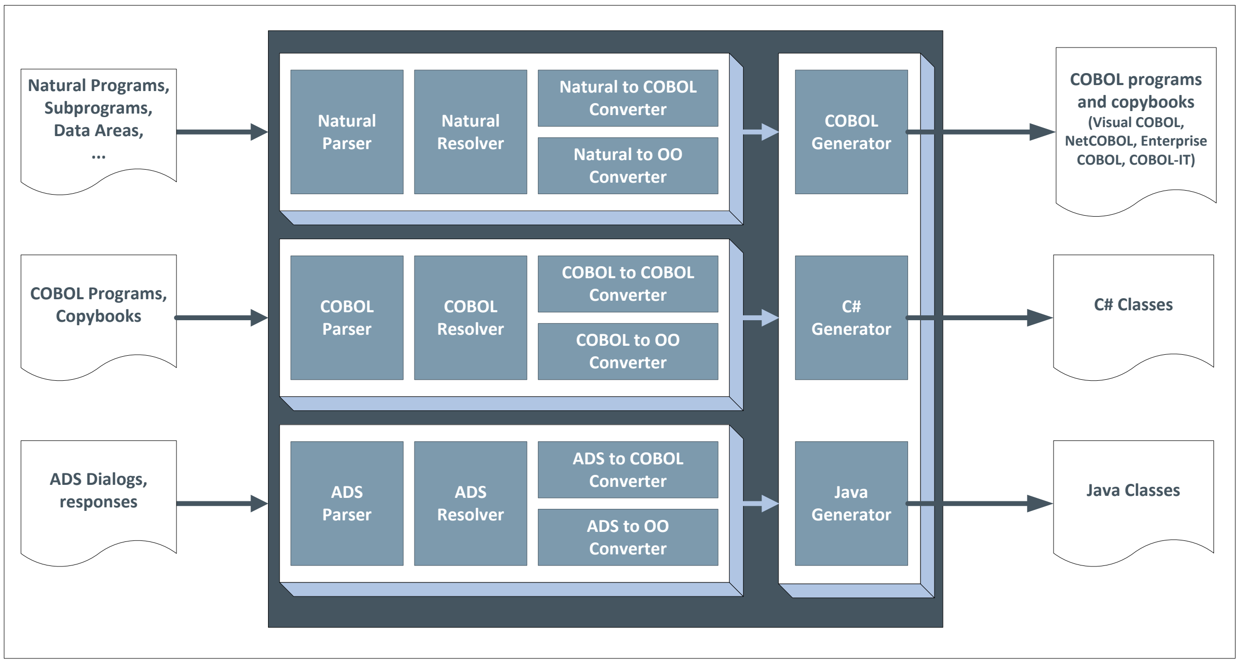
2.3.1. Rationale

For conversions from one programming language to another, Astadia has built a set of tools, collectively called the Language Convertor, with one important consideration in mind: each customer is different, and each migration is different. Every organization for example has its own development and design standards, patterns, and frameworks. Some prefer all data access in a separate layer, others choose for embedded data access.

When migrating from Natural to C#, some organizations will prefer to keep the resulting C# relatively close to the original Natural, for reasons of readability and maintainability by the existing developers. Other organizations will choose a more radical approach and prefer C# that uses more Object-Oriented design patterns.
In order to be able to accommodate these considerations, standards, and frameworks the Astadia tools are parameterized and customized for each project.

To address the stringent requirements listed in the above two sections, Astadia have selected a modular approach to building migration tools for all of its supported source platforms (Natural, IDMS, COBOL, …).
For language-to-language conversion tools, such as Natural to C#, the architectural overview of the internals of these tools looks like this:

- A parser that supports the complete Natural syntax.
- A resolver that links together the AST[3] that is produced by the parser with control-flow and data-usage information.
- Conversion rules for Natural syntax, from single statements to complex patterns of code, implemented in function of the target language.
- Code generators for Java, C#, and various COBOL dialects, enabling any desired coding style.



[3] Abstract Syntax Tree, a hierarchical representation of source code

2.3.2. Intelligent Conversion

To keep up with the design principle of a fully automated migration, conversion rules have been implemented in the Natural to OO Converter module to cover all possible edge cases. From time to time this could lead to relatively complicated C# code. Therefore, the conversion tools also recognize coding patterns (by static code analysis) that indicate where simpler, more elegant C# code can be generated while still staying 100% functionally equivalent to the Natural original.

Examples of such optimization can be found in the example code of the migration of the Natural loop statements to C#, and in the handling of the DECIDE statements.

3. Natural to C#: A High-Level Tour

This section first lists the main challenges involved in converting Natural to C#. Next, some examples will be shown of converted Natural code. These examples use the default conversion settings and are provided as-is, except that line breaks may have been added or removed in order for the code to fit on the page.

3.1. Challenges

When converting Natural, with both its procedural syntax and rich runtime, to a modern OO language such as C#, there are a lot of architectural and behavioral mismatches that need to be overcome:

- While Natural data items have an associated type (alphanumeric, date, logical, …), they can also be redefined and treated as a completely difference data type. This makes it possible to store “illegal” values (for example, the value “HELLO” could be stored in an alphanumeric redefinition of a numeric field, resulting in an invalid[4] number).
  C# on the other hand is a strongly typed language, which makes it impossible to assign a number to a `string` variable or vice versa.

- Natural programs typically use copycodes as a means to re-use code in multiple programs. C# on the other hand does not have the concept of an “include file”.

- Natural control-flow instructions, such as `REINPUT` and `RETRY`, simply don’t have a counterpart in C#.

- Some Natural control-flow instructions, such as `ESCAPE`, map rather nicely to exception handling, but this incurs an overhead that may not be desirable (e.g. if used in tight loops).

- Natural is known for its high-precision arithmetic (29 digits), which many financial applications rely upon and while C# has a “`decimal`” data type, its range is not sufficient[5]. The standard .NET library also doesn’t offer a “`BigDecimal`” type as an easy workaround.

- Natural programs can import database fields using only their name, with types taken from the corresponding DDM module. This means that a data dictionary change, such as extending a text field from 20 to 30 characters, does force a recompilation but no actual code change.

3.2. Target Framework

The migration can target.NET Framework 4.7.2 or later, .NET Core 3.1 or later, and/or .NET 5 or later. The plan is to retain support for .NET Framework for the foreseeable future (possibly moving to 4.8 as the only supported version), plus the two most recent LTS (long term support) releases of .NET Core.

[4] Strictly speaking, “HELLO” is a valid value for a Natural numeric field (format length “(N5)”); on EBCDIC systems, “HELLO” corresponds to hexadecimal value X'C8C5D3D3D6', and Natural would interpret this as a valid numeric value (-85336).
[5] It can only store 28 digits reliably; 29 digits can only be relied upon if the first two digits are no larger than 79.

3.3. Hello World

Natural program objects get mapped to C# classes that extend a corresponding base class (Program, Subprogram, Map, …); and the top-level program code is located in the “`Run()`” method.
A NaturalObject attribute is used to mark classes representing migrated program objects, in order to support dynamic steplib-based lookup at runtime (and to avoid any forced correlation between the namespace/class name and the original library/object name).

Let’s start with a simple example, a main program called HELLO from library WHITEPPR, which prints “HELLO WORLD”.

Natural

```
0010 WRITE 'HELLO WORLD'
0020 END
```

C#

```
using Astadia.NaturalServices;
using Astadia.NaturalServices.Objects;

namespace Customer.WhitePaper.Code.Programs {

  [NaturalObject("HELLO")]
  public sealed class HELLO : Program {

    protected override void Run() {
      Natural.Write().Text("HELLO WORLD").Execute();
    }

  }

}
```

Notes:

- The formatting of the C# code can be customized to a certain degree. For more advanced formatting rules, dedicated tools and/or IDE features are expected to be used. The samples in this document use the default settings.
- The namespace used for migrated code is configurable. The samples in this document will use a fixed prefix (“Customer”) and have the WHITEPPR library configured to use “WhitePaper” as package name. The final component of the namespace is a classification of the original Natural object (e.g. “Code.Maps” for maps, “DB” for DDMs, “Data.Parameter” for PDAs, …).
- The mapping for most Natural names (library name, object name, subroutine name, field name, …) is fully configurable, whether via general rules (e.g. “use lower camel case”) or explicit specification. The default for C# class names is “same as the Natural name, with invalid characters mapped to underscores”, which is why HELLO remained HELLO (and ABC-DEF would have become ABC_DEF).

The second example is a subprogram and includes a subroutine:

Natural

```
0010 DEFINE DATA
0020   PARAMETER USING MY-PDA
0030 END-DEFINE
0040 PERFORM WRITE-GREETING
0050 DEFINE SUBROUTINE WRITE-GREETING
0060   WRITE (2) #GREETING 'WORLD'
0070 END-SUBROUTINE
0080 END
```

C#

```
using System.Collections.Generic;

using Astadia.NaturalServices;
using Astadia.NaturalServices.Objects;

using MY_PDA = Customer.WhitePaper.Data.Parameter.MY_PDA_Area;

namespace Customer.WhitePaper.Code.Subprograms {

  [NaturalObject("HELLO2")]
  public sealed class HELLO2 : Subprogram {

    #region DEFINE DATA

    #region PARAMETER

    private readonly MY_PDA My_Pda = new MY_PDA();

    #endregion

    protected override IEnumerable<DataArea> DataAreas {
      get {
        // PARAMETER
        yield return this.My_Pda;
      }
    }

    #endregion

    protected override void Run() {
      Natural.Perform(this.Write_Greeting);
    }

    [InternalSubroutine("WRITE-GREETING")]
    private void Write_Greeting() {
      Natural.Write(2)
             .Field(this.My_Pda._Greeting)
             .Text("WORLD")
             .Execute();
    }

  }

}
```

Notes:

- The `DEFINE DATA PARAMETER` block is represented by a private field matching the type of the referenced PDA. Note that for data areas, the default name mapping includes an "_Area" suffix. This is because data areas commonly contain a group item of the same name as the area, which would otherwise cause clashes. An alias import is used so that only the "clean" name is actually used in code.
- The `DataAreas` property returns all areas set up via `DEFINE DATA`, which allows the framework to apply proper initialization and parameter matching behind the scenes.
- Region markers are used so that at maintenance time, large sections of the migrated application can be easily collapsed in the IDE.

- The mapping for the `PERFORM` statement uses a reference to the method created for the `DEFINE SUBROUTINE`, instead of calling it directly. This allows the framework to apply pre/post processing around the method call behind the scenes (e.g. handling `ESCAPE`s, updating system variables, …).
- The default mapping for field names is used; this converts “`#GREETING`” to “`_Greeting`”, because ‘`#`’ is not a valid identifier component in C#.

3.4. Additional Examples

Our extended Natural to C# Transformation Research Paper delves deeper into the main challenges of migrating Natural to C#, with examples of C# equivalents of transformed Natural code, with the tools configured to their current defaults which produce OO-style C# code, with many other mappings and configurations possible.

A more comprehensive program sample is also provided. Further examples include Natural loop statements, data items, the different types of basic control flow, and REINPUT and RETRY statements.

Request the Natural to C# Transformation Research Paper at info@astadia.com.

About Astadia

For nearly three decades, Astadia has performed mainframe modernization projects for government agencies and enterprises throughout the world:

- 300+ mainframe projects
- Billions of lines of COBOL transformed
- The Astadia FastTrack Factory: a world-class software platform that industrializes the refactoring of legacy applications
- Unparalleled access to mainframe modernization subject matter experts, architects, developers, engineers, and project managers
- Industry-leading migration success rates

Contacts us

info@astadia.com
www.astadia.com
US: +1 877 727 823
EU: +32 3 450 42 51