Software Renovation

Software Evolution — L4T3

Dr. Vadim Zaytsev aka @grammarware, February/March 2021

UNIVERSITY OF TWENTE. | Formal Methods & Tools

Realities of Working with Legacy

- Source code may be “lost”
- Documentation unavailable
- Old tech extended ad hoc
- Why does it work?
- Language cocktail
- Interoperability
- No choice, no design
- Painful details are painful


feature
programming languages

The Realities of Language Conversions

Billions of lines written in Cobol, PL/I, and other mature languages are still in active use. Many developers have tried to convert these languages to more modern ones, but few have succeeded. This article sheds light on the realities of language conversions and discusses the possibilities and limitations of automated language converters.

Andrey A. Terekhov, St. Petersburg State University
Chris Verhoef, Free University of Amsterdam

The most influential cost drivers of software engineering relate to management, personnel, and team capability—not software tools, although software vendors and academic researchers emphasize them. Many managers become victims of quack software modification tools and services. This mechanism, also known as the silver-bullet syndrome, is what anthropologists call *name magic*: you just say the name of the thing—“Cobol to Java”—and you have its full power at your disposal.

As the term *name magic* indicates, you don't need proof to support your claims. Anyone desperate for solutions will wholeheartedly accept those unsupported claims.[1] In fact, many software industry decision-makers find themselves trapped in huge amounts of legacy code, in dire need of modification. At the same time, software engineering educators deliver people skilled in contemporary development rather than enhancement programming, let alone geriatric care for aging legacy applications. Capers Jones[2] even thinks that this phenomenon is one of the 60 most important risks in software engineering. You do not have to be a rocket scientist to figure out that a language converter could solve your personnel problems: a language conversion is supposed to bridge the gap between available knowledge of people and the knowledge needed to solve legacy problems. Is it that easy? This article sheds light on the realities of language conversions; it also provides simple examples that expose the problems—anyone involved in software engineering can understand them.

So How Hard Is Language Conversion?

We encountered a company stating about a Cobol-to-Visual Basic converter, “The converter runs as a simple wizard; it is intended to be something a secretary can run.” Furthermore, at a panel of the International Conference on Software Maintenance, a researcher discussing the next century's challenges implied that automated conversion is solved and that the next challenge is paradigm-shift conversion. This means not only converting Cobol to C++ but also text-based procedural code into Web-enabled object-oriented C++. From such observations, we

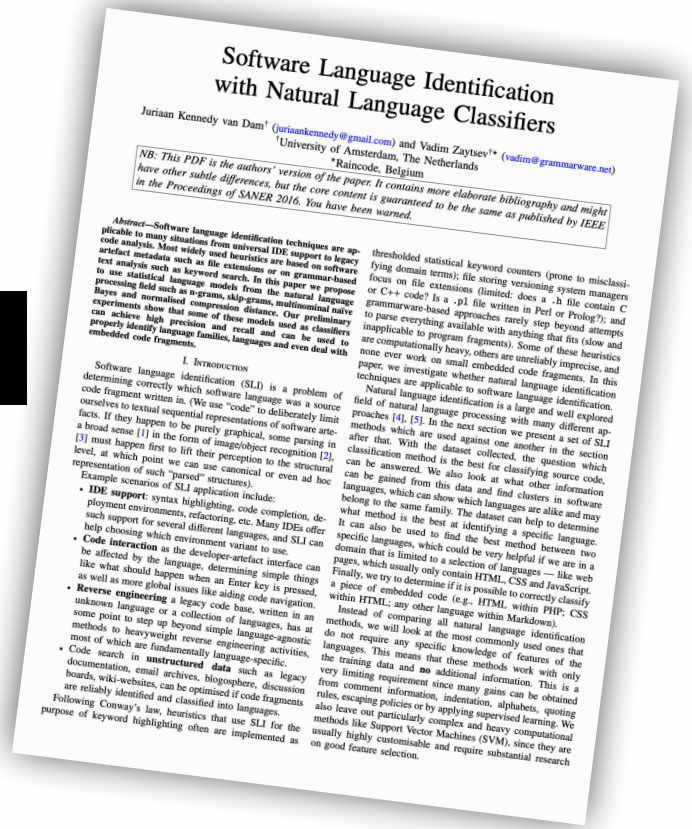
A.A.Terekhov, C.Verhoef, *The Realities of Language Conversions*, IEEE Software, 2000

Path 1: Rewrite

- "Why don't we just rewrite it?!"
- Crucial question:
  - if it took 30 years to build, how much time will it take to rebuild?
- Pro
  - software engineering is more efficient now than 30 years ago
- Con
  - development is still tech-biased
  - reverse engineer before reimplementation
  - verification of equivalence
- Thus
  - possible if flexibility is tolerable
  - good for project parts (e.g., exotic languages)

Path 1 Example: [redacted]

- Raincode proof of concept with [redacted]
  - 10k+ files
  - 50M+ LOC
  - names like “WMGCLP9M”
  - no extensions
- Customer says “only COBOL”
- Language (+ library!) identification
  - file, grep, …
  - vocabulary
  - parsing/resolution attempts
  - machine learning



Software Language Identification with Natural Language Classifiers

Juriaan Kennedy van Dam† (juriaankennedy@gmail.com) and Vadim Zaytsev†* (vadim@grammarware.net)
†University of Amsterdam, The Netherlands
*Raincode, Belgium

NB: This PDF is the authors' version of the paper. It contains more elaborate bibliography and might have other subtle differences, but the core content is guaranteed to be the same as published by IEEE in the Proceedings of SANER 2016. You have been warned.

Abstract—Software language identification techniques are applicable to many situations from universal IDE support to legacy code analysis. Most widely used heuristics are based on software artefact metadata such as file extensions or on grammar-based text analysis such as keyword search. In this paper we propose to use statistical language models from the natural language processing field such as n-grams, skip-grams, multinominal naïve Bayes and normalised compression distance. Our preliminary experiments show that some of these models used as classifiers can achieve high precision and recall and can be used to properly identify language families, languages and even deal with embedded code fragments.

I. Introduction

Software language identification (SLI) is a problem of determining correctly which software language was a source code fragment written in. (We use "code" to deliberately limit ourselves to textual sequential representations of software artefacts. If they happen to be purely graphical, some parsing in a broad sense [1] in the form of image/object recognition [2], [3] must happen first to lift their perception to the structural level, at which point we can use canonical or even ad hoc representation of such "parsed" structures).

Example scenarios of SLI application include:

- **IDE support**: syntax highlighting, code completion, deployment environments, refactoring, etc. Many IDEs offer such support for several different languages, and SLI can help choosing which environment variant to use.
- **Code interaction** as the developer-artefact interface can be affected by the language, determining simple things like what should happen when an Enter key is pressed, as well as more global issues like aiding code navigation.
- **Reverse engineering** a legacy code base, written in an unknown language or a collection of languages, has at some point to step up beyond simple language-agnostic methods to heavyweight reverse engineering activities, most of which are fundamentally language-specific.
- Code search in **unstructured data** such as legacy documentation, email archives, blogosphere, discussion boards, wiki-websites, can be optimised if code fragments are reliably identified and classified into languages.

Following Conway's law, heuristics that use SLI for the purpose of keyword highlighting often are implemented as thresholded statistical keyword counters (prone to misclassifying domain terms); file storing versioning system managers focus on file extensions (limited: does a `.h` file contain C or C++ code? Is a `.pl` file written in Perl or Prolog?); and grammarware-based approaches rarely step beyond attempts to parse everything available with anything that fits (slow and inapplicable to program fragments). Some of these heuristics are computationally heavy, others are unreliably imprecise, and none ever work on small embedded code fragments. In this paper, we investigate whether natural language identification techniques are applicable to software language identification.

Natural language identification is a large and well explored field of natural language processing with many different approaches [4], [5]. In the next section we present a set of SLI methods which are used against one another in the section after that. With the dataset collected, the question which classification method is the best for classifying source code, can be answered. We also look at what other information can be gained from this data and find clusters in software languages, which can show which languages are alike and may belong to the same family. The dataset can help to determine what method is the best at identifying a specific language. It can also be used to find the best method between two specific languages, which could be very helpful if we are in a domain that is limited to a selection of languages — like web pages, which usually only contain HTML, CSS and JavaScript. Finally, we try to determine if it is possible to correctly classify a piece of embedded code (e.g., HTML within PHP; CSS within HTML; any other language within Markdown).

Instead of comparing all natural language identification methods, we will look at the most commonly used ones that do not require any specific knowledge of features of the languages. This means that these methods work with only the training data and **no** additional information. This is a very limiting requirement since many gains can be obtained from comment information, indentation, alphabets, quoting rules, escaping policies or by applying supervised learning. We also leave out particularly complex and heavy computational methods like Support Vector Machines (SVM), since they are usually highly customisable and require substantial research on good feature selection.

Kennedy van Dam, Zaytsev. *Software Language Identification…*, SANER 2016.

Path 2: Refactor

- Tech is bad but code doesn't have to
- Massive refactoring is possible
  - GO TO elimination
  - objectification
  - wrapping
- Pro
  - possible to automate
  - can use transformation languages
- Con
  - stay with the same tech (best case: upgrade)
- Thus
  - often feasible, sometimes satisfactory

Path 2 Example: mBank

- Raincode project with a Polish bank
  - 37405 GO TOs
  - 99% removed
  - 203 GO TOs left
  - quality code as output
- Focus on quality up front
  - devs happy
  - analysts happy
  - owners happy

Quality First! A Large Scale Modernisation Report

Leszek Włodarski
mBank, Warsaw, Poland
Leszek.Włodarski@mbank.pl

Boris Pereira, Ivan Povazan, Johan Fabry, Vadim Zaytsev
Raincode Labs, Brussels, Belgium
firstname@raincodelabs.com

***Abstract*—Typically in modernisation projects any concerns for code quality are silenced until the end of the migration, to simplify an already complex process. Yet, we claim from experience that prioritising quality above many other issues has many benefits. In this experience report, we discuss a modernisation project of *mBank*, a big Polish bank, where bad smell detection and elimination, automated testing and refactoring played a crucial role, provided pay-offs early in the project, increased buy-in, and ensured maintainability of the end result.**

I. Introduction

It is always tempting to migrate the legacy system first, and then perform quality improvements with modern technology. *Raincode Labs* routinely takes on such projects, and observes how the wishes of the customers are mostly to migrate "as is" and *plan* to improve the quality later. There are mostly two linked risks here: (1) migrating bad code makes the migration trickier, and (2) the plans never realise themselves.

The project we report on in this paper, concerns migrating the central banking system of *mBank* (CBS from now on) from an old technological stack to the .NET Framework. We show that prioritising quality resulted in significant benefits up front and incomparably more maintainable code. Quality as a migration enabler is visible in three ways:

1) Showing early pay-offs at different levels, gaining support for the migration project.
2) Minimising migration-induced disruption since documentation and testing are deployed before migration.
3) Reducing maintenance effort of the legacy system, freeing resources for the migration as such.

We found that bad smell detection and removal (§ III) was met positively by the developers who acquired incentives to create better code to prevent future problems, and thus also saw the migration activities positively. ~3600 tests were written by developers and 680 by analysts, to aid maturity and comprehension of the legacy system (§ IV). Finally, refactoring out over 99% of 37405 `GOTO` statements (§ V), dramatically increased readability of the code.

II. Context

mBank is one of the top banks in Poland with 5.3M customers, 6.3% market share in loans and 5.9% market share in deposits. It was founded in 1986 and was the first fully-digital bank in Poland in 2000. Unlike many banks, *mBank* does not extensively rely on customising existing software packages. Instead, as a way of developing and maintaining competitive advantage, since 2005 *mBank* software is internally developed by several departments (300 FTE total). The codebase contains both banking-specific packages and common packages (*e.g.*, a CRM system) with a banking flavour. In this report we focus on the modernisation of the CBS of the corporate branch (~22'000 clients), which is separate from the retail branch.

Client products come with *hundreds* of degrees of freedom, stemming from different customer profiles: some only perform end-of-month wages payments (a large set of simple low-amount transactions, where a single failure is undesirable but acceptable); others perform transactions with much larger amounts, at unpredictable times, and failure or even delay in processing any of them, means losing the client. Such different profiles, the sheer number of features and the combinatorially explosive number of possible interactions among them has required extensive program management and manual testing.

Beyond day-to-day operations, the focus on customisability also has an impact on long-term survivability of the software. The major source of this impact is that 3MLOC of the core banking software is written in a fourth generation language (4GL) that we will call $\mathcal{B}$, detailed to some extent below.

The platform developed at *mBank* is based on a MultiValue database system [1], which is a data model predating relational databases. In short, MultiValue databases can be viewed as relational multidimensional databases where columns can hold lists of values instead of only atomic values. There have been multiple MultiValue databases available on the market, each with its own set of tools and/or 4GLs to aid in development of applications that use the database. For legal reasons, we cannot disclose which product is in use at *mBank*, and call it $\mathcal{M}$. The following two elements of this system are relevant:

1) $\mathcal{B}$ is an imperative, dynamically typed 4GL with automatic memory management. It has structured programming features like `LOOP` and `FOR` statements, plus simpler control flow statements like `GOTO` and `GOSUB`. It has strings, numbers and dynamically sized arrays.
2) $\mathcal{U}$ is a UI and application server that allows $\mathcal{B}$ programs to communicate with a GUI written in a more modern language through a stateful conversational telnet session.

The flexibility of $\mathcal{M}$ and simplicity of $\mathcal{B}$ allowed continual product development for 20+ years by people of varying skill sets, for different banks. As a consequence, the production code is inconsistent, written by people who no longer work at *mBank*, and is of varying quality.

Quality issues were made evident by an incident in production on a vast scale, which caused the bank to stop processing orders for several hours. There were multiple failures in the

Włodarski, Pereira, Povazan, Fabry, Zaytsev. *Quality First!*, SANER 2019.

Path 3: Dethrone

raincode LABS
compiler experts

- Run the generator for the last time
- Transform generated code
- Pro
  - full automation
  - language retirement
  - represents semantics perfectly
- Con
  - must have a powerful refactoring facility
  - badly generalisable across 4GL
- Thus
  - kills one 4GL at a time

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

```
PARA-1-EVALUATE.
*----------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

raincode LABS
compiler experts

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
     IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
     CONTINUE
*
     ELSE
     IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
     PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
     ELSE
     IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
     IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
     PERFORM PARA-0281559788-QDELETE THRU
         PARA-0281559788-QDELETE-EXIT
*
     ELSE
     IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
     IF PROCESSQ-FLAG = 'Y'
     PERFORM PARA-0281559788-QDELETE THRU
         PARA-0281559788-QDELETE-EXIT
     END-IF
*
     ELSE
     IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
     IF PROCESSQ-FLAG = 'Y'
     CONTINUE
     END-IF
*
     END-IF
     END-IF
     .

PARA-0281558749-EVALUATE-EXIT.
     EXIT.
```

```
PARA-1-EVALUATE.
*---------------

     EVALUATE TRUE
       WHEN V2PRESENT NOT = 'Y'
         CONTINUE
       WHEN V1PRESENT = 'N'
         PERFORM PARA-2
       WHEN V1PRESENT NOT = 'Y'
         CONTINUE
       WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
         OR ( CASCADE1 = 'Y'
           AND CASCADE2 = 'N'
           AND PROCESSQ-FLAG = 'Y' )
         PERFORM PARA-3-QDELETE
       WHEN OTHER
         CONTINUE
     END-EVALUATE
     .
```

Consequent paragraphs

raincode LABS
compiler experts

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

Input arguments

```
PARA-1-EVALUATE.
*----------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

raincode LABS
compiler experts

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

Actions performed

```
PARA-1-EVALUATE.
*----------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

raincode LABS
compiler experts

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

```
PARA-1-EVALUATE.
*---------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

Visually indented
Single level
Logic preserved

3 levels of nesting
No visual structure

raincode LABS
compiler experts

Path 3 Example: PACBASE

raincode LABS
compiler experts

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 =
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

```
PARA-1-EVALUATE.
*---------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    D-EVALUATE
```

Harmful constructs

Maintainable alternatives

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

Duplication

```
PARA-1-EVALUATE.
*----------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

raincode LABS
compiler experts

Path 3 Example: PACBASE

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

Boilerplate

```
PARA-1-EVALUATE.
*----------------

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

raincode LABS
compiler experts

Path 3 Example: PACBASE

raincode LABS
compiler experts

```
PARA-0281558749-EVALUATE.
    IF V1PRESENT = 'Y' AND V2PRESENT = 'N'
    CONTINUE
*
    ELSE
    IF V1PRESENT = 'N' AND V2PRESENT = 'Y'
    PERFORM PARA-0281559787 THRU PARA-0281559787-EXIT
*
    ELSE
    IF V1PRESENT = 'Y' AND V2PRESENT = 'Y'
    IF CASCADE1 = 'Y' AND CASCADE2 = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
*
    ELSE
    IF CASCADE1 = 'Y' AND CASCADE2 = 'N'
    IF PROCESSQ-FLAG = 'Y'
    PERFORM PARA-0281559788-QDELETE THRU
        PARA-0281559788-QDELETE-EXIT
    END-IF
*
    ELSE
    IF CASCADE1 = 'N' AND CASCADE2 = 'Y'
    IF PROCESSQ-FLAG = 'Y'
    CONTINUE
    END-IF
*
    END-IF
    END-IF
    .

PARA-0281558749-EVALUATE-EXIT.
    EXIT.
```

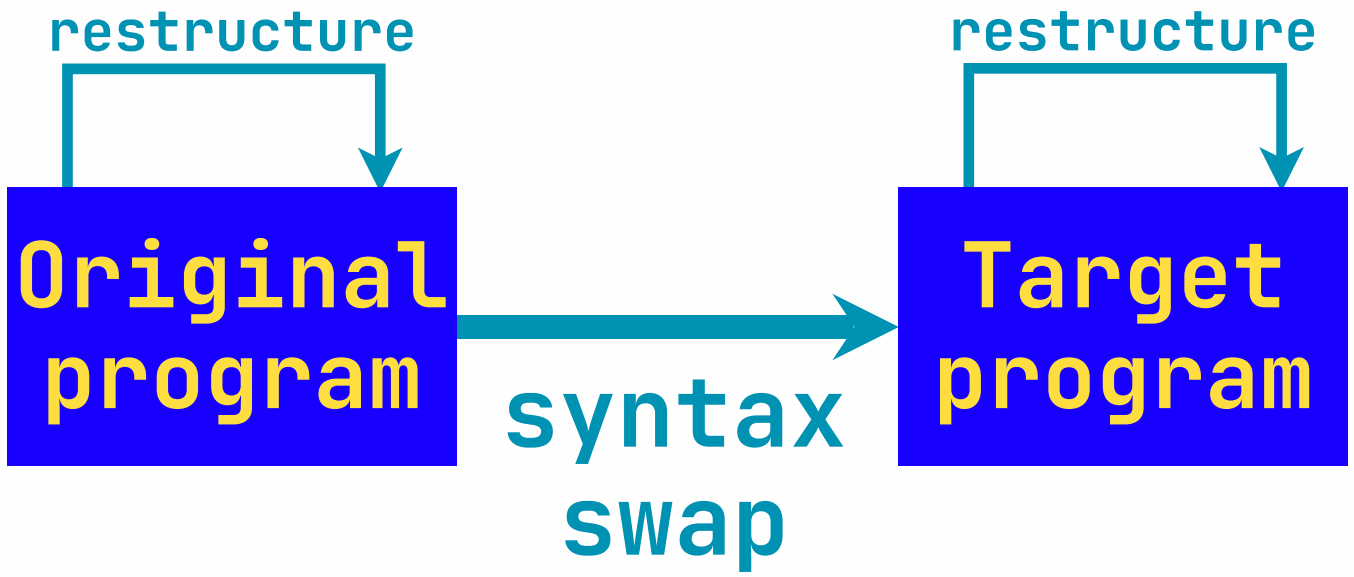
Names ☹

```
PARA-1-EVALUATE.

    EVALUATE TRUE
      WHEN V2PRESENT NOT = 'Y'
        CONTINUE
      WHEN V1PRESENT = 'N'
        PERFORM PARA-2
      WHEN V1PRESENT NOT = 'Y'
        CONTINUE
      WHEN (CASCADE1 = 'Y' AND CASCADE2 = 'Y')
        OR ( CASCADE1 = 'Y'
          AND CASCADE2 = 'N'
          AND PROCESSQ-FLAG = 'Y' )
        PERFORM PARA-3-QDELETE
      WHEN OTHER
        CONTINUE
    END-EVALUATE
    .
```

Names ☺

Path 4: Migrate



- Change the language
- Keep the logic
- Pro
  - sounds perfect
- Con
  - must restructure the original
  - must restructure the target
  - syntax swap must be feasible
  - native constructs map to simulated ones
- Thus
  - feasible under specific circumstances

Path ~~2~~4 Example: mBank

- Raincode project with a Polish bank
  - 37405 GO TOs
  - 99% removed
  - 203 GO TOs left
  - 499 due to translation
  - quality C# code as output
- Focus on quality up front
  - devs happy
  - analysts happy
  - owners happy

Quality First! A Large Scale Modernisation Report

Leszek Włodarski
mBank, Warsaw, Poland
Leszek.Wlodarski@mbank.pl

Boris Pereira, Ivan Povazan, Johan Fabry, Vadim Zaytsev
Raincode Labs, Brussels, Belgium
firstname@raincodelabs.com

***Abstract*—Typically in modernisation projects any concerns for code quality are silenced until the end of the migration, to simplify an already complex process. Yet, we claim from experience that prioritising quality above many other issues has many benefits. In this experience report, we discuss a modernisation project of *mBank*, a big Polish bank, where bad smell detection and elimination, automated testing and refactoring played a crucial role, provided pay-offs early in the project, increased buy-in, and ensured maintainability of the end result.**

I. Introduction

It is always tempting to migrate the legacy system first, and then perform quality improvements with modern technology. *Raincode Labs* routinely takes on such projects, and observes how the wishes of the customers are mostly to migrate "as is" and *plan* to improve the quality later. There are mostly two trickier, and (2) the plans never realise themselves.

The project we report on in this paper, concerns migrating the central banking system of *mBank* (CBS from now on) from an old technological stack to the .NET Framework. We show that prioritising quality resulted in significant benefits up front and incomparably more maintainable code. Quality as a migration enabler is visible in three ways:

1) Showing early pay-offs at different levels, gaining support for the migration project.
2) Minimising migration-induced disruption since documentation and testing are deployed before migration.
3) Reducing maintenance effort of the legacy system, freeing resources for the migration as such.

We found that bad smell detection and removal (§ III) was met positively by the developers who acquired incentives to create better code to prevent future problems, and thus also saw the migration activities positively. ~3600 tests were written by developers and 680 by analysts, to aid maturity and comprehension of the legacy system (§ IV). Finally, refactoring out over 99% of 37405 `GOTO` statements (§ V), dramatically increased readability of the code.

II. Context

mBank is one of the top banks in Poland with 5.3M customers, 6.3% market share in loans and 5.9% market share in deposits. It was founded in 1986 and was the first fully-digital bank in Poland in 2000. Unlike many banks, *mBank* does not extensively rely on customising existing software packages. Instead, as a way of developing and maintaining competitive advantage, since 2005 *mBank* software is internally developed by several departments (300 FTE total). The codebase contains both banking-specific packages and common packages (*e.g.*, a CRM system) with a banking flavour. In this report we focus on the modernisation of the CBS of the corporate branch (~22'000 clients), which is separate from the retail branch.

Client products come with *hundreds* of degrees of freedom, stemming from different customer profiles: some only perform end-of-month wages payments (a large set of simple low-amount transactions, where a single failure is undesirable but acceptable); others perform transactions with much larger amounts, at unpredictable times, and failure or even delay in processing any of them, means losing the client. Such different profiles, the sheer number of features and the combinatorially explosive number of possible interactions among them has required extensive program management and manual testing.

Beyond day-to-day operations, the focus on customisability also has an impact on long-term survivability of the software. The major source of this impact is that 3MLOC of the core banking software is written in a fourth generation language (4GL) that we will call $\mathcal{B}$, detailed to some extent below.

The platform developed at *mBank* is based on a MultiValue database system [1], which is a data model predating relational databases. In short, MultiValue databases can be viewed as relational multidimensional databases where columns can hold lists of values instead of only atomic values. There have been multiple MultiValue databases available on the market, each with its own set of tools and/or 4GLs to aid in development of applications that use the database. For legal reasons, we cannot disclose which product is in use at *mBank*, and call it $\mathcal{M}$. The following two elements of this system are relevant:

1) $\mathcal{B}$ is an imperative, dynamically typed 4GL with automatic memory management. It has structured programming features like `LOOP` and `FOR` statements, plus simpler control flow statements like `GOTO` and `GOSUB`. It has strings, numbers and dynamically sized arrays.
2) $\mathcal{U}$ is a UI and application server that allows $\mathcal{B}$ programs to communicate with a GUI written in a more modern language through a stateful conversational telnet session.

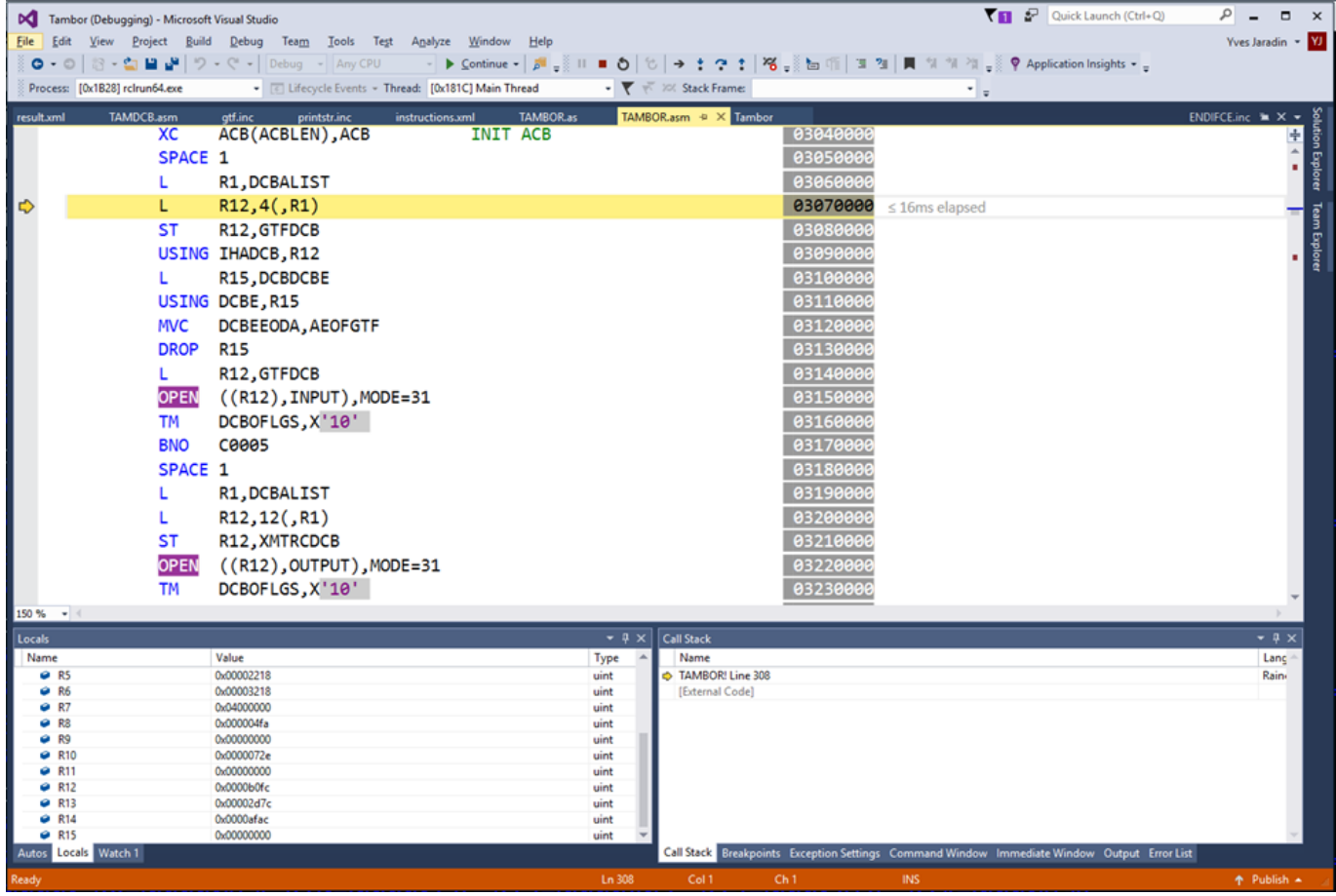
The flexibility of $\mathcal{M}$ and simplicity of $\mathcal{B}$ allowed continual product development for 20+ years by people of varying skill sets, for different banks. As a consequence, the production code is inconsistent, written by people who no longer work at *mBank*, and is of varying quality.

Quality issues were made evident by an incident in production on a vast scale, which caused the bank to stop processing orders for several hours. There were multiple failures in the

Włodarski, Pereira, Povazan, Fabry, Zaytsev. *Quality First!*, SANER 2019.

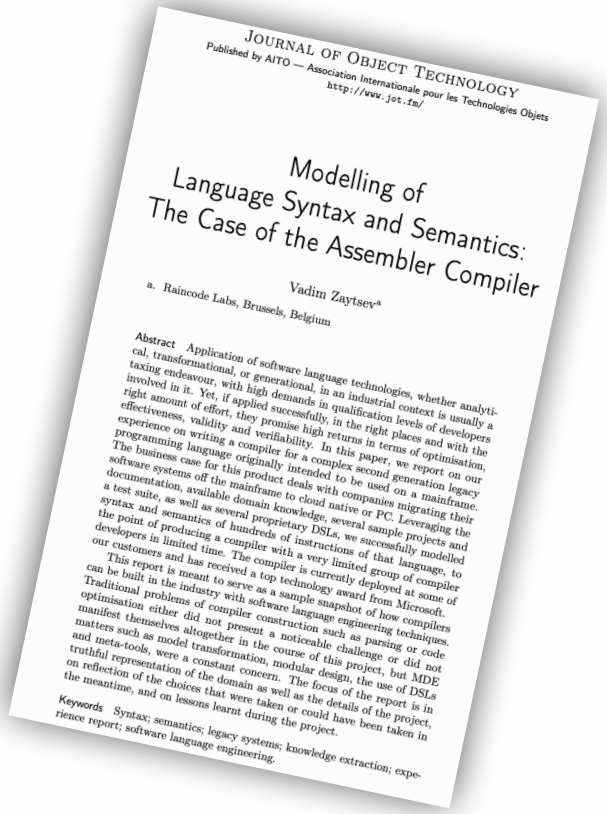
Path 5: Upgrade

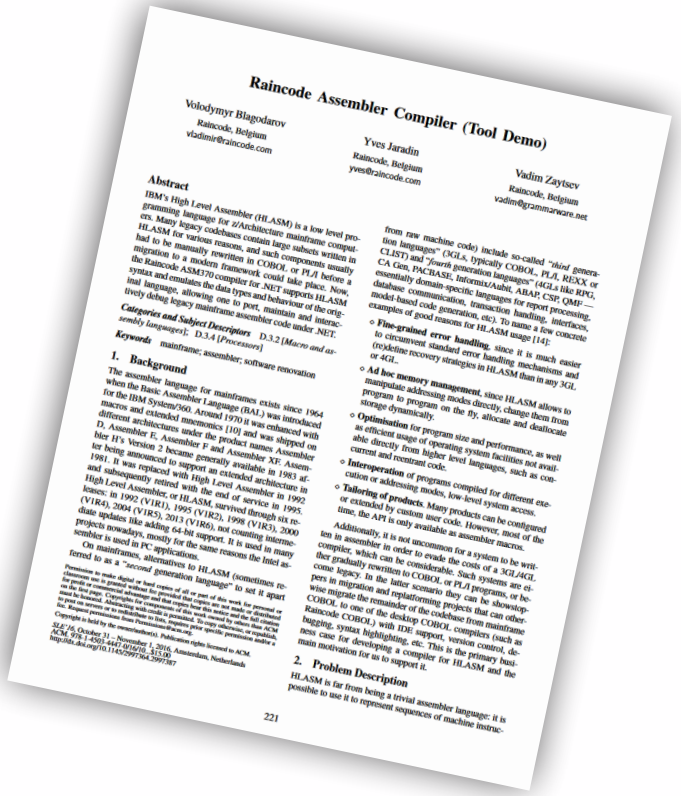
- Change the platform
- Keep *all* the rest
- Pro
  - no change to code
  - executes in new env
  - empowers later change
- Con
  - tech dependencies remain
  - hard to develop solutions
- Thus
  - can build a business



Path 5 Example: HLASM

- 2GL as a showstopper
- Pretend HLASM is a HLPL
  - 1568 pages of documentation
  - ~950 instructions
  - self-modifying code
  - powerful macro language
  - non-orthogonal design
  - model-driven (re)engineering
  - performance analysis
- Top Performer 2016 – MS



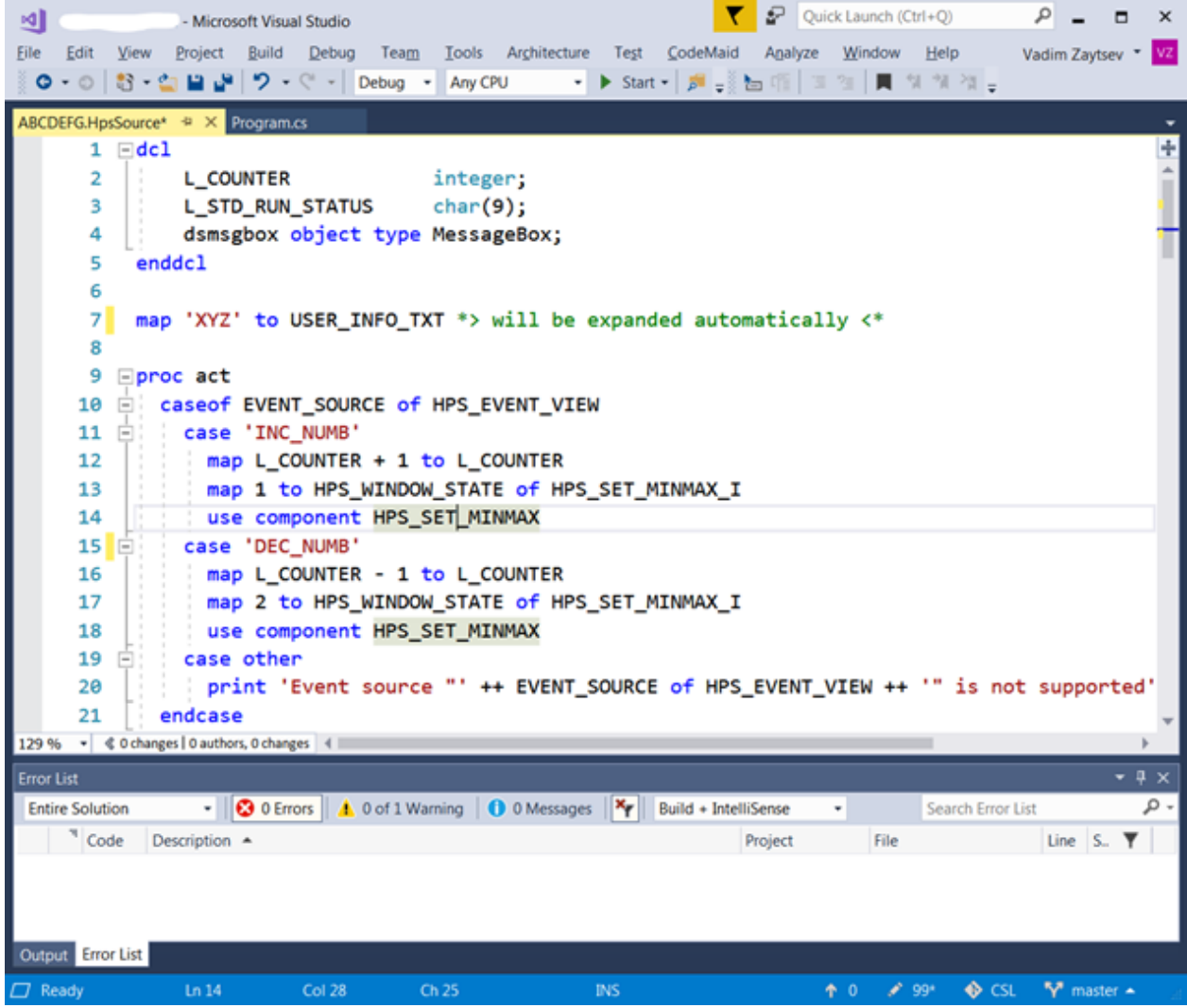


Blagodarov, Jaradin, Zaytsev. *Raincode Assembler Compiler*. SLE 2016.
Zaytsev, *The Case of the Assembler Compiler*. ECMFA 2020.

UNIVERSITY OF TWENTE.

Path 6: Reinvent

- Redevelop a compiler
- Cover language subset
  - base on customer's code
- Pro
  - code mostly stays
  - modern dev & exec envs
  - full control over target env
- Con
  - documentation may not exist
  - substantial effort
- Thus
  - perfect for rich client + expert migrator



V.Zaytsev, *The Secret Evolution of Bad Languages*, keynote @ SATToSE 2018.

Path 6 Example: TIALAA

```
$$FILE 06/07/2017 23:59:59
$$FOO    ABCD      Y 06/07/2017 23:59:59 XYZ
 A 1 00010 00 0000 Y Y N Y NAMEA     NAMEB     S
 C 2 00015 02 0000 Y Y Y Y NAMEDDDD NAME EEE S
 F 5 00030 00 0020 Y N N Y NAMEG     NAMEH     S
$$BAR    EFGHKLMN Y 06/07/2017 23:59:59 N/A
 A LONGER_NAME_FOR_ENTITY                 999 10.0
 A ANSWER_TO_THE_ULTIMATE_QUESTION         42  7.5
```

Parser Generation by Example for Legacy Pattern Languages

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadims@grammarware.net

Abstract

Most modern software languages enjoy relatively free and relaxed concrete syntax, with significant flexibility of formatting of the program/model/sheet text. Yet, in the dark legacy corners of software engineering there are still languages with a strict fixed column-based structure—the compromises of times long gone, attempting to combine some human readability with some ease of machine processing. In this paper, we consider an industrial case study for retirement of a legacy domain-specific language, completed under extreme circumstances: absolute lack of documentation, varying line structure, hierarchical blocks within one file, scalability demands for millions of lines of code, performance demands for manipulating tens of thousands multi-megabyte files, etc. However, the regularity of the language allowed to infer its structure from the available examples, automatically, and produce highly efficient parsers for it.

CCS Concepts • Software and its engineering → Programming by example; Translator writing systems and compiler generators; Parsers; • Theory of computation → Grammars and context-free languages; Pattern matching; Keywords parser generation, engineering by example, pattern languages, legacy software, grammar inference, language acquisition

ACM Reference Format:
Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17). ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3136040.3136058

GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17), https://doi.org/10.1145/3136040.3136058.

1 Problem

When working in legacy analysis and renovation industry, we come across binarre file formats with alarming regularity. It is a world where language identification cannot rely on file extensions and may require anything up to and including machine learning [20], and where dealing with a priori unknown formats has been elevated from an idle thought experiment to a routinely used job interview question [36]. In this paper, we will share a success story of handling one of such file formats, with the pattern language technology (terminology by Angluin [1])

Raincode Labs is an independent company providing bespoke compiler services. One of our clients in the banking sector, which, being NDA-bound, we will have to call ₰, owns a multi-million line codebase, developed over decades of company growth and containing most of its business rules and IT assets. Besides COBOL and PL/I which we have learnt to handle with ease, grace and experience, the codebase contains almost 70k modules in a fourth-generation language we will call ₰. Even though ₰ has over 100 developers actively creating new software in that language on a daily basis, it has been classified as a liability for the future and scheduled for retirement in its current incarnation. We are now in the process of writing a full-fledged compiler for ₰ targeting the .NET framework. When the project is completed, it will allow ₰ to deploy their products on commonplace hardware or modern platforms such as Azure, to write hand-tweaked components in modern programming languages such as C♯ and most importantly, to hire young professionals otherwise frightened off by the prospect of learning an obscure dying language as the first job requirement.

The documentation of ₰ is partly non-existent, partly outdated and ultimately protected legally by an explicit disclaimer that only paying customers of ₰'s current rights owner are allowed to read it. The source artefacts come in the form of five different serialisation languages that ₰'s infrastructure exports them in. These five notations are not synchronised: only one looks like a programming language, one more is more of a markup language, another one is syntactically and conceptually close to JSON, another one is LISP, and finally there is one notation with position-based strings (think Excel in ASCII, example on Figure 1). We will call the latter notation ¢. All five are important for the healthy functioning of the system, since they define data and

Zaytsev, *Parser Generation by Example for Legacy Pattern Languages*. GPCE 2017.

UNIVERSITY OF TWENTE.

Never has been

raincode LABS
compiler experts

Software renovation
is not easy

Conclusion

- Renovation is not easy
- There are ways
  - rewrite, refactor, dethrone, migrate, upgrade, retire
- Each way has pros and cons
  - best decision depends on the context
- Q&A Sessions @ Canvas
  - ⇒ v.zaytsev@utwente.nl
  - ⇒ https://discord.gg/n7VQAPNBPD