JOURNAL OF APPLIED QUANTITATIVE METHODS

Quantitative Methods Inquires

AUTOMATED SOFTWARE TRANSLATION – THEORETICAL BACKGROUND AND CASE STUDY

Robert ENYEDI [1]
PhD, CEO, Numiton Ltd.

E-mail: robert.e@numiton.com , **Web-page:** http://www.numiton.com

Abstract: *The necessity for software migration is presented. The concept of software migrator is introduced. A generality metric for software translations is proposed. The business feasibility of automated versus manual migration is studied. nTile PHPtoJava, a working software migrator, is reviewed.*

Key words: software translation; software migrator; programming languages; generality metrics; PHP; Java

1. Software Translation Overview

The evolution of programming languages has produced a wide variety of exponents, grouped into generations. Evolving computer hardware together with evolving operating systems have led to the emergence of new programming languages and to the marginalization of others. The O'Reilly History of Programming Languages poster [OREI07][2] displays more than 50 major programming languages that have been used in the last 50 years of software development, while Bill Kinnersley's Language List [KINN07] contains over 2500 known programming languages, ranging from the obscure to the widely used.

As a consequence, the allotment of specialists in different programming languages and architectures has changed over time. A programming language doesn't disappear, but once there are no tools to run it on new platforms, the language is less and less used and specialists are increasingly more difficult to find. This is the case of many COBOL dialects, for example.

Another cause of language outdating is that in time their features cover less and less of the current needs – lack of Web support is an example in this sense. That is why new and improved versions should be continuously developed for languages, their libraries and tools. When this does not happen, possibly because the vendor is out of business or shifts its priorities, some languages do get to a dead point. For instance, according to [REED07] the

future of SAP ABAP language is uncertain precisely because a more suitable language (Java) is available to develop code for SAP systems.

In such situations it becomes necessary to migrate application systems developed with outdated languages towards modern languages and technologies, more adequate for business needs and with support from established vendors. The migration process also offers the possibility of switching the underlying hardware and software platform (e.g. migration from mainframes to PCs).

While this migration can be accomplished manually by a complete rewrite of the application system, there are situations when an automated process is more suitable. Some reasons are detailed in section 4 Economical Feasibility. Automated migration is performed using specialized products called software migrators.

The concept of software migrator derives from the one of language migration, which refers to a language transformation without altering semantics. A software migrator is a tool that translates sentences written in a source language into the semantically equivalent sentences in a target language. It also emulates the semantic dependencies generated by the libraries of the application system.

The theoretical background of software migrators is a generalization of the compilers theory. In this context, a compiler becomes a particular case of software migrator that produces sentences in a low-level language (assembly or machine code).

Many issues that a software migrator must solve depend on the characteristics of the source and target languages. These characteristics include:

- generations: for example, migrating from a structured language to an object-oriented one must provide mechanisms for fully making use of encapsulation, abstraction, polymorphism and inheritance;
- type systems: migrating from a weakly/dynamically-typed language to a strongly/statically-typed one requires type inference algorithms;
- execution model: migrating from an interpreted language to a compiled one means that the migrator must handle dynamic constructs (variables, procedures, dependencies), error reporting paradigms etc.

This paper presents the theoretical background and business rationale behind the nTile PHPtoJava software migrator developed by Numiton Ltd. The design and implementation of this migrator had to take into account all three challenges mentioned above.

2. nTile PHPtoJava Architecture

PHP is a programming language used for developing most of the small to medium public Web sites. It is a weakly typed interpreted language with several dynamic capabilities. Up to version 4 it was a structured language, afterwards being endowed with object-oriented features.

Java is a general-purpose programming language, widely used for desktop, Web and mobile applications. It is a strongly typed compiled language, with a pronounced object-oriented character. The Java Enterprise Edition platform powers most of the medium to large business applications.

As the name suggests, nTile PHPtoJava is a software migrator from the PHP language to Java EE. Figure 1 presents its overall architecture.

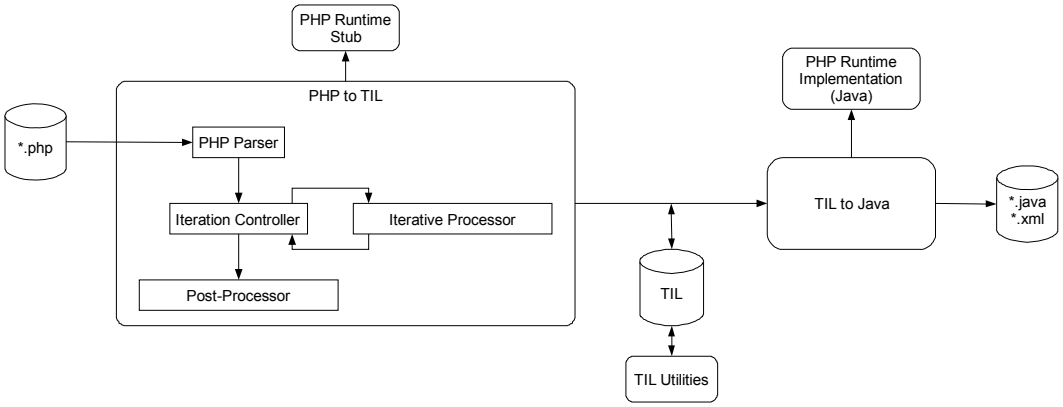

Figure 1. nTile PHPtoJava architecture

The software migrator is composed of two main modules: PHPtoTIL and PHPtoJava. PHP source files are processed by PHPtoTIL and stored in an intermediate representation called the **T**ranslation **I**ntermediate **L**anguage - **TIL**. TILtoJava then turns this intermediate representation into Java source files, packaged as a JavaEE application.

Further detailing the architecture, PHP2TIL is composed of several submodules, as follows:

1. PHP Parser – processes the source files based on the PHP grammar rules and builds a first version of the TIL representation. From this point on, all work is carried out on this representation;
2. Iterative Processor – the migration is performed in several iterations. Each iteration uses the already gathered information in order to further refine resolving of entities, type inference etc.
3. Iteration Controller – this component monitors iterations and decides when the gathered information can no longer be refined. Control is then transferred to the post-processor.
4. Post-Processor – performs some finishing touches on the intermediate representation, such as collecting translation statistics, and finalizing the type inference.

Throughout its operation PHPtoTIL collaborates with an utility module containing the interface of the PHP runtime, in order to have a fully resolved TIL model.

TILtoJava simply traverses the well-formed TIL representation and translates it into appropriate Java constructions. The Java abstract syntax tree (AST) is built in-memory, then saved as Java source code together with JavaEE support classes and XML descriptors. The Java implementation of the runtime is also linked to the final application.

A sample translation is presented below. The PHP input consists of the files *interpreterTest.php*, *included.php* and *included2.php*. This sample code performs a dynamic PHP include operation depending on the value of a HTTP session variable.

interpreterTest.php:

```
<?php
echo "Body -> interpreterTest.php\n";
$b = ".php";

if($_SESSION['selector']=="included") {
        $a = 'included';
} else {
        $a = 'included2';
}

$includeResult = require $a.$b;
echo "includeResult =
    $includeResult\n";

?>
```

included.php:

```
<?php
echo "Body -> included.php\n";
return 3;
?>
```

included2.php:

```
<?php
echo "Body -> included2.php\n";
return 10;
?>
```

The migrator produces the Java output consisting of the source files *interpreterTest.java*, *included.java*, *included2.java* and *GlobalVars.java*. For brevity, the XML descriptors and some support classes are not presented.

interpreterTest.java:

```
public class interpreterTest extends NumitonServlet {
  public Object generateContent(PhpWebEnvironment webEnv) throws IOException, ServletException {
    gVars.webEnv = webEnv;
    VarHandling.echo(gVars.webEnv, "Body -> interpreterTest.php\n");
    gVars.b = ".php";
    if (VarHandling.equals(gVars.webEnv, gVars.webEnv._SESSION.getValue(gVars.webEnv,
       "selector"), "included")) {
      gVars.a = "included";
    }
    else {
      gVars.a = "included2";
    }
    gVars.includeResult = new DynamicConstructEvaluator<Integer>() {
      public Integer evaluate() {
        Integer evalResult = null;
        if (VarHandling.equals(gVars.webEnv, gVars.a, "included")
            && VarHandling.equals(gVars.webEnv, gVars.b, ".php")) {
          evalResult = (Integer) PhpWeb.include(gVars, gConsts, example.included.class);
        }
        if (VarHandling.equals(gVars.webEnv, gVars.a, "included2")
            && VarHandling.equals(gVars.webEnv, gVars.b, ".php")) {
          evalResult = (Integer) PhpWeb.include(gVars, gConsts, example.included2.class);
        }
        return evalResult;
      }
    }.evaluate();

    VarHandling.echo(gVars.webEnv, "includeResult = "
        + VarHandling.strval(gVars.webEnv, gVars.includeResult) + "\n");
    return null;
  }

  public interpreterTest() {
  }
}
```

included.java:

```
public class included extends NumitonServlet {
  public Integer generateContent(PhpWebEnvironment webEnv) throws IOException, ServletException {
    gVars.webEnv = webEnv;
    VarHandling.echo(gVars.webEnv, "Body -> included.php\n");
    return 3;
```

```
  }

  public included() {
  }
}
```

included2.java:

```
public class included2 extends NumitonServlet {
  public Integer generateContent(PhpWebEnvironment webEnv) throws IOException, ServletException {
    gVars.webEnv = webEnv;
    VarHandling.echo(gVars.webEnv, "Body -> included2.php\n");
    return 10;
  }

  public included2() {
  }
}
```

GlobalVars.java:

```
public class GlobalVars extends GlobalVariablesContainer {
  public GlobalConsts              gConsts;

  public GlobalVars() {
  }

  public String       b;
  public String       a;
  public int           includeResult;
}
```

Apart from objectification, the above sample illustrates some of the advanced migration processes available in nTile TILtoJava:

- generating declarations and type inference for variables – the declarations of global variables *a*, *b* and *includeResult* are generated inside *GlobalVars.java*;
- transforming dynamic constructs into static ones, by performing static analysis – see the include expressions in *interpreterTest.java*.

Applying to the migration output the source lines of code software metric [LAIR06],but not counting non-relevant source code lines (emtpy lines, commented lines, lines containing empty braces etc.), the results from Table 1 are obtained.

Table 1. Effective source lines of code comparison

Translated Entity	***eLOC in PHP***	***eLOC in Java***
interpreterTest	7	24
included	2	6
included2	2	6
GlobalVars	N/A	6
TOTAL	**11**	**42**

The results underlines that comparing applications based on the number of lines of code is not a relevant metric, especially when the compared applications are written in different programming languages. The substantially larger Java migration output has a much improved clarity as well as maintainability than the much more compact original PHP code. Additionally, there is a fixed overhead an average Java source file has over a PHP source file, whose overall percentage decreases as the source file gains in complexity.

3. Generality Metrics of Software Migrators

The design and implementation of a software migrator is a complex task. The migrator should ideally cover all variations of the source language syntax and semantics, and be able to translate them into optimal target language constructs. Nevertheless, a partially developed migrator could successfully translate a well chosen set of applications. An incremental approach to development is thus possible, each iteration increasing the coverage degree of the source language.

In order to track progress and have a decisional basis for the features of each iteration, some sort of metric must be devised. In [ENYE04] a generality metric for software migrators is proposed, referring to the coverage degree of source language constructions and runtime libraries.

Source language constructions are described in the language's grammar and include program structures, data and instruction definitions. Measurement of the generality degree for software migrators takes into consideration the following aspects:

a) The coverage degree for constructions and functionalities of the source languageThis degree does not only refer to the number of distinct constructions and functionalities. In the case of complex migrators, the implementation coefficient of each source language feature must be taken into account. A situation where some constructions are partially supported may appear during the development iterations. Determining the implementation coefficient is done by the migrator's development team, taking into account for example man-days or cost for the current implementation and the estimated effort to completion.

b) The importance coefficient of each constructionThe importance coefficient of a language construction is also determined by the translator's developers and is based on the average usage frequency of the construction in a representative set of applications.

c) The translation degree of runtime librariesUsually, apart from language constructions a software application uses a standard program library and possibly a set of third-party libraries. Translating an application also requires translating these libraries. A difficulty arises when the source code for third-party libraries is not available. Even more, for many languages not even the source code of the standard library is available. When the source code of a library is available, the translation degree of the library does not influence the migrator's generality. This is because the same migrator is used to automatically translate the library as well as the application. When the source code of a library is not available, automated translation cannot be accomplished. Translating the behavior of the source language library into an equivalent behavior in the target language becomes a manual operation. Because the standard library is used by any application written in the source language, it is mandatory that the generality metric include the translation degree of the standard library.

Taking these factors into consideration, the proposed generality metric formula is:

$$G_{TF} = C_L \cdot \frac{\sum_{i=1}^{n} CoImp_i \cdot CoImpl_i}{ConT} + (1 - C_L) \cdot G_T B_{st}, CoImp_i \in |0;1|, CoImpl_i \in |0;1|$$

where:

G_{TF}	– the software migrator generality degree,
$CoImp_i$	– the importance coefficient for each construction of the source language,
$CoImpl_i$	– the implementation coefficient of each source language construction,
$ConT$	– the total number of constructions in the source language,
C_L	– the importance coefficient of the language,
G_TB_{st}	– the translation degree of the standard library.

The first term of the formula represents the coverage degree of the language constructions.

The C_L importance coefficient is the translation generality ratio in relation with the translation generality of the standard library. It depends on the source language. When the source code of the standard library is available, C_L has maximum value 1 because the translator's generality is entirely conditioned by the coverage degree of the language constructions. This ideal situation does not often occur, because standard libraries usually have proprietary implementations. Availability might also depend on the licensing model for the source code of the standard library.

The development of software migrators should aim to increase the generality metric as much as it is economically feasible, the development costs are justified. Other aspects, such as execution speed and memory requirements of the software migrator, should also be put in balance.

Applying the above generality metric formula to the nTile PHPtoJava software migrator means analyzing the structure of the PHP language and assigning suitable values to the influencing factors.

With respect to source language constructs, Table 1 contains the importance and implementation coefficients that have been determined at a certain point in the development cycle of the migrator.

Table 2. PHP source language constructs

Source Language Construct	***Importance Coefficient***	***Implementation Coefficient***
Top-Level Constructs		
Class	1	1
Class Field	1	1
Function and Class Method	1	1
Source File	1	1
Statements		
BREAK/CONTINUE	1	0.7
Compound	1	1

Source Language Construct	*Importance Coefficient*	*Implementation Coefficient*
Display	1	1
DO-WHILE	1	1
Exception Handling	0.7	1
Expression Container	1	1
FOR	1	1
FOR-EACH	1	1
LIST	0.7	0.5
Multiple Decision (SWITCH)	1	1
RETURN	1	1
Simple Decision (IF-THEN-ELSE)	1	1
WHILE	1	1
Expressions		
Array	1	1
Binary	1	1
Class Instantiation	1	1
Dynamic Function Call	0.5	0
Dynamic Include	0.9	0
Dynamic Variable Reference	0.5	0
Function Call	1	0.9
Include	1	1
Literal	1	1
Ternary Conditional	1	1
Type Cast	1	1
Type Test (INSTANCEOF)	1	1
Unary	1	1
Variable Reference	1	1

Dynamic constructs - variables, function calls and includes - are not supported yet. Dynamic variables and function calls are not widely-used in PHP programs and therefore have a smaller importance coefficient. Dynamic includes however should be considered a priority in subsequent development iterations, because they are used very frequently.

Partially implemented language constructs are BREAK/CONTINUE statements (no nesting level jumps), LIST statements (no character strings can be used as assigners) and function call expressions (no optional arguments are supported).

The standard PHP library is divided into function groups. The implementation coefficient of each function group is presented in Table 2.

Table 3. PHP runtime implementation details

Function Group	***Total***	***Implemented***	***Implementation Coefficient***
Arrays	75	36	0.48
Date and Time	37	5	0.14
Directories	9	3	0.33
Error Handling and Logging	11	1	0.09
Filesystem	78	27	0.35
Function Handling	11	1	0.09
Mail	2	1	0.5
Mathematical	48	11	0.23
Miscellaneous	25	3	0.12
Network	32	4	0.13
Output Control	17	6	0.35
PHP Options&Information	47	10	0.21
POSIX Regex	7	4	0.57
Sockets	25	1	0.04
Strings	95	39	0.41
URLs	10	6	0.6
Variables handling	36	28	0.78
Zlib Compression	22	4	0.18
TOTAL	**587**	**190**	**0.32**

The more frequently used library functions have taken priority when developing the migrator's runtime support. Also, some other implemented functions are are not taken into account, as they belong to non-standard library extensions (MySQL, FTP, image handling).

According to these particularizations, the value of the generality metric for nTile PHPtoJava becomes:

$$G_{TF}=0.6\cdot\frac{26.65}{31}+(1-0.6)\cdot 0.32=0.64$$

This value can be increased by further development of the migrator, notably by implementing the dynamic constructs and by supporting a larger number of PHP runtime library functions. The current functionality is sufficient however for translating numerous real-life PHP projects.

4. Economical Feasibility

The need to migrate a software application to a new programming language appears in several cases. One of these cases is that the code base of the application has outgrown the possibilities of the source language and its tools, maintenance and development of new features becoming problematic. Another scenario is that vendor support is no longer satisfying or that the specialists for the source language have become/are about

to become scarce. Usually all these aspects are interlinked and tend to occur simultaneously, due to the inherent life-cycle of programming languages.

Figure 2 illustrates vendor support and specialists' availability over time for a typical programming language.

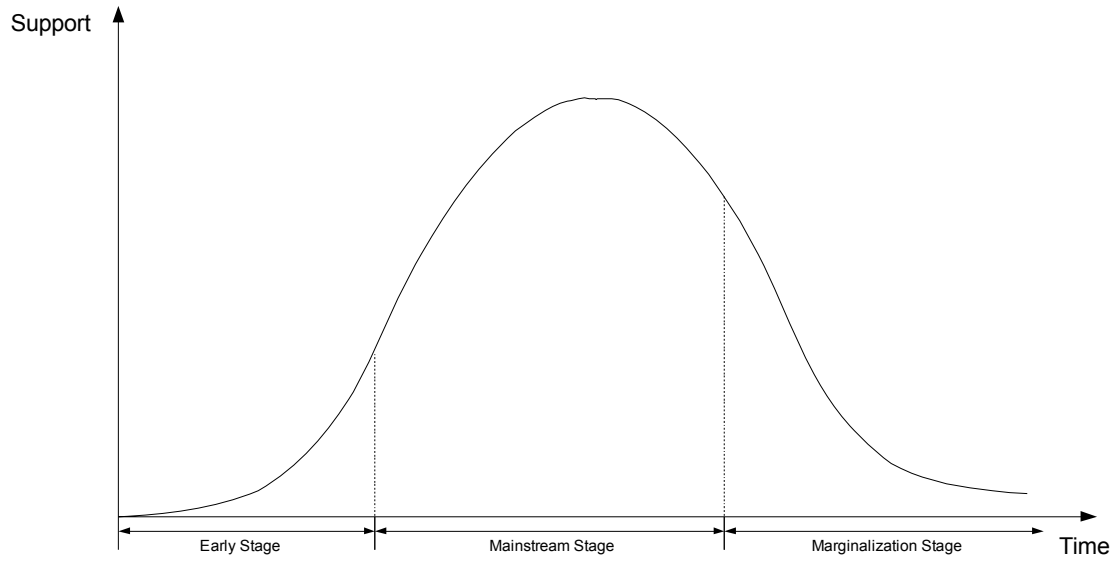

Figure 2. Programming language life-cycle

Once the need to migrate has been established, a matter to be thoroughly considered is that of the target language. The most suitable type of language is usually a general-purpose one, with high expressivity, well-supported by vendors. The available tools should be sophisticated enough to allow efficient control over the code base. Finally, the specialists with a suitable skill degree should be readily available.

The straightforward way of migrating a software application is to perform a complete manual rewrite. This approach presents several advantages:

- possibility of application redesign, having a better understanding of the business domain;
- optimal use of the technologies available for the target language;

There are however disadvantages as well:

- significant effort, as this involves a complete development life-cycle for what is in essence a new application: financial costs, long period of time, significant allocation of human resources;
- the inherent bugs that are produced by any software development process, even if most bugs in the original application had been detected and resolved throughout the application's usage over time;
- stakeholders will probably make pressures to add new features while rewriting the application; this multiplies the risks of defects;
- new features, or even existing features that are altered by the rewrite, might change the user experience and cause learning difficulties; users are normally capable of dealing with few changes at a time, but not with many/radical ones – affecting their productivity and incurring training costs.

Automated translation using a software migrator is by default limiting the scope of the change to porting existing functionality without adding new features. This can be done at a later stage, benefiting from all the advantages of the target language: refactoring support, more powerful technologies and tools.

Among the advantages of automated software translation are:

- lower effort, as the analysis, design and implementation processes are not executed; only testing and deployment need to be performed;
- as the existing functionality is closely reproduced, no application-specific bugs are introduced into the software;
- the usability closely matches that of the original application; even if the user interface changes (e.g. transform a text-based interface into a GUI), there is a close correspondence between each interface element of the old and new application;
- the back-end of the application can be re-engineered using optimization algorithms during the migration process; being automated rather than applied manually, the result is uniform and less prone to bugs;
- by-products of the migration process include detailed information about the structure and flows of the application, information that can be used by implementing analysis tools.

Inherent disadvantages of automated software migration are:

- not taking full advantage of the technologies available for the target language, since a generic automated process cannot capture project-specific optimization nuances as well as the human mind; refactoring can be efficiently performed a later stage though;
- all bugs that are still undetected in the original application will be ported into the new one (these are probably few in number and unimportant since the original application had been in use for a significant period of time);
- new bugs could be introduced into the new application, due to bugs in the software migrator itself; this risk can be minimized by thorough testing of the migrator and by test harnesses/pilot migration projects to check the translation of individual constructs.

These disadvantages can be countered by implementing software migrators that are customizable for specific translation projects. Individual particularities of each application can be better addressed this way.

These general considerations about the economical feasibility of software migration are particularized as follows in the case of nTile PHPtoJava.

PHP is suitable for the development of small-to-medium Web sites. Once the code base reaches a certain size, maintenance becomes difficult due to the characteristics of the language: procedural (up to version 4), weakly and dynamically typed, interpreted. Specialists are readily available, but they tend to be entry-level. Due to the permissive and sometimes inconsistent nature of the language, it is difficult to develop advanced tools for it, therefore these tools are not in widespread use.

Java is a widely used, well-supported and expressive language. Tools for Java and for the Java EE platform are in large supply. The number and quality of specialists is fully satisfactory.

The advantages of migrating from PHP to Java arise from the nature of the target language:

- better error detection and traceability, both at compile-time and at runtime;
- better maintenance, performance and scalability;
- easy access to many modern technologies, based on Java EE but not only;
- good tools, including the ones for refactoring.

Some of the qualitative improvements offered by the automated translation process in nTile PHPtoJava refer to extraction of objects and components (based on dynamic includes) and to detection of ambiguities (erroneous function returns, duplicated formal function parameters, class constructors used as regular functions etc.).

The migration process of nTile PHPtoJava produces detailed information about the structure of the translated application. This information could be used to implement various by-products, such as:

- visualization of control and data flows, source file dependencies;
- quality metrics and detection of problematic constructs;
- intelligent PHP source code editors (e.g. auto-completion, navigation to declaration/usage, syntax highlight, type hinting);
- refactoring features.

The detail level of the intermediate representation for the PHP source code can provide the basis for implementing all these analysis algorithms.

Finally, feasibility considerations apply to the development of software migrators themselves. Economical constraints must be balanced with quality needs, more stringently so because a migrator is a product and not a project. It thus needs to have a superior quality and to be designed for high maintainability, extensibility and reuse.

In practice, a software migrator will not offer a 100% coverage for the source language and its libraries. Each development iteration should prioritize the most commonly used constructions, whose implementation effort does not surpass a reasonable limit. The translator won't usually cover the constructions that are rarely used and/or require extensive effort to be translated – these shall be translated manually.

If a certain software application contains specific patterns, these can be translated in a special manner by elaborating heuristic algorithms that are custom-tailored to the respective situation. The disadvantages of automated translation using an off-the-shelf software migrator are thus minimized.

5. Conclusions

The natural evolution of programming languages induces the need to migrate legacy application systems towards modern languages and platforms. Economical factors play a role just as important as technical factors in the migration's decision-making process.

A viable alternative to manual migration is using a specialized tool called a software migrator. Software migrators have a life-cycle that is also determined by economical and technical factors. Their quality is extremely important and needs to be measured and continuously improved. Generality is one of the key metrics of software migrators.

The case-study presented in this paper, nTile PHPtoJava, was designed and developed with all these considerations in mind.

Bibliography

1. Enyedi, R. **Metricile generalitatii translatoarelor software,** "Informatica Economica" Journal, no. 34, 2004, pp. 65-68
2. Enyedi, R. **Tehnici si metode de translatare a aplicatiilor informatice,** Doctoral Thesis, ASE Bucharest, 2006
3. Ivan, I., and Popescu, M. **Metrici Software,** INFOREC Publishing House, Bucharest, 1999
4. Kassem, N. **Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition,** Addison-Wesley, 2000
5. Kinnersley, B. **The Language List,** http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm
6. Laird, L. M. **Software Measurement and Estimation: A Practical Approach,** Wiley-IEEE Computer Society, 2006
7. Reed, J. **What SAP says about ABAP's future,** http://searchsap.techtarget.com/columnItem/0,294698,sid21_gci1282035,00.html
8. Sebesta, R. **Concepts of Programming Languages (8th Edition),** Addison Wesley, 2007
9. Watt, D. A. **Programming Language Processors: Compilers and Interpreters,** Prentice Hall International, 1993
10. * * * **Numiton nTile PHPtoJava,** http://www.numiton.com
11. * * * **PHP Manual,** http://www.php.net/manual/en/
12. * * * **The History of Programming Languages,** O'Reilly, 2007, http://www.oreilly.com/pub/a/oreilly/ news/languageposter_0504.html

[1] Robert Enyedi graduated the Faculty of Cybernetics, Statistics and Economic Informatics from the Bucharest University of Economics. His doctoral thesis was on the subject of software translation. He has several years of professional experience in this field. He is the co-founder and CEO of Numiton Ltd., a start-up company specialized in developing software migrators.

[2] Codification of references:

[ENYE04]	Enyedi, R. **Metricile generalitatii translatoarelor software,** "Informatica Economica" Journal, no. 34, 2004, pp. 65-68
[ENYE06]	Enyedi, R. **Tehnici si metode de translatare a aplicatiilor informatice,** Doctoral Thesis, ASE Bucharest, 2006
[IVAN99]	Ivan, I., and Popescu, M. **Metrici Software**, INFOREC Publishing House, Bucharest, 1999
[Kass00]	Kassem, N. **Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition,** Addison-Wesley, 2000
[KINN07]	Kinnersley, B. **The Language List,** http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm
[LAIR06]	Laird, L. M. **Software Measurement and Estimation: A Practical Approach**, Wiley-IEEE Computer Society, 2006
[NUMI07]	* * * **Numiton nTile PHPtoJava,** http://www.numiton.com
[OREI07]	* * * **The History of Programming Languages,** O'Reilly, 2007, http://www.oreilly.com/pub/a/oreilly/ news/languageposter_0504.html
[PHPM07]	* * * **PHP Manual,** http://www.php.net/manual/en/
[REED07]	Reed, J. **What SAP says about ABAP's future,** http://searchsap.techtarget.com/columnItem/0,294698,sid21_gci1282035,00.html
[SEBE07]	Sebesta, R. **Concepts of Programming Languages (8th Edition),** Addison Wesley, 2007
[Watt93]	Watt, D. A. **Programming Language Processors: Compilers and Interpreters,** Prentice Hall International, 1993