Applied Computer Systems

ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
December 2016, vol. 20, pp. 27–35
doi: 10.1515/acss-2016-0012
https://www.degruyter.com/view/j/acss

Extensible Model for Comparison of Expressiveness of Object-Oriented Programming Languages

Ruslan Batdalov[1], Oksana Ņikiforova[2], Adrian Giurca[3]
[1, 2] *Riga Technical University, Latvia,* [3] *Brandenburg Technical University, Germany*

Abstract **– We consider the problem of comparison of programming languages with respect to their ability to express programmers' ideas. Our assumption is that the way of programmers' thinking is reflected in languages used to describe software systems and programs (modelling languages, type theory, pattern languages). We have developed a list of criteria based on these languages and applied it to comparison of a number of widely used programming languages. The obtained result may be used to select a language for a particular task and choose evolution directions of programming languages.**

Keywords **– Programming languages, programming language comparison, programming language expressiveness, programming language expressive power.**

I. Introduction

A variety of existing object-oriented programming languages require programmers to be able to choose the most appropriate one (or ones) for a particular task. It is especially important in the current situation, when the traditional leaders among programming languages are gradually losing their positions [1] and the existing desire to have the "next great language" is acknowledged even by the designers of Java (although they do not share this desire) [2]. Therefore, comparison of programming languages is a topical issue for both programmers and language designers.

One of the possible ways to compare programming languages is to compare functional opportunities that they give to programmers. A focus group discussion during the European Conference on Pattern Languages of Programs (EuroPLoP) acknowledged that programmers often experience difficulties when a programming language lacks features existing in other languages and when a particular language feature is only a special case of a more general concept [3]. In both cases, the desired behaviour is not supported by a programming language and must be implemented manually. This consideration shows that the information about correspondence between features of different languages and the supported / unsupported features is useful to programmers for switching between languages and to language designers for deciding on the features their languages lack.

In our study, we consider functional opportunities of programming languages from the point of view of language expressiveness, i.e., its ability to express ideas of programmers' thinking. This concept is hard to formalise, but it is often used in reasoning about languages (not only programming languages). For example, Stuart Russell and Peter Norvig indicated the limited expressiveness of programming languages in comparison with natural languages [4]. We believe that this fact always motivates development of programming languages towards higher expressiveness. A systematic analysis of expressiveness of programming languages may clarify the current stage of this process and the possible further directions.

The goal of our study is to develop a solid basis for such analysis and validate it by means of application to a number of popular programming languages. The obtained comparison model may be used both by programmers for choosing a particular language for their tasks and by language designers for deciding on possible directions of language development.

The remainder of the paper is organised as follows: Section II describes the background of the study and similar approaches. Section III explains method of comparison, including the set of comparison criteria. Section IV demonstrates how the developed criteria can be applied, and Section V concludes the paper.

II. Background and Related Research

The notion of expressiveness (also called expressive power or expressivity) of a programming language is the measure of the breadth of ideas that can be expressed using this language [5]. One should distinguish it from computational power related to the complexity of problems that can be solved. Almost all programming languages are Turing-complete and therefore have the same computational power, whereas their expressive power may differ significantly [5].

William Farmer distinguished between the *theoretical expressiveness* related to the set of ideas that can be expressed at all and the *practical expressiveness*, measuring how easy they can be expressed [6]. In our study, we are primarily interested in the practical expressiveness since it can provide a better guidance for choosing a programming language from a practical perspective.

In the practical sense, expressiveness is often associated with the amount of code required to implement some functionality. For example, this approach is used in popular COCOMO II model [7]. Donnie Berkholz used the same meaning of the term in an empirical study covering and ranking over 50 programming languages [8]. In our opinion, this approach allows obtaining useful information relatively quickly, but it is insufficient for a deeper analysis. It reduces the whole language expressiveness to one numeric indicator only and omits all information related to the particular language facilities expressing programmers' ideas. Although, in general, quantitative indicators should be preferred to the qualitative ones, one quantitative indicator is definitely not enough to choose between languages consciously. Therefore, we believe that a comparison taking into account particular language features would provide data that are more valuable.

The importance of considering particular language-level features when assessing expressiveness of programming languages was mentioned by Yizhou Zhang et al. [9]. This aspect of expressiveness is important for not only code brevity, but also efficiency because, according to Donald Knuth, programmers tend to use easier constructs instead of optimal ones [10]. Therefore, we believe that analysis of the range of facilities available in programming languages is essential in the study of their expressiveness.

Žilvinas Vaira and Albertas Čaplinskas discussed the problem of programming language suitability to implement programmers' design decisions [11]. This discussion is close to the concept of expressiveness as we understand it here and has the advantage of using the formal concept of a 'design decision' instead of a rather vague 'idea'. Nevertheless, this formality may be too restrictive, for example, as we say below, one of our sources is the type theory, which is only indirectly related to design decisions. Therefore, we prefer to tolerate some vagueness and continue using the term 'idea'.

It seems that a model for comprehensive comparison of programming languages in terms of their features does not exist yet. Published studies tend to contain a deep analysis of one specific question or task, without covering the full breadth of language facilities [12]–[17]. Therefore, our goal is to fill this gap with a model that would be flexible enough to be applied to various object-oriented programming languages and that would cover a wide range of language facilities related to their expressiveness.

III. Method of Comparison

This section describes our method of comparison, explains the limits of the study and presents the set of comparison criteria identified in the study. The comparison criteria are organised hierarchically according to similarities between them and summarised in Tables I–XII. Each row in these tables contains a number of language-level features, which are separate comparison criteria. Some of them are accompanied by clarifications in parentheses. We assume that for each such criterion, it is possible to say whether a particular language of language version supports it or not (although in some cases, it may be supported only in specific situations, for example, only for specific types of objects) and which language facilities (keywords, concepts, etc.) correspond to it. We strive to make our model as complete as possible, but absolute completeness is certainly an impossible ideal. Therefore, we intentionally design our model to be extensible in the sense that the hierarchical structure described below is suitable for inclusion of other language-level features, both existing and invented in future. The feature hierarchy is designed in such a way that we expect a higher level of the hierarchy to be the most stable one. The list of low-level language features will be definitely augmented in future, but we estimate the probability of appearing of a new element at the top level of the hierarchy as negligible (at least, within the object-oriented paradigm of programming).

A. Basic Principles

Since the expressive power of programming languages is related to the ideas representable in these languages, it may be compared by choosing a set of common ideas in terms of which developers see their programs and studying how easy they can be expressed. In our opinion, such ideas manifest themselves in languages used to describe programs, projects and solutions (for example, modelling languages) since such descriptions are to form a bridge between programmers' mind and actual programs. We assume that these languages and descriptions represent, to some extent, the way of programmers' thinking.

The choice of particular description languages to analyse, on the one hand, should allow us to create a model that would be universal enough, but on the other hand, should be limited to fit into the scope of the study. We consider the following classes of description languages to be the most important for our task:

- Modelling languages used to create models of software systems (typically, but not necessarily, before actual programming);
- Mathematical theories providing a formal basis for concepts used by programmers and relationships between them;
- Pattern languages, which give ready to use solutions to many practical problems. It is known that one of the most important functions of design patterns is to create a language for communicating solution description between programmers [18].

Many constructs used in these languages have direct counterparts in commonly used programming languages, and this relationship facilitates their usage for comparison. At the same time, some of them do not have such counterparts or are supported only partially. Our previous studies showed that dissecting these languages allows finding potential language-level features that are not supported in the most popular existing programming languages at all or have only limited support [19], [20]. These considerations suggest that the constructs of the mentioned languages may form (after analysis and systematising) an appropriate basis for comparison. Thus, our approach may be summarised as follows: dissect the mentioned languages into elementary constructs (at the level with constructs of programming languages), systematise them into a structured set of comparison criteria and perform comparison of languages according to these criteria.

It should be noted that description languages by themselves are unable to provide a complete set of comparison criteria since they are much less detailed than programming languages. Instead, we built a skeleton based on description languages and detailed it further in the course of actual comparison between programming languages. The data flow of this process is shown in Fig. 1.

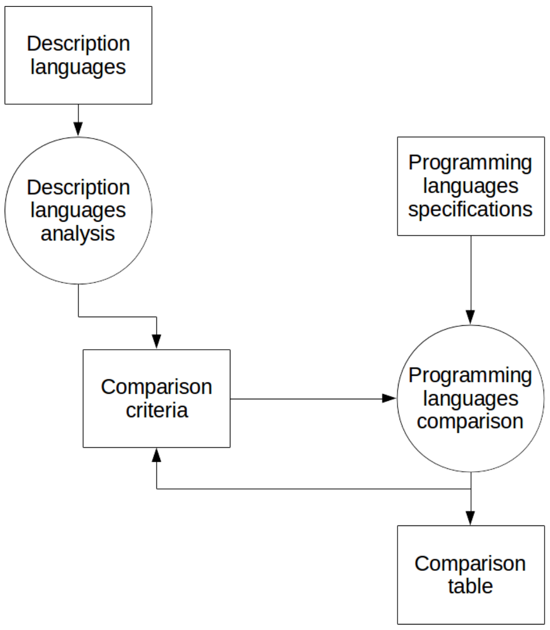
Description languages
Description languages analysis
Programming languages specifications
Comparison criteria
Programming languages comparison
Comparison table

Fig. 1. Data flow of the process of building and applying comparison model.

B. *Scope of the Study*

This section describes the decisions taken with respect to the boundaries of the study and their motivation.

We consider the following description languages, which are, in our opinion, representative of the classes mentioned above:

- Unified Modeling Language (UML) – the industrial standard of software modelling [21];
- Type theory – the basis of calculi commonly used for formal description of programming languages [22];
- Patterns described in three classical sources: "Design Patterns" by Erich Gamma et al. [23], "Pattern-Oriented Software Architecture, A System of Patterns" by Frank Buschmann et al. [24], "Patterns of Enterprise Application Architecture" by Martin Fowler [25].

Similarities in the internal structure of these sources immediately suggest the top-level classes of comparison criteria that we can identify from them. The UML divides its diagrams into structure and behaviour diagrams [21]. The type theory pays less attention to this distinction, but studies both structural and behavioural aspects of data types [22]. At the same time, one of the classification dimensions used by Erich Gamma et al. in their seminal work on design patterns adds the third element to this classification: they divided patterns into creational, structural and behavioural [23]. Probably, a separate concept of object creation management and its difference from other types of behaviour is not as important for UML purposes as it is for the tasks solved by design patterns. In our opinion, for the goal of our study, the class of creational features is significant as well, but it should be generalised to the whole lifecycle of objects, not only their creation. Languages often vary in their approach to the object lifecycle management and provide a programmer with different opportunities. Thus, a three-part classification is used at the top level of our criteria hierarchy: structural, lifecycle and behavioural features.

According to the primary goal of having a set of comparison criteria related to the ability of programming languages to express ideas of programmers' thinking, some types of features are deliberately not included in our study:

- Features of syntactical nature only that do not reflect separate ideas (although the decision on when to apply this requirement is often subjective);
- Features dealing primarily with implementation issues, such as memory allocation rules in C++ (in a sense, these features also express some ideas, but they are hard to compare between languages because of different assumptions about the run-time environment and because languages other than C++ are generally very poor in such features);
- Features that are incomparable between languages due to their tight coupling to a concrete language and its structure, such as reflection mechanisms;
- Features of popular frameworks, not languages themselves (primarily, this restriction is imposed only to make the goal feasible because a similar comparison of frameworks would also be useful).

The languages to compare are chosen according to the following criteria:

- 9 out of the first 10 programming languages in the TIOBE index of the popularity of programming languages as of June 2016: Java, C++, Python, C#, PHP, JavaScript, Perl, Visual Basic .NET and Ruby [1]. C, occupying the second place in this ranking, is omitted from our study due to the lack of object-oriented capabilities.
- Three more recent languages, which have many features absent from more popular ones and show the current trends in programming language development (although these languages have not gained great popularity yet): Scala, Go (Golang) and Kotlin.

We believe that the boundaries of the study defined in this way allow us to obtain a useful and sufficiently universal set of comparison criteria.

C. *Structural Features*

Structural features are related to the static structure of programs. In our opinion, two main respects in which languages differ are the types of objects that exist in the language and the way they are related to each other. Therefore, our list of comparison criteria is organised according to these questions.

Possible types are well studied in the established type theory. The type theory distinguishes between base (or uninterpreted) types and compound types built from other ones [22]. Languages differ in supported base types (integers, Booleans, etc.), methods of building compound types (arrays, records, etc.), and types of literals supported for each type (for example, decimal, hexadecimal, octal or binary integer literals). The identified language features are given in Tables I and II.

TABLE I
BASE TYPES

Group	Features
1. Signed integer	Supported type(s); decimal, hexadecimal, octal, binary literals
2. Unsigned integer	Supported type(s); decimal, hexadecimal, octal, binary literals
3. Floating-point	Supported type(s); decimal, hexadecimal literals
4. Complex	Supported type(s); literals
5. Boolean	Supported type; literals
6. Character	Supported type(s); literals; character escaping
7. String	Supported type(s); string literals; regular expression literals; character escaping; variable interpolation
8. Date	Supported type(s); literals
9. Raw bit string	Supported type(s); literals
10. Function as an object	Supported type(s)

TABLE II
COMPOUND TYPES

Group	Features
1. Reference	Supported type(s)
2. Number-indexed compounds (array)	Supported type(s); literals
3. Name-indexed compounds	
3.1. With fixed set of keys (record)	Supported type(s); literals
3.2. With variable set of keys (associative array)	Supported type(s); literals
3.3. Set	Supported type(s); literals
3.4. Class	Supported type(s); underlying implementation
3.5. Interface	Supported type(s)
4. Variants	
4.1. General variants	Supported type(s)
4.2. Option (nullable type)	Supported type(s); undefined value
4.3. Enumeration	Supported type(s); underlying value may be set

It should be noted that the distinction between base and compound types is not the same as the distinction, existing in Java [2] and some other languages, between primitive and reference types. The latter distinction does not form a good basis for comparison because it does not exist in all languages (for example, Python considers every value or variable to be an object [26]). For this reason, as well as in order to make our comparison criteria closer to the established type theory, we differentiate between base and compound types depending on whether the *data* belonging to an object are treated as an atomic value or as a group of values of other types. This interpretation is still not universal (for example, strings may be treated as either atomic values or sequences of characters, i.e., cannot be uniformly classified as a base or as a compound type), but allows forming a set of comparison criteria.

For the sake of universality, we consider classes and interfaces to be separate compound types. This distinction generally does not exist in the languages themselves – for example, in the type theory interpretation, C++ and Java classes are just a specific kind of records [22]. Nevertheless, in some other languages (for example, JavaScript and Python), classes are implemented by means of associative arrays instead of records [26], [27], and Perl allows using arbitrary data structures [28]. To make our model suitable for all these cases, we consider classes and their possible implementations separately and include underlying class implementation as a comparison criterion as well.

Relationships between classes (the most important kind of types in object-oriented languages [22]) and objects are reflected in UML structural diagrams [21], but different semantics of relationships, which is not always reflected in a diagram, should also be considered. The list of types of relationships used in our model is created by systematisation of relationships described in the UML standard [21], type theory [22], our previous studies [19], [20], and directly in language specifications [2], [26]–[36]. These relationships are given in Tables III, IV and V.

In addition to the opportunity to define a relationship of a particular kind, our set of comparison criteria contains features related to the opportunity to prohibit or restrict creation of such relationships. For example, a language may allow creating a class inheritance from which is prohibited or imposing a restriction on multiplicity of a relationship between objects.

TABLE III
CLASS-TO-CLASS RELATIONSHIPS

Group	Features
1. Specialisation (inheritance)	
1.1. Interface extension	Relationship may be defined; classes may be incomplete without inheritance (abstract); inheritance may be prohibited; inheritance may be restricted to a set of classes known in advance (algebraic types)
1.2. Implementation inheritance	Relationship may be defined; methods may be declared without implementation (abstract); method overriding may be prohibited; how the problem of diamond inheritance of data and methods is solved
1.3. Subclassing a member	Relationship may be defined
1.4. Arbitrary predicate	Relationship may be defined
2. Generalisation	Relationship may be defined
3. Interface realisation	Relationship may be defined; how the problem of diamond inheritance of data and methods is solved
4. Interface extraction	Relationship may be defined
5. Parameterised types	
5.1. Type parameters	Relationship may be defined; parameter may be bounded; parameter may be variant; default parameters
5.2. Usage-site variance of type parameters	Relationship may be defined; parameter may be bounded
5.3. Value parameters	Relationship may be defined; default parameters

TABLE IV

OBJECT-TO-OBJECT RELATIONSHIPS

Group	Features
1. Binary	
1.1. * -> 0..1	Relationship may be defined; source multiplicity may be restricted; destination may be declared obligatory; data may be assigned to a link; aggregation semantics may be added; inheritance semantics may be added (prototypal inheritance)
1.2. * -> *	Relationship may be defined; source multiplicity may be restricted; destination multiplicity may be restricted; data may be assigned to a link; aggregation semantics may be added; inheritance semantics may be added (prototypal inheritance)
2. *n*-ary	Relationship may be defined

TABLE V

CLASS-TO-OBJECT RELATIONSHIPS

Group	Features
1. Class members (static members)	
1.1. 1 -> 0..1	Relationship may be defined; member may be declared obligatory
1.2. 1 -> *	Relationship may be defined; multiplicity may be restricted
2. From class to objects of this class	
2.1. 1 -> 1 (singletons)	Relationship may be defined
2.2. 1-> * (classes as collections)	Relationship may be defined; multiplicity may be restricted

The last group of structural features is related to namespace manipulation, i.e., directives affecting visibility of classes and objects. These features are given in Table VI.

TABLE VI

NAMESPACE MANIPULATION

Group	Features
1. Import	
1.1. Individual type	May be imported; alias may be defined
1.2. Individual object	May be imported; alias may be defined
1.3. Whole namespace/package	May be imported; supplier may define what can be imported; importer may choose what to import
2. Package merge	May be performed
3. Access control	
3.1. Namespace/package	Access may be restricted
3.2. Top-level types and objects	Access may be restricted; access may be provided individually
3.3. Inner types and objects (defined within another type)	Access may be restricted; access may be provided individually

D. Lifecycle Features

This group of criteria contains language features related to a specific kind of program behaviour – management of object lifecycle and bindings (for example, binding of a name to an object, an object to a value, etc.). Programming languages tend to undertake these functions in order not to distract a programmer from implementing the main logic of a program; therefore, we consider these features separately from the behavioural ones.

The lifecycle of an object consists of a number of phases (allocation, initialisation, etc.). A compiler or a run-time environment (RTE) may provide the logic of these phases, but does not always do this. For example, C++ does not have a garbage collector performing automatic memory deallocation [29], and this fact requires a special care from a programmer. On the other hand, even if the lifecycle management logic is provided, a language may allow changing it when necessary. Object initialisers (often called constructors) are an example of such an opportunity. These features (standard lifecycle management logic and the opportunity to redefine it) are given in Table VII.

TABLE VII

OBJECT LIFECYCLE

Group	Features
1. Allocation	Provided by a compiler/RTE; may be (re)implemented by a programmer
2. Initialisation	Provided by a compiler/RTE; may be (re)implemented by a programmer
3. Finalisation	Provided by a compiler/RTE; may be (re)implemented by a programmer
4. Deallocation	Provided by a compiler/RTE; may be (re)implemented by a programmer

The language-level features related to various bindings are the opportunity to set or change the binding, the opportunity to check current binding (for example, whether an object belongs to a class) and the opportunity to prohibit further changes (for example, to declare an object to be constant). A language may provide different opportunities in this respect at different stages of program or object lifecycle. Therefore, Table VIII contains the list of bindings considered in our model together with different phases when a binding may be defined or changed.

TABLE VIII

BINDINGS

Group	Features
1. Name-to-type (compile-time type)	
1.1. Compilation phase	Binding may be defined; binding may be inferred
2. Object-to-type (run-time type)	
2.1. Allocation phase	Binding may be defined; further rebinding may be prohibited
2.2. Initialisation phase	Binding may be changed; further rebinding may be prohibited
2.3. Run-time phase	Binding to exact class may be checked; binding to a class or its subclasses may be checked; binding may be changed
3. Name-to-object (in a practical sense, name-memory bucket)	Binding may be defined; further rebinding may be prohibited

4. Object-to-value (object state)	
4.1. Initialisation phase	Binding may be defined; binding may be deferred (lazy evaluation); further rebinding may be prohibited for an object; further rebinding may be prohibited for the whole class (immutable classes)
4.2. Run-time phase	Further rebinding may be prohibited
5. Name-to-value	Binding may be defined; further rebinding may be prohibited
6. Abstract class-to-default implementation	
6.1. Compile-time	Binding may be defined; further rebinding may be prohibited
6.2. Run-time	Binding may be changed; further rebinding may be prohibited

Binding of a name to a value is not a separate type of binding, but rather the result of joint action of the two preceding ones (name-to-object and object-to-value). At the same time, it is convenient to reason in terms of this binding; therefore, we include it into the list of comparison criteria. Binding of an abstract class to its default implementation is not widely supported at the language level [20]; it is rather associated with dependency injection in frameworks (for example, Spring Framework [37]). Nevertheless, we include it too because some languages have limited support of it.

E. Behavioural Features

The group of behavioural features contains features that are used to implement program logic. Their primary source is direct comparison and systematisation of programming language specifications [2], [26]–[36] because theoretical studies tend to use just a few basic operations, not reducible to each other. For example, lambda-calculus, used in the type theory, contains only one operation of beta-reduction and a few reduction strategies (for example, call by name and call by value) [22]. Real languages, on the other hand, contain many different operations, even if some of them may be implemented using other ones. At the same time, unlike theoretical studies, design pattern descriptions are an appropriate source to identify potential language-level behavioural features as we showed in [20]. Behavioural features used in our comparison model are given in Tables IX, X, XI and XII.

TABLE IX
CALLS BETWEEN OBJECTS

Group	Features
1. Object call	Synchronous request-response; asynchronous request-response; pipe&filter; broadcast; blackboard; publish-subscribe; callable objects
2. Call arguments	Self-reference (the object for which the method is called); call by value; changeable call by reference; unchangeable call by reference; call by name (lazy evaluation of expressions); default arguments; named arguments; open argument list
3. Returning result	All at once; piece by piece
4. Exception transfer	Checked exceptions; unchecked exception; exceptions carrying values
5. Function overloading	Functions can be overloaded; operators can be overloaded; call dispatch by compile-time class; call dispatch by run-time class
6. Method overriding	Methods can be overridden; call dispatch by compile-time class; call dispatch by run-time class; duck typing; covariant return types

TABLE X
CONTROL STRUCTURES

Group	Features
1. Assignment	Data copy; aliasing; destructuring assignment
2. Conditional branching	If-then-else; switch
3. Iteration	While-loop starting with condition test; while-loop starting with loop body execution; for-alias for while-loop; for-each loop
4. Control transfer	Arbitrary control transfer (goto); exit from a loop; next iteration; immediate return; exception raising; condition assertion
5. Exception handling	Exception handler; blocks executed independently on whether an exception occurred (finally); automatic resource closing
6. Thread synchronisation	Explicit locks; implicit locks; transactional logic

TABLE XI
OPERATORS IN EXPRESSIONS

Group	Features
1. Arithmetic	Increment; decrement; addition; subtraction; multiplication; integer division; division of integers with floating-point result; floating-point division; division of floats with rounding to integer; remainder for integer operands; remainder for floating-point operands; exponentiation; matrix multiplication; shortcut assignment
2. Bitwise	Complement; left shift; signed right shift; unsigned right shirt; and; or; xor; shortcut assignment
3. Logical	Not; and; or; xor; shortcut assignment
4. String	Concatenation; repetition; shortcut assignment
5. Comparison	Comparison for equality; comparison for identity; numerical relational operators; string relational operators; regexp matching; collection membership check
6. Compound-type accessors	Number-indexed; name-indexed; name-indexed removal; complex queries
7. Type control	Type cast; referencing; dereferencing
8. Pseudo-control structures	Conditional expressions; coalescing; for-comprehension
9. Object-oriented operators	Object instantiation; object destroy; reference to superclass; instance of
10. Anonymous types	Anonymous functions; anonymous classes; anonymous variant types; anonymous enumerations

TABLE XII
OTHER OPERATIONS

Group	Features
1. Delegation	Get/set; generator; arbitrary delegation; self-reference preservation
2. Undo/redo/logging	Undo/redo/logging implemented in object itself; undo/redo/logging implemented in executor

IV. Application Cases of Comparison Criteria

In this section we demonstrate three use cases of application of the developed model and explain how the results of the comparison should be interpreted (especially, the negative ones).

A. Comparison of Supported Features

Table XIII shows differences in support of structural relationships (features from Tables III–V) between Java and Scala. The features supported in both languages or supported in neither language are not included. This use case may be applied when we are interested in features of one language that are not supported in another one.

TABLE XIII
Java vs Scala: Structural Relationships

Feature	Java [2]	Scala [34]
III. Class-to-class relationships		
1. Specialisation (inheritance)		
1.1. Interface extension		
Inheritance may be restricted to a set of classes known in advance (algebraic types)	No	If subclasses are defined in the same source file (**sealed** classes)
5. Parameterised types		
5.1. Type parameters		
Relationship may be defined	>=5.0 (generics)	Yes (parameterised types)
Parameter may be bounded	>=5.0 (**extends**)	Yes (<:, >:)
Parameter may be variant	Only for standard arrays (they are covariant)	Yes (+/− annotations)
Default parameters	No	Yes (**implicit**)
5.2. Usage-site variance of type parameters		
Relationship may be defined	>=5.0 (? type parameters)	>=2.6 (**forSome**)
Parameter may be bounded	>=5.0 (**extends**, **super**)	>=2.6 (<:, >:)
V. Class-to-object relationships		
1. Class members (static members)		
1.1. 1 -> 0..1		
Relationship may be defined	Yes (**static** members)	No
1.2. 1 -> *		
Relationship may be defined	Yes (**static** member collections)	No
2. From class to objects of this class		
2.1. 1 -> 1 (singletons)		
Relationship may be defined	No	Yes (**object**)

B. Looking for Equivalent Keywords

Table XIV matches equivalent phases of object lifecycle management (features from Table VII) in C++ and in Python. Unlike Table XIII, features supported in both languages are included. This use case may be applied when we want to find constructs of one language equivalent to the constructs of another one (for example, which method plays the same role in Python as a constructor in C++).

TABLE XIV
C++ vs Python: Object Lifecycle

Feature	C++ [29]	Python [26]
1. Allocation		
Provided by a compiler/RTE	Yes (memory required for class data)	Yes (empty dictionary)
May be (re)implemented by a programmer	Yes (**new** operator)	Yes (__new__ method, metaclass definition)
2. Initialisation		
Provided by a compiler/RTE	No-op (memory is not initialised)	Yes (empty dictionary)
May be (re)implemented by a programmer	Yes (initialisation, constructor)	Yes (__init__ method)
3. Finalisation		
Provided by a compiler/RTE	No-op	No-op
May be (re)implemented by a programmer	Yes (destructor)	Yes (__del__ method)
4. Deallocation		
Provided by a compiler/RTE	Yes, but for pointer types requires explicit call (**delete**)	Yes (garbage collection)
May be (re)implemented by a programmer	Yes (**delete** operator)	No

C. Studying Spread of Feature Support

Piecewise return of the return value of a function is a relatively new feature in modern programming languages (although it is just a special case of co-programs, known since the late 1950s [10]). Table XV shows which languages and their versions support this feature and which keywords they use to implement it. This use case may be applied when we are interested in how widely a particular language feature is supported.

TABLE XV
Support of Piece-by-Piece Return from a Function

Language	Supported
Java	No
C++	No
Python	>=2.2 (**yield**)
C#	>=2.0 (**yield return**)
PHP	>=5.5 (**yield**)
JavaScript	>=2015 (**yield**)

Perl	No
VB .Net	>=11.0 (**Yield**)
Ruby	No
Scala	May be emulated (lazily evaluated data structures, e.g., Stream)
Go	May be emulated (channels)
Kotlin	No

As a side effect, comparison of programming languages in this way helps avoid pitfalls related to the different functionality of the same keywords in different languages (similar to ‘false friends’ in natural languages). For example, Ruby has the **yield** keyword too, and it may be appealing to assume that this keyword does the same as in other languages. However, Ruby **yield** is a specific way of calling a subroutine (calling a block of code passed as a method parameter), not of returning result [33]. Therefore, it would be incorrect to equate Ruby **yield** with **yield** in other languages, although all of them are related to the control transfer and can be used to solve the same tasks.

D. Interpretation of Comparison Results

The comparison tables given above consider only those features to be supported that have direct or almost direct counterparts in the languages under analysis. They do not describe how to implement equivalent functionality in the language if such a direct counterpart does not exist.

In order to avoid misunderstanding, it is necessary to explain what the absence of support means from the practical point of view. It does not mean that the corresponding task cannot be solved using the language. All considered languages are Turing-complete [38], which means that we can solve any task in any of these languages or cannot solve it on a computer at all. Therefore, new language features do not extend the set of solvable tasks. Instead, they can give such benefits as, for example, programmers’ convenience and program safety.

In our opinion, the classification used in our study provides a general guidance on the consequences of the absence of a particular feature (even though these rules do not work always):

- Absence of a behavioural feature means that the required behaviour should be implemented using other behavioural features. Obviously, it is always possible, but may require extra code. Often, but not always, this code may be componentised into a separate function and reused (many such examples may be found in popular libraries and frameworks).
- Absence of a structural feature related to a type means that the task should be solved using other data types. It may lessen type safety. For example, if the program logic requires that the value a particular variable lies in a particular range, using an integer variable instead requires manual conditions checks, which can be accidentally omitted.
- Absence of a structural feature related to a relationship means that a programmer must bear this relationship in mind without any support from the compiler. It may require re-implementing some code (otherwise provided by the compiler) and lessen type safety because the relationship is not under the type control.
- Absence of a lifecycle feature is a more difficult case. Since lifecycle management and binding management are usually ensured by a compiler, insufficiency of its facilities may require the programmer to write a lot of low-level code. Inability to change a binding may require re-creating an object completely. At the same time, inability to prohibit changing a binding (for example, inability to ensure that the value of an object will never change) lessens the safety of a program.

Therefore, we believe that even though the absence of a feature still allows a programmer to implement its function using other ones, it may make a program more complex or less safe (or both).

V. Conclusion and Future Research

Our study addressed the problem of comparison of object-oriented programming languages in respect of their expressive power, i.e., their ability to express ideas in terms of which programmers think about their programs. Analysis of a few languages used for the description of programs and software systems (UML, type theory, most popular design patterns) allowed us to form a basis for such comparison. This basis was validated and detailed in actual comparison of a number of object-oriented programming languages. The resulting comparison tables may be used for pairwise or group-wise comparison of programming language expressiveness, determining language construct equivalent to the ones of another language, as well as studying the degree of support of a particular language-level feature in different languages.

Possible future directions of the research are as follows:

- Development of a graphical interface for convenient usage of full comparison tables developed in the study.
- Studying how the developed qualitative indicators are related to the quantitative ones (such as number of lines of code) and to higher-level tasks (for example, which features are important in web programming and which are not).
- Application of the developed comparison model to other programming languages.
- Extending the model with new language-level features, both existing in some languages and not existing currently, but motivated by the theory or practical tasks.
- Studying opportunities to add missing features to programming languages that currently do not have them.

Acknowledgment

We would like to thank two anonymous peer reviewers for their valuable comments, questions and suggestions.

References

[1] Matrix Resources, “June TIOBE index indicates the fall of programming market leaders,” June 2016. [Online]. Available: http://www.matrixres.com/resources/tech-trends/june-tiobe-index-indicates-the-fall-of-programming-market-leaders/ [Accessed: Nov. 28, 2016].

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, “The Java language specification: Java® SE 8 edition,” March 2015. [Online].

Available: http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf [Accessed: Nov. 28, 2016].

[3] R. Batdalov, "Is there a need for a programming language adapted for implementation of design patterns?" in *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany, July 6–10, 2016.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ, USA: Prentice Hall, 1995.

[5] A. Leitão, S. Proença, "On the expressive power of programming languages for generative design: the case of higher-order functions," in *Proceedings of the 32nd International Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe)*, Newcastle upon Tyne, England, Sep. 22–26, 2014, pp. 257–266.

[6] W. M. Farmer, "Chiron: a multi-paradigm logic," *Studies in Logic, Grammar and Rhetoric*, vol. 10, no. 23, 2007, pp. 1–19.

[7] S. C. McConnell, *Code Complete*, Microsoft Press, 2004.

[8] D. Berkholz, "Programming languages ranked by expressiveness," March 2013. [Online]. Available: http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/ [Accessed: Nov. 28, 2016].

[9] Y. Zhang, M. C. Loring, G. Salvaneschi, B. Liskov and A. C. Myers, "Lightweight, flexible object-oriented generics" in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, June 13–17, 2015, pp. 436–445. https://doi.org/10.1145/2737924.2738008

[10] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1997.

[11] Ž. Vaira and A. Čaplinskas, "Software engineering paradigm independent design problems, GoF 23 design patterns, and aspect design," *Informatica*, vol. 22, no. 2, pp. 289–317, Apr. 2011.

[12] Z. Anik and O. F. Baykoç, "Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: a pilot study," *Computer Applications in Engineering Education*, vol. 19, no. 1, pp. 89–96, March 2011. https://doi.org/10.1002/cae.20294

[13] N. Archvadze and M. Pkhovelishvili, "Reforming the trees – C# and F# comparison," in *Proceedings of the 4th International Conference on Problems of Cybernetics and Informatics (PCI)*, Baku, Azerbaijan, Sep. 12–14, 2012, pp. 1–4. https://doi.org/10.1109/ICPCI.2012.6486287

[14] B. M. Brosgol, "A comparison of generic template support: Ada, C++, C#, and Java ™," in *Proceedings of the 15th Ada-Europe International Conference on Reliable Software Technologies* (Lecture Notes in Computer Science), Valencia, Spain, June 14–18, 2010, pp. 222–237. https://doi.org/10.1007/978-3-642-13550-7_16

[15] R. Lämmel, M. Leinberger, T. Schmorleiz and A. Varanovich, "Comparison of feature implementations across languages, technologies, and styles," in *Proceedings of Software Evolution Week / IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Antwerp, Belgium, Feb. 3–6, 2014, pp. 333–337. https://doi.org/10.1109/csmr-wcre.2014.6747188

[16] M. Stein and A. Geyer-Schulz, "A comparison of five programming languages in a graph clustering scenario," *Journal of Universal Computer Science*, vol. 19, no. 3, pp. 428–456, 2013.

[17] N. Togashi and V. Klyuev, "Concurrency in Go and Java: performance analysis" in *Proceedings of the 4th IEEE International Conference on Information Science and Technology (ICIST)*, Shenzen, China, Apr. 26–28, 2014, pp. 213–216. https://doi.org/10.1109/icist.2014.6920368

[18] F. Buschmann, K. Henney and D. C. Schmidt, *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, Wiley, 2007.

[19] R. Batdalov, "Inheritance and class structure," in *Proceedings of the 1st International Scientific-Practical Conference Object Systems – 2010*, Rostov-on-Don, Russia, May 10–12, 2010, pp. 92–95.

[20] R. Batdalov and O. Nikiforova, "Towards easier implementation of design patterns," in *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, Rome, Italy, August 21–25, 2016, pp. 123–128.

[21] OMG, "OMG Unified Modeling Language ™ (OMG UML)," March 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/PDF [Accessed: Nov. 28, 2016].

[22] B. C. Pierce, *Types and Programming Languages*, MIT Press, 2002.

[23] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, 1995.

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 2013.

[25] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[26] Python Software Foundation, "The Python Language Reference," Oct. 2016. [Online]. Available: https://docs.python.org/3/reference/index.html [Accessed: Nov. 28, 2016].

[27] Mozilla Developer Network, "JavaScript Reference," November 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference [Accessed: Nov. 28, 2016].

[28] Perl 5 Porters, "Language Reference." [Online]. Available: http://perldoc.perl.org/index-language.html [Accessed: Nov. 28, 2016].

[29] ISO/IEC, "Information technology – Programming languages – C++," ISO/IEC standard 14882:2014(E), Dec. 15, 2014.

[30] Microsoft Corporation, "C# Language Specification: Version 5.0," 2012. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=7029 [Accessed: Nov. 28, 2016].

[31] PHP Group, "Language Reference," 2016. [Online]. Available: http://www.php.net/manual/en/langref.php [Accessed: Nov. 28, 2016].

[32] Microsoft Corporation, "The Microsoft® Visual Basic® Language Specification: Version 11.0," 2016. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=15039 [Accessed: Nov. 28, 2016].

[33] ISO/IEC, "Information technology – Programming languages – Ruby," ISO/IEC standard 30170:2012(E), April 15, 2012.

[34] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir et al., "Scala Language Specification: Version 2.11." [Online]. Available: http://www.scala-lang.org/files/archive/spec/2.11/ [Accessed: Nov. 28, 2016].

[35] Google, "The Go Programming Language Specification," May 31, 2016. [Online]. Available: https://www.golang.org/ref/spec [Accessed: Nov. 28, 2016].

[36] JetBrains, "Kotlin Language Documentation." [Online]. Available: http://www.kotlinlang.org/docs/kotlin-docs.pdf [Accessed: Nov. 28, 2016].

[37] R. Johnson, J. Hoeller, K. Donald, C. Shampaleanu, R. Harrop et al., "Spring Framework Reference Documentation," 2016. [Online]. Available: http://docs.spring.io/spring/docs/5.0.0.M3/spring-framework-reference/htmlsingle/ [Accessed: Nov. 28, 2016].

[38] TIOBE, "TIOBE Programming Community Index Definition." [Online]. Available: http://www.tiobe.com/tiobe-index/programming-languages-definition/ [Accessed: Nov. 28, 2016].

Ruslan Batdalov received the Specialist degree in Applied Mathematics and Informatics from Kazan State University, Russia, in 2003.

He is a second-year Master student and Research Assistant at the Department of Applied Computer Science, Riga Technical University. Previously, he worked as a Business Analyst, Systems Analyst, Activity-Based Costing Specialist. His current research interests include programming languages, their capabilities, structure and design.

He has been the ACM member since March 2010.

E-mail: Ruslan.Batdalov@edu.rtu.lv

Oksana Ņikiforova received the Doctoral degree in Information Technologies (system analysis, modelling and design) from Riga Technical University, Latvia, in 2001.

She is a Professor at the Department of Applied Computer Science, Riga Technical University. Her current research interests include object-oriented system analysis, design and modelling, especially the issues in model driven software development.

E-mail: oksana.nikiforova@rtu.lv

Adrian Giurca received the Doctoral degree in Computer Science (Artificial Intelligence) from the University of Bucharest, Romania in 2004.

He is a Research Associate of Brandenburgische Technische Universität Cottbus-Senftenberg. His current research interests include methods and applications for information systems of the next generation, especially reasoning in social media/software and reasoning in the Web and Semantic Web.

E-mail: giurca@b-tu.de