NUSA (Neat Uniform Simple Architecture): A Highly Orthogonal Programming Language

Bernaridho I. Hutabarat, Mochamad Hariadi, Ketut E. Purnama, and Mauridhi H. Purnomo

Abstract—**This paper describes NUSA, a newly developed programming-language which is based on orthogonality and modular programming. Compared to OOPLs (Object-Oriented Programming Languages) that lack orthogonality, NUSA is designed with orthogonality in mind. Issues found during the creation of highly orthogonal language were the inorthogonality in OOPLs, the real semantic of class, encapsulation of code and data through module, and the unbundling of operators from record-type. The result is NUSA, a highly orthogonal programming-language. NUSA provides input-output, user-defined types, and user-defined (infix and prefix) polymorphic operators for user-defined types. Quantitative comparison with C# in terms of number of source-code lines is presented from the perspective of pragmatic advantage.**

Index Terms — **Orthogonality, Module, Type-based Encapsulation, Module-based Encapsulation**

I. Introduction

ORTOGONALITY is found in many theories surrounding electronic engineering and programming language. OFDM (Orthogonal Frequency Division Multiplexing) is one example in telecommunication [29].

Orthogonal means independent. An orthogonal axis is independent toward any other axis [1]. Orthogonality (independence) is desired because we can operate or process an axis independently [1].

In programming languages, orthogonality takes form in the uniformity of rules (syntax and semantics; see [2], [14], [25], [31]). By contrast, C++ is inorthogonal. For example, comparison operator == cannot be used to compare values of any type ([28]). Java as defined in [15] is inorthogonal, that causes several confusions as detailed in [16]. Inorthogonality in programming languages ends up with requiring users to remember many exceptions in the syntax and/or semantics ([2], [14], [25], [31]).

The world of programming languages has experienced progress for several decades. The decade of the 1980s was dominated by procedural programming, while the decade of 1990s was dominated by OOPLs. OOPLs continued to be dominant in the decade of 2000s, but approaching the decade of 2010, a number of lecturers and students within the movement of The Third Manifesto (TTM for short) pointed out the OOPLs' lack of orthogonality ([5], [6], [7], [8], [9]), and they created code-translators for highly orthogonal programming language named Tutorial D in response to the lack of orthogonality in OOPLs.

The problems that remain unsolved by TTM are the complexity of the underlying theory (using a concept called *possrep*) and the complexity of programming language (Tutorial D). Tutorial D is very different from C, hindering its acceptance by industry. TTM has another important problem: their proposed theory is not related to the theory of modular programming. Consequently there is nothing in the Tutorial D that helps implement modular programming.

OOPLs' shortcomings (inorthogonalities) are investigated in this paper. A programming-language that avoids the inorthogonalities yet can incorporate polymorphism, inheritance, encapsulation, and modular-programming is designed. We propose NUSA programming language to overcome the inorthogonality.

The rest of this paper is organized as follows: Section 2 explains Type-based Encapsulation, the approach used by all OOPLs. This section elaborates seven inorthogolities that are caused by Type-based Encapsulation. Section 3 explains Module-based Encapsulation, the (opposite) approach used by NUSA. This section details NUSA solution to the inorthogonality problems and thus shows the orthogonality of NUSA. Section 4 shows the pragmatic advantage (quantitative comparison) of NUSA over one representative OOPL: C#. Section 5 extracts some conclusions and describes some future works for the theory of object-orientation and NUSA programming-language.

II. Type-based Encapsulation

A. Problem in the underlying theory: bundling of operators

Type-based encapsulation is the theory underlying the OOPLs. While this term is not yet widely used, we can find its usage related to OOP in [22], [23], [27] and [30]. In type-based encapsulation, record-type is used to encapsulate code (operator) and data. To paraphrase, operators are bundled into record-type.

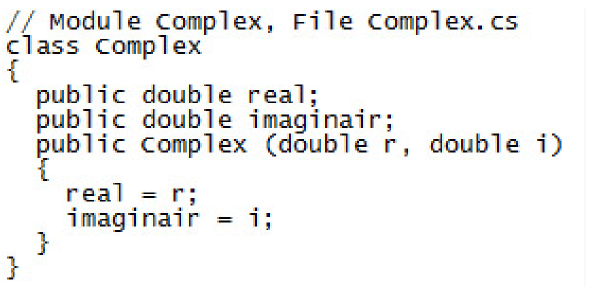
Bundling of the operators is the root of inorthogonalities ([7], [8], [9]). This paper lists seven items of inorthogonality, with five of them universally apply to all OOPLs (the first two items do not apply to C++ and Oracle PL/SQL). For simplicity, this paper focuses on the comparison with C# as defined by [12] and [13].

The record-type is called class in most OOPLs, but not all (Oracle PL/SQL and Delphi are some exceptions). C# and C++ use the term *class* and *struct*. In the remaining sections and subsections, the terms class and record-type are interchangeable, unless otherwise is stated.

Manuscript received June 2, 2011, revised August 15, 2011. The first author is with Bisnis Tekno Ultima, Jakarta 12810, Indonesia. Phone: +62-21-83705972, Fax: +62-21-83706170, e-mailbernaridho@biztek.co.id
The remaining authors' affiliation is Institut Teknologi September, Phone: +62-31-5994251. Fax: +62-31-5931237. Department of Electrical Engineering. Mochamad Hariadi e-mail is mochar@ee.its.ac.id . Ketut E. Purnama e-mail is ketut@ee.its.ac.id . Mauridhi H. Purnomo e-mail is hery@ee.its.ac.id
.

B. *Inorthogonality #1: On definable user-defined types*

Type-based encapsulation limits the user-defined types to record-types only. This is the first inorthogonality. C# and Java disallow the user-defined basic types (often called primitive types) and user-defined collection types. Fig. 1 shows user-defined types in OOPL which are limited to user-defined record-types. The record-type Complex depicted in Fig. 1 serves as a basis for other examples in this paper. The left part of Fig. 2 models the inorthogonality. Only one axis is present, representing that the user-definable types are limited to record-type category only.



```
// Module Complex, File Complex.cs
class Complex
{
  public double real;
  public double imaginair;
  public Complex (double r, double i)
  {
    real = r;
    imaginair = i;
  }
}
```

Fig. 1 Inorthogonal rule #1: User-defined types are limited to record-types

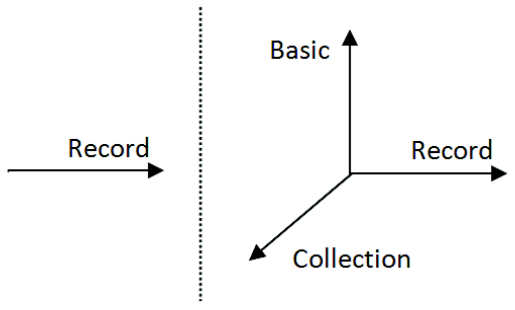


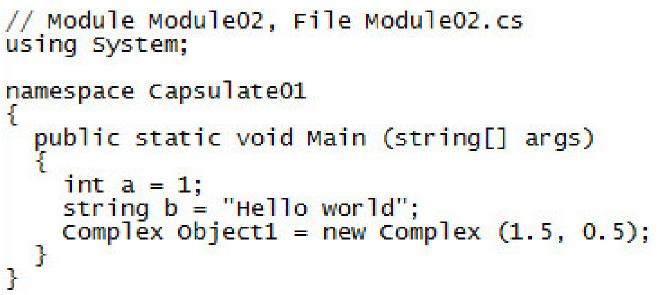
Fig. 2 User-defined types are limited to record types (left part)

Type categories presented here are the result of observing textbooks on programming language theory ([2], [14], [25], [31]). The term type categories is not explicitly listed in those textbooks, but can be found on publications like [18], [19], [21], [24], and [32].

C. *Inorthogonality #2: Memory Allocation*

Type-based encapsulation introduces inorthogonality in memory allocation. Objects of record-types must be allocated dynamically. C# source-code in Fig. 1 depicts the situation.

Source-code in Fig. 3 shows the inorthogonality. Objects of record-types must be allocated dynamically, while objects of other type-categories can be allocated statically. The left part of Fig. 4 models the inorthogonality. Only two axis are present, representing the fact that only objects of two type-categories can be allocated statically.



```
// Module Module02, File Module02.cs
using System;

namespace Capsulate01
{
  public static void Main (string[] args)
  {
    int a = 1;
    string b = "Hello world";
    Complex Object1 = new Complex (1.5, 0.5);
  }
}
```

Fig. 3 Inorthogonal rule #2: record-objects must be allocated dynamically

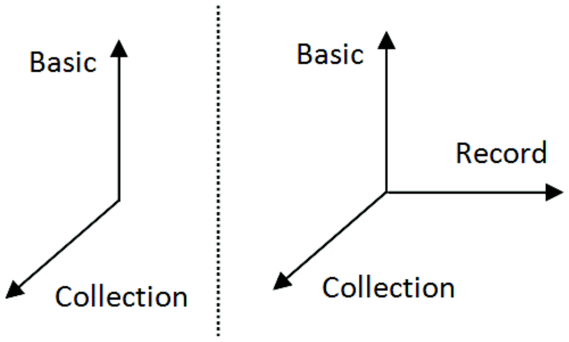
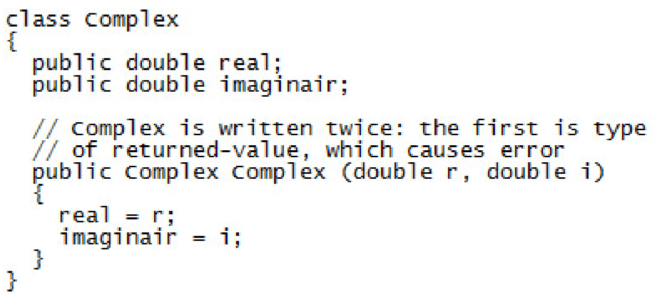


Fig. 4 Inorthogonal rule #2: Objects of only two type-categories can be allocated statically

D. *Inorthogonality #3: Prohibition of specification of type of returned value*

Type-based encapsulation has inorthogonal rules about type of returned-value in operator-header. It prohibits specification of type of returned-value for value-constructor but requires such specification for other operators. The source-code in Fig. 5 is rejected because type of returned-value must be left out. The left part of Fig. 6 shows value-constructors for record-types cannot contain type of returned-value.



```
class Complex
{
  public double real;
  public double imaginair;

  // Complex is written twice: the first is type
  // of returned-value, which causes error
  public Complex Complex (double r, double i)
  {
    real = r;
    imaginair = i;
  }
}
```

Fig. 5 Inorthogonal rule #3: value-constructor cannot specify type of returned-value

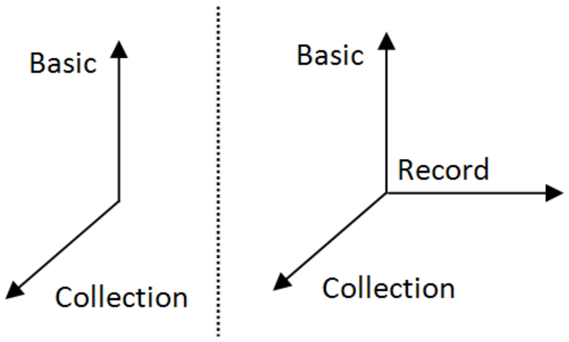
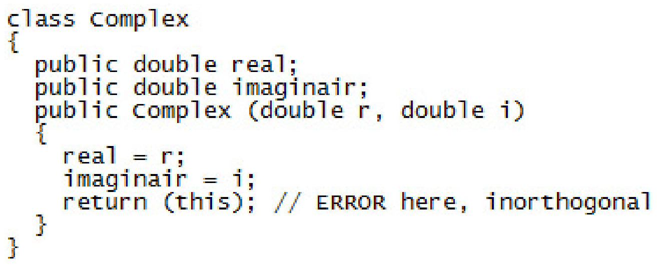


Figure 6 Inorthogonal rule #3: prohibition on specifying the type of returned-value (in left part)

E. *Inorthogonality #4: Prohibition of calling the operator return()*

All OOPLs prohibit the call of operator return (or its equivalents) to return value from constructor. In C#, C++, and Java; call to return() inside value-constructor operator-body raises error. The source-code in Fig. 7 demonstrates it.



```
class Complex
{
  public double real;
  public double imaginair;
  public Complex (double r, double i)
  {
    real = r;
    imaginair = i;
    return (this); // ERROR here, inorthogonal
  }
}
```

Figure 7 Inorthogonal rule #4: value-constructor cannot call operator return()

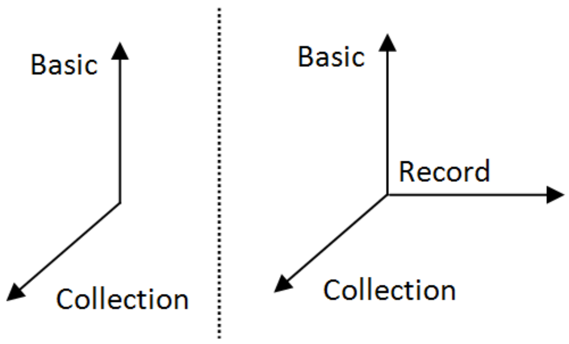
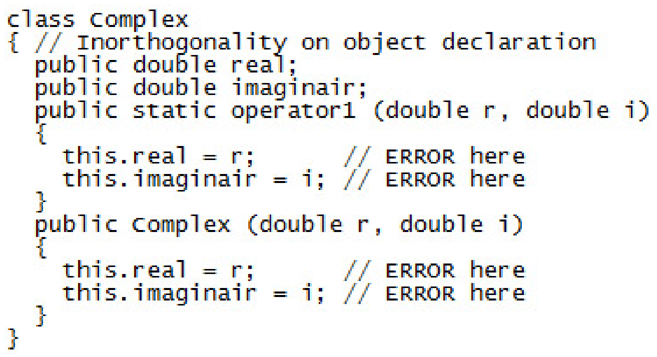


Fig. 8: Inorthogonal rule #4: only function returning basic- and collection-value can call operator return (left part)

The left part of Fig. 8 models the inorthogonality. Type-based encapsulation permits call to operator return() inside operator-body only within the body of operators returning basic-value and collection-value.

F. *Inorthogonality #5:Rules on object-declaration*

The fourth inorthogonality comes from the rules on object declaration. Type-based encapsulation enforces the presence of implicit object (with different enforced names in different language) in some operators, while at the same time prohibits the declaration and usage of the same implicit object in some other operators. The C# source-code in Fig. 9 contains error due to this inorthogonality on object declaration. The left part of Fig 10 models the inorthogonal systems that permit and enforce implicit object of record-type only for dynamic operators.



```
class Complex
{ // Inorthogonality on object declaration
  public double real;
  public double imaginair;
  public static operator1 (double r, double i)
  {
    this.real = r;       // ERROR here
    this.imaginair = i;  // ERROR here
  }
  public Complex (double r, double i)
  {
    this.real = r;       // ERROR here
    this.imaginair = i;  // ERROR here
  }
}
```

Figure 9: Inorthogonal rule #5 is on object-declaration

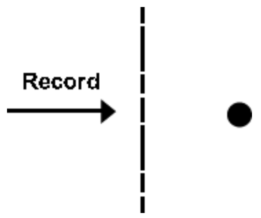
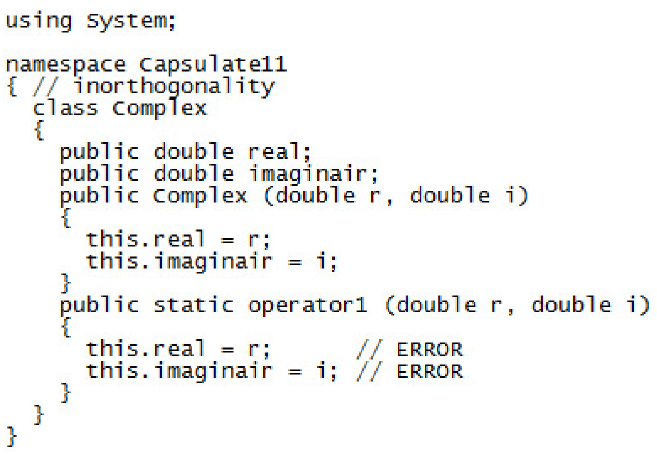


Fig. 10 Inorthogonal rule #5 only objects of record-type can explicitly be present in method

G. *Inorthogonality #6: Rules on operand-passing*

Designers of some OOPLs theorize that the operators should be categorized into dynamic and static ones. While it looks like the difference between the two categories on memory allocation strategies only, more significant differences take place in the matter of operand passing. To dynamic operators ('dynamic methods') in C# and Java implicit operand named *this* is passed (different programming languages use different names: Smalltalk uses Self, Eiffel uses Current). To static operands no implicit operands are passed. This is an inorthogonality. Fig 11 repeats the source-code of Fig 9 emphasizing the difference on operand passing between operator1 and Complex.



```
using System;

namespace Capsulate11
{ // inorthogonality
  class Complex
  {
    public double real;
    public double imaginair;
    public Complex (double r, double i)
    {
      this.real = r;
      this.imaginair = i;
    }
    public static operator1 (double r, double i)
    {
      this.real = r;       // ERROR
      this.imaginair = i;  // ERROR
    }
  }
}
```

Fig. 11 Inorthogonal rule #6: implicit operand is passed to dynamic operators (e.g., operator1)

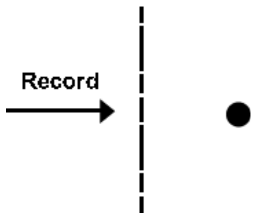


Fig. 12 Inorthogonal rule #6: no implicit operand is passed to static operators (e.g., operator2)

The left part of Fig 12 models the inorthogonality. Implicit operand of record-type is enforced for dynamic operators.

H. *Inorthogonality #7: Record-type name*

In type-based encapsulation record-type name is inorthogonal (dependent) toward module-name. Type-name must equate the module-name. Fig. 13 shows C# source-code where a record-type named Type1 is enforced to be created due to the creation of a module-object named Type1. In other words, the users have no liberty to create record-type with other name(s). Fig 13 shows object named Object1 of type Type1.

This inorthogonality has unpleasant consequence. It is difficult to explain the error within the code like in Fig. 13. Proper explanation must make use the fact that a class in OOPLs like C# is a module (a translation unit (called compilation unit in [12] and [15]), and a record-type Some operators, a type, and an implicit object in classes are inorthogonal (dependent) to the class/module. The error in Fig 11 is in fact due to the inorthogonality #5, #6, and #7.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Capsulate13
{ // inorthogonality
  class Type1
  {
    public /* dynamic */ int column1;
    public static int column2;
    public void operator1 (int an_operand)
    {
      column1 = an_operand;
      column2 = an_operand;
    }
    static void Main (string[] args)
    {
      operator1 (5);|
    }
  }
}
```

Fig. 13 Inorthogonal rule #7: dependent / inorthogonal and implicit record-type

The left part of Fig. 14 models the inorthogonality. Module-name in type-based encapsulation enforces dependent (inorthogonal) record-type name. The right part models an orthogonal system where no type is implied by or dependent upon module name.

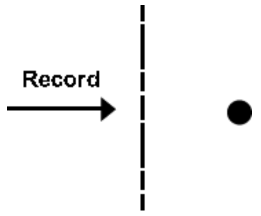


Fig. 14 Modeling the inorthogonal rule #7 . Left: an enforced record-type that is dependent upon module name. Right: no type is dependent upon module name

III. Module-based Encapsulation

This section describes module-based encapsulation. In relationship to the previous section, this section presents the solution to inorthogonality problems. The orthogonality of NUSA is represented by the 0 or 3 axis in the right part of seven figures in the previous section.

A. Module can encapsulate code and data

The term 'Module-based Encapsulation' can be found on [3], [4], [17], [20], [26], and [28]. Module is logical unit of translation, and the means of encapsulation [17]. Both type-based and module-based encapsulation encapsulate code and data. Class in C# and Java are firstly module, and secondly record-type, as is evident in the seventh item of inorthogonality. The following subsections describe how module-based encapsulation in NUSA removes the seven items of orthogonality.

B. Removing the inorthogonality #1

Fig. 15 shows that NUSA is more orthogonal compared to C# because it permits user-defined types of all type-categories. The first user-defined type is basic-type (hundred). The second one is collection-type (Matrix). The third one is record-type (Complex). The right part of Fig. 2 models the orthogonality.

```
Program Prog1 Module Main;// File Main.source

type hundred := subtype (shortword)
  constrain (value < 101);

type Matrix := float[3][3];

type Complex := Record(float real, imaginair;);
```

Fig. 15 Orthogonal user-defined types

C. Removing the inorthogonality #2

NUSA removes the inorthogonality #2 by permitting objects of record-types to be allocated statically, just like objects of basic-types and collection-types. NUSA does not enforce objects of record-types to be allocated dynamically. Object definition's syntax is independent (orthogonal) toward the type of objects (see Fig. 16).

```
Program Prog1 Module Main;

void main()
{
 integer a := 3; // basic
 char[] B := 'Hello'; // collection
 Complex C := Complex (1.0, 0.0); // record
}
```

Fig. 16 Orthogonality #2: memory allocation is orthogonal

D. Removing the inorthogonality #3

Fig. 17 shows the result of NUSA's orthogonal syntax for operator-header and header-declaration. A value-constructor is an operator. Its syntax is exactly the same with any other operator. Assuming record-type Complex has been declared, Fig. 17 shows the declaration of two operators. All operator-headers consist of these mandatory parts: type of returned-value, operator-name, and pair of parentheses. The syntax can accomodate optional parts (like operands, operator-qualifier) without affecting the orthogonality.

```
Complex Complex (float r, i); // constructor
void operator3 ();          // non constructor
```

Fig. 17 Orthogonality #3: specification for type of returned-value is orthogonal

E. Removing the inorthogonality #4

NUSA removes the inorthogonality #4 by requiring ALL functions to explicitly return values. This semantic applies to all operators. Fig. 18 shows an example.

```
Complex Complex (float r, i)
{ // Orthogonal: must call return()
  Complex this;
  this.real := r;
  this.imaginair := i;
  return (this); // explicit return
}
```

Fig. 18 Orthogonality #4: function must call return()

F. Removing the inorthogonality #5

Orthogonal syntax in NUSA helps us understand the errors like shown in Fig. 19. All objects must be declared explicitly. Errors like in Fig. 19 take place simply because no object is declared or passed.

```
Module Complex;
// Shows orthogonality on object declaration

interface

type Complex := Record { float real, imaginair; };
Complex Complex (float r, i);
void operator1 (float r, i);

implementation

Complex Complex (float r, i) // constructor
{
  this.real := r;      // Error here
  this.imaginair := i; // Error here
}

void operator1 (float r, i) // non-constructor
{
  this.real := r;      // Error here
  this.imaginair := i; // Error here
}
```

Fig. 19 Orthogonality #5: objects must be declared explicitly

G. Removing the inorthogonality #6

Changing the previous code (in Fig. 19) into the new code in Fig. 20 removes the inorthogonality #6. The code in Fig. contains no error. In the constructor of Complex, object named this is declared. In the definition of operator1 object named this is passed. Operand-passing within operator1 and operator2 use the same orthogonal syntax. C# requires different codes for doing what is essentially the same thing.

```
Complex Complex (float r, i);
void operator1 (Complex& this; float r, i);

implementation

Complex Complex (float r, i) // constructor
{
  Complex this;
  this.real := r;
  this.imaginair := i;
  return (this);
}

void operator1 (Complex& this; float r, i)
{
  this.real := r;      // No error now
  this.imaginair := i; // No error now
}
```

Fig. 20 Orthogonality #6: objects must be declared or passed explicitly

H. Removing the inorthogonality #7

Fig. 21 shows NUSA ability to emulate type-based encapsulation where record-type name equals module-name.

```
Module Type1; // module-name

interface // explicit record-type named Type1

type Type1 := Record { integer column1; };
integer column2; // column2 isn't record-column
Type1 Type1 (integer an_operand);

implementation

Type1 Type1 (integer an_operand)
{
  Type1 this;
  this.column1 := an_operand; // Not error here
  this.column2 := an_operand; // Error here
  return (this);
}
```

Fig. 21 Orthogonality #7: record-type name is independent toward module-name; record-type name equals module name

Fig. 22 shows that record-type name in NUSA is orthogonal toward module-name. NUSA helps understanding the error in both Fig. 19. Object column2 is not record-column. Consequently, for any record-object the user cannot call the dot operator to access column2.

```
Module Module1; // module-name

interface // explicit record-type named Type1

type Type1 := Record { integer column1; };
integer column2; // column2 isn't record-column
Type1 Type1 (integer an_operand);

implementation

Type1 Type1 (integer an_operand)
{
  Type1 this;
  this.column1 := an_operand; // Not error here
  this.column2 := an_operand; // Error here
  return (this);
}
```

Fig. 22 Orthogonality #7: record-type name is independent toward module-name; different from module name

IV. COMPARING THE NUMBER OF LINES

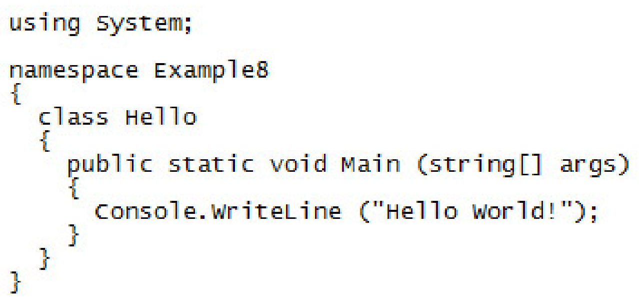
We seek pragmatic advantage from NUSA orthogonality. One idea is to compare the number of source-code lines with an OOPL. Deliberately C# is chosen.

C#	NUSA	Percentage (NUSA / C#)
20	18	90.00
24	23	95.83
43	39	90.70
43	36	83.72
55	56	101.82
117	101	86.32
105	112	106.67
99	94	94.95
15	5	33.33
9	5	55.56
34	22	64.71
36	29	80.56
94	61	64.89
12	5	41.67

Table 1 Comparison of C# and NUSA source-code

The comparison is drawn using the C# source-code from books authored by Deitel and Deitel, using Visual C# 2005 and Visual C# 2008 ([10], [11]); with the final line is adopted from the example in this paper. The result is shown on Table 1. On two examples NUSA source-code is longer than C# (with percentage 101.82 and 106.67), but on the remaining examples NUSA source-code is shorter than C#.

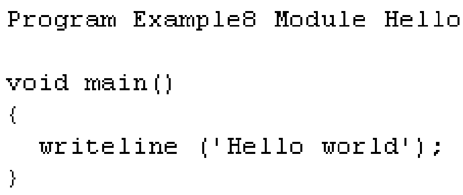
The superiority of NUSA over C# can be seen on the examples representing final line of comparison. Fig. 23 shows 12-line C# source-code to create a typical Hello world program. Fig. 24 shows 5-line NUSA source-code for equivalent program.



```
using System;

namespace Example8
{
  class Hello
  {
    public static void Main (string[] args)
    {
      Console.WriteLine ("Hello World!");
    }
  }
}
```

Fig. 23 C# source-code for Hello world program



```
Program Example8 Module Hello;

void main()
{
  writeline ('Hello world');
}
```

Fig. 24 NUSA source-code for Hello world program

V. CONCLUSION AND FUTURE WORKS

Encapsulation of code and data in type-based encapsulation is manifested in the bundling of operators (code) and columns (data) into record-type. This is the root of the seven items of inorthogonalities in the OOPLs.

Encapsulation of code and data can be manifested in the modules, i.e., module-based encapsulation. This approach removes the seven items of inorthogonalities, leading to the theory of polymorphism and encapsulation that is integrated with theory of modular programming. Module-based encapsulation is the theoretical foundation for orthogonal programming languages that impose less exceptions in the syntax and semantic. The author plans to use module-based encapsulation to establish the formal theory for object-orientation.

The orthogonality in NUSA proves to have pragmatic advantage: shorter source-code to achieve the same result produced by OOPLs. In average, the number of lines of NUSA source-code is 77.91% of C#.

ACKNOWLEDGMENT

The authors thank the Minitry of Information & Communication Technology for partial funding of the development of NUSA code-translator. The first author thanks Prof. Dr. Hery, Dr. Hariadi, and Dr. Ketut for their suggestions.

REFERENCES

[1] Howard Anton, *Elementary Linear Algebra*, 9th ed. Wiley: McGraw-Hill, 1964, pp. 15–64.
[2] Doris Appleby, Julius J. Vandekopple, *Programming Languages: Paradigm and Practice, 2nd ed*: McGraw-Hill, 1997.
[3] Marco Cantu, *Mastering Delphi 6*. Sybex, 2001.
[4] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle; "*Parents are shared parts of objects: Inheritance and encapsulation in SELF, in LISP and Symbolic Computation*", Springer, 2005.
[5] Hugh Darwen, "Valid Time and Transaction Proposals: Language Design Aspects," *in Opher Etzion, Sushil Jajodia, and Suryanaryan Sripada (editors)*, Temporal Database: Research and Practice; 1998.
[6] Hugh Darwen, "An Introduction to Relational Database Theory" *Ventus Publishing*, 2009.
[7] C. J. Date, Hugh Darwen; *The Third Manifesto: Foundation for Object-Relational Databases*; Addison Wesley; 1998.
[8] C. J. Date, Hugh Darwen, "*The Third Manifesto: Foundation for or Future Database Systems*", 2nd ed, Addison Wesley; 1998.
[9] C. J. Date, Hugh Darwen, *The Third Manifesto: Databases, Types, and the Relational Model*. 3rd ed, Addison Wesley; 2007.
[10] H. M. Deitel, P. J. Deitel, "Visual C# 2005: How To Program", 2nd ed, Pearson; 2006
[11] P. J. Deitel, H. M. Deitel, "Visual C# 2008: How To Program", 2nd ed, Pearson; 2009.
[12] ECMA, "ECMA-334: C# Language Specification," 4th ed, ECMAInternational, 2006.
[13] ECMA, "ECMA-335: Common Language Infrastructure (CLI): Partitions I to VI,", ECMAInternational, 2006.
[14] Carlo Ghezzi, Mehdi Jazayeri, "Programming Language Concepts," 3rd ed, Wiley, 1997.
[15] James Gosling, Bill Joy, Guy Steele, "The Java Language Specification," Addison Wesley, 1996.
[16] José O. Guimaräes, "*The Green Language Type System*", in Computer Languages, Systems, Vol 35, Dec 2009, pp 435-447, Elsevier
[17] Bernaridho I. Hutabarat, Ketut E. Purnama, Mochamad Hariadi, "Module, Modular Programming, and Module-based Encapsulation: Critiques and Solutions" The 5th International Conference on Information & Communication, Technology, and Systems (ICTS); 2009.
[18] Bernaridho I. Hutabarat; *Programming Concepts: with NUSA Programming Language*; Ma Chung Press; 2010.
[19] IBM, "Data Type Categories," http://publib.boulder.ibm.com /infocenter/tivihelp/v8r1/index.jsp? topic=/com.ibm.netcool_impact. doc/im31sg/xF1996263.html, IBM, 2002.
[20] Carine Lucas, Patrick Steyaert;, "Modular Inheritance of Objects Through Mixin-Methods" Technical Report, Vrije Universiteit Brussel; 2007.
[21] Microsoft*; Data Types (Transact-SQL)* in msdn. microsoft.com/en-us/library/ms187752.aspx, November, 2009.
[22] Peter Müller, *Konzepte objektorientierter Programmierung*, Lecture Notes; 2007.
[23] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, Dave Clarke, "*Towards a Model of Encapsulation*," IWACO (International Workshop on Aliasing, Confinement and Ownership in object-oriented programming) ECOOP, 2003.
[24] Oracle, "Developers Guide," Oracle Corp., 2010.
[25] Robert W. Sebesta, "*Programming Language Concepts*", 8th ed., Addison Wesley, 2006.
[26] Norbert Schirmer, "*Analysing the Java Package/Access Concepts in Isabelle/HOL*", Concurrency and Computation: Practice and Experience, John Wiley and Sons; 2003.
[27] Patrick Steyaert, Open Design of Object-Oriented Languages: A Foundation for Specialisable Reflective Language Frameworks, Department Informatica, Vrije Universiteit Brussel; 1994.
[28] Bjarne Stroustrup. The C++ Programming Language. 3rd ed, Addison Wesley; 1997
[29] Andrew S. Tanenbaum; Computer Networks, 4th ed; Prentice Hall; 2003.
[30] David Ungar, Craig Chambers, Bay-Wei Chang and Urs Hölzle; "Organizing programs without classes," Springer, 2005.
[31] David A. Watt, "Programming Language Design Concepts," in John Wiley and Sons; 2004
[32] Wikipedia, *Common Type Systems*, http://en.wikipedia.org/wiki/ Common_Type_System; Wikipedia;available 2009