Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TOWARD THE MATURITY OF SOFTWARE ENGINEERING: UNIVERSAL, FORMAL, AND MATHEMATICAL DEFINITION FOR TYPE AND OBJECT AS TWO DISJOINT BASIC CONCEPTS

[1]BERNARIDHO I HUTABARAT, [2]KETUT E PURNAMA, [3]MOCHAMAD HARIADI

[1] Student, Department of Electrical Engineering, ITS, Surabaya

[2] Lecturer, Department of Electrical Engineering, ITS, Surabaya

[3]Assoc Prof, Department of Electrical Engineering, ITS, Surabaya

E-mail: [1]bernaridho@gmail.com, [2]ketut@ee.its.ac.id , [3]mochar@ee.its.ac.id

ABSTRACT

Software engineering has not reached maturity level as classic engineering. Theoretical foundation for software engineering lacks the precision and universal agreement of terms. By contrast, classic engineering are founded on the seven base dimensions that are precise and universally agreed.

This paper aims to bring software engineering into maturity, in terms of the precision of terms by establishing and mathematically defining two basic concepts: type and object. Just like the seven base dimensions in physics be part of theoretical foundation for classic engineering, the two basic concepts type and object are the theoretical foundation for software engineering.

This paper lists twelve problems with current definitions of type and object. The proposed definition and concepts are linguistically tested and mathematically formulated using thirty five formulas. Each concept – type, object – is unique and has single interpretation. This paper shows that class is a derived concept – not a basic concept – and that class can be defined on the proposed disjoint basic concepts: type and object.

Keywords: *Type, Object, Conceptual Integrity, Basic Concept, Engineering*

1. INTRODUCTION

Engineering books such as [1] excluded Software Engineering as engineering branch. Reference [1] does not write any reasons for rejection, but software engineering lack of well-definedness of something similar to the seven base dimensions is perhaps the primary factor.

Many software engineering books and research papers have been written. Few – if any – attempt to solve the above very important problem. This paper is an attempt to solve the problem.

The authors of this paper examine the problems with the prevalent theory: Object-Orientation. The authors examine the definitions in standards, textbooks, research papers, and webpages about Object-Orientations (or Object-Oriented [2]).

In examining the problems and proposing the solutions, two approaches are used: **linguistic** and mathematic. *It is the linguistic approach that has not been taken extensively by any previous paper*. Despite the proliferation of mentions about formal approach in software engineering research paper, *informal explanations dominate the research paper, textbooks, and international standards*. It is the informal text explaining the formal things (e.g., programming-language, equations) that has not been exposed to scrutiny.

Fig 1 shows the total seven base dimensions and several derived dimensions in physics that underlie classic engineering. There are two characteristics worth mentioning. Firstly, the number of base dimensions is fixed; while the number of derived dimensions can vary over time. Secondly, all derived concepts are based on base dimensions.

Having those two properties is the consequence of this paper's aim. Adjusting to the proposed concepts at hand, the authors aim to establish four basis concepts that are fixed forever. The second similar property is that all derived concepts should always be based on the basic concepts. These two properties are absent in software engineering.

Fig 2 shows the idea in which there are only four basic concepts, and all derived concepts are based on the basic concepts. The scope of this paper is the two of the four basic concepts: type and object.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

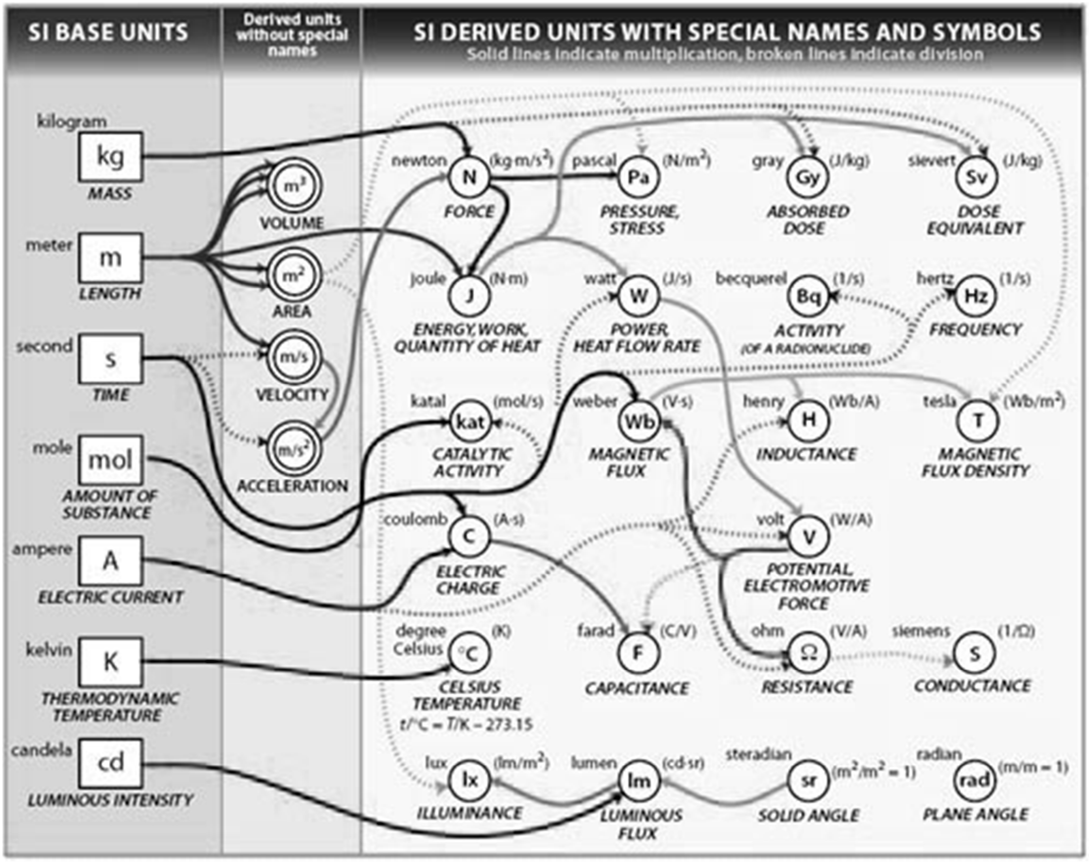


Fig 1 Base-dimensions and Derived-dimensions in Physics, foundation for Classic Engineering

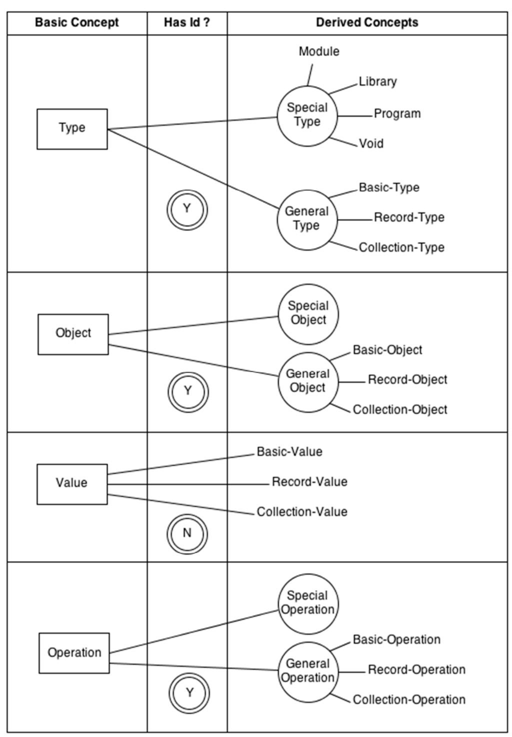


Fig 2 Proposed Basic-concepts and Derived-concepts for Software Engineering.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

This paper is organized as follows. The first section introduces the subject, the problem, and scope of the solution. The second section lists twelve (12) problems in Object-Orientation from research papers, international standards, textbooks, and webpages. The third section describes the solutions: type, object, and their disjointness are mathematically formulated using propositional calculus (set theory). The solution is guided by the paraphrased definition of conceptual integrity. The fourth section applies the proposed theory to solve the problems in the second section. The fifth section presents NUSA programming-language that shows how a non-OOPL can have expressive power of OOPLs (encapsulation, inheritance, polymorphism) without resorting to class. The sixth section compares the old versus new theory. The seventh section concludes the study. Appendix 1 explains the universal definition of class based on Basic Concepts. Appendix 2 explains non-universal definition of class, based on the basic concepts. Appendix 3 proves that class is derived concept, similar to the way of physics.

2. THEORY OF OBJECT-ORIENTATION

2.1 The Informality of Theories

One of weaknesses of OO theory is presented in ref [3]. Bertino and Martino wrote.

> Object-Oriented systems can be classified into two main categories: systems supporting the notion of *class* and those supporting the notion of *type* ... Although there are no clear lines of demarcation between them"

The second example is from Grady Booch, the key author of UML who in ref [4] defines class as

> The terms **class** and *type* are usually (but not always) interchangeable, a **class** is slightly different concept than a *type*, in that it emphasizes the importance of hierarchies of **class**es.

There are two deficiencies to find in the definition of class:

- It is informal (and consequently imprecise): note the word **usually**
- It is incorrect. The notion of type hierarchy also exists. The term **class hierarchy** does not give more emphasis than *type hierarchy*.

The vagueness, the ambiguity, and the lack of well-founded boundaries have given way to the obscurity of definitions. It is up to the individual authors to come up with their opinion regarding the difference between class and type. UML standards [5, 6] do not define what object-orientation is.

2.2 Lack of universality

Definitions of object are ad hoc. We provide four examples among literally thousands of webpages containing the definition of object. Following each sample definition is one or two thought-provoking questions. A webpage from Monash University (www.csse.monash.edu.au/damian/papers/PDF/cyberdigest.pdf, accessed 2011-07-19) defines

> an *object* is anything that provides a way to locate, access, modify, and secure data.

Fig 3 First Sample Definition of Object

One valid question for the first sample definition is "If a procedure provides a way to locate data, is the procedure an object?".

The second sample comes from webpage www.wordiq.com/definition/Object-oriented (accessed 2011-07-01). It defines object as

> Packaging data and functionality together into units within a running computer program; *object*s are the basis of modularity and structure in an object-oriented computer program.

Fig 4 Second Sample Definition of Object

One valid question for the second definition is: "has there been no modular program before Object-Oriented Programming Languages were made?" The webpage does not answer the question.

The third sample comes from SAP - a giant software corporation. Authorized personnels in http://help.sap.com/saphelp/nw2004s/helpdata/en/c3/225b5654f411d19... (accessed 2011-07-19) define object as

> These *object*s are first defined by their character and their properties which are represented by their internal structure and their attributes (data).

Fig 5 Third Sample Definition of Object

Two valid questions for the second definition are "What is the character of an object?" and "Why do other authors not define that 'character of an object defines the object'?"

The fourth example of webpage defining the object is www.slideshare.net/rickogden/beginners-guide-to-object-orientation (accessed 2011-07-19).

> An *object* is created by creating a new instance of a **class**. *Object*s of the same **class** have exactly the same functionality, but the **properties** within the object are what makes them different.

Fig 6 Fourth Sample Definition of Object

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

That definition is keyword dependent, being dependent on the usage of keyword **class**. One valid question is "Oracle Corp claims that Oracle PL/SQL is object-oriented, and Borland claims that Turbo Pascal 5.5 up to Turbo Pascal 7.0 is object-oriented; but those programming-languages do not have class. Does the author mean that Oracle PL/SQL and Turbo Pascal are not object-oriented language just because they do not have any classes?" Referring to three previous examples, the webpage does not answer those questions.

In several previous paragraphs the authors above mentioned webpages written by both academics and giant corporations. The ad hoc definitions they have given do not match the expectation of high degree of precision and trustworthiness of writings.

2.3 Redundancy of the term object and instance

Object and instance are redundant in OO literature. The fourth sample definition in sec II.B “Lack-of-universality” serves as an example, yet this problem is not limited to including the webpages not trusted by academic community. International standards suffer the same problem. C++ standard [7] sec 1.9 contains these two statements:

> An **instance** of each *object* with automatic storage duration (3.7.2) is associated with each entry into its block. Such an *object* exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal)

Fig 7 Fifth Sample Definition of Object

Here is another example from research paper [8]. The term **instance** is redundant.

> An *object* is an **instance** of a **class**.

Fig 8 Sixth Sample Definition of Object

2.4 Redundancy of the term object and instance

This is the consequence of the absence of formal differences among class and type. Here is an example copied from http://www.delphibasics.co.uk/Article.asp? Name=OO.

> We have defined a **class** called TSimple as a new ***data*** *type*.

Fig 9 Sample Sentence Containing Redundant Terms: Class and Data Type

The redundancy is also present in the C++ Standard [7]. The initial sentence of chap 9 (titled "Class") in the standard equates class to type. Class is redundant.

> A **class** is a *type*.

Fig 10 Sample sentence containing redundant terms: class and type

The redundancy is present in subsec 4.2.16 of Java standard [9] that is boxed in Fig 11. It shows the redundancy and vagueness.

> We often use the term *type* to refer to either a **class** or an **interface**.

Fig 11 Sample Sentence Containing Redundant Terms: Type, Class, and Interface; with No Clear Boundaries

Redundancy also happens in relation to the way both terms are used. Here is an example taken from a research paper [10].

> "Generics for the Masses" (GM) and "Scrap your Boilerplate" (SYB) are generic programming approaches based on some ingenious applications of Haskell **type class**es.

Fig 12 Sample Sentence Containing Term that is Unnecessarily Long: Type Class

Reference [11] is a paper about Java type qualifier. One statement contains redundancy regarding class and type (see Fig 13).

> For any **class** C, a reference of *type* readonly C may not be used to modify the *object* it refers to, which is a particularly useful annotation for **method** arguments and results.

Fig 13 Another Sample Sentence Containing Redundant Terms: Class and Type

Reference [12] is a research paper titled "Typeless programming" which contains this boxed sentence below with the word "type".

> The term Vector< super Vector< extends List<Integer>> is for example a correct *type* in Java 5.0.

Fig 14 A Sample Sentence Containing the Term Type

Yet the source-code that follows contains the word "class" instead of type. The code explaining that statement starts with this line (note the word **class** instead of *type*).

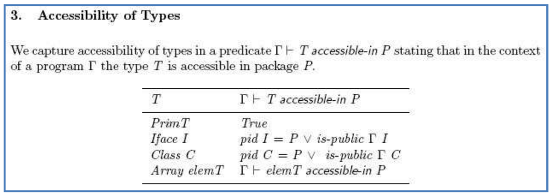
```
class Matrix extends Vector<Vector <Integer> >
```

Fig 15 A Source-code Containing the Word Class, Inconsistent with the Sentence Preceding It.

Fig 16 shows the latest example in this subsec,

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

taken from [13] sec 3. The author speaks about type; however, the third entry contains class.



3. Accessibility of Types

We capture accessibility of types in a predicate $\Gamma \vdash T$ *accessible-in* P stating that in the context of a program Γ the type T is accessible in package P.

T	$\Gamma \vdash T$ *accessible-in* P
PrimT	*True*
Iface I	*pid* $I = P \vee$ *is-public* Γ I
Class C	*pid* $C = P \vee$ *is-public* Γ C
Array elemT	$\Gamma \vdash$ *elemT accessible-in* P

Fig 16 Redundancy of Terms: the Heading and Human Language Sentences are Inconsistent with the Mathematical Formula

2.5 Conceptual disintegrity: type equals object

Type should not be equal to object, and vice versa. Oracle PL/SQL equates type with object as pictured in Fig 17. PL/SQL programming-language creators confused object with type.

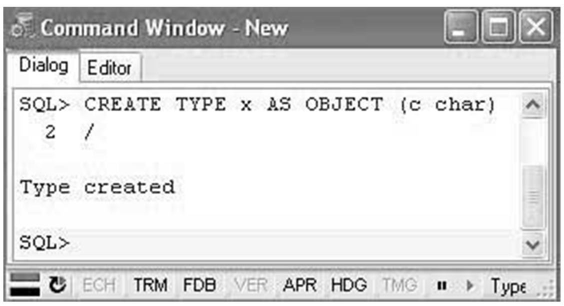


Fig 17 Creating the Type in Oracle PL/SQL

We test our hypothesis by creating an object. If object = type, then the contents of metadata-view **`user_objects`** and **`user_types`** will be the same. But fig 18 shows the contents are different. Hence:

- ➢ The concept of type looks to be equal to the concept of object, but
- ➢ The concept of type is different from that of object (hence a contradiction, a conceptual disintegrity).
- ➢ The differences are not precisely formulated.

Similar disintegrity takes place in subsection 8.2.1 of C# Standard [14] which contains this entry for object. The header says Type but the entry says object. C# standard committee equates type with object (see Table II).

Reference [15] also mistook object with type. In one of the definition the author wrote "*objects are implementations of abstract data types*."

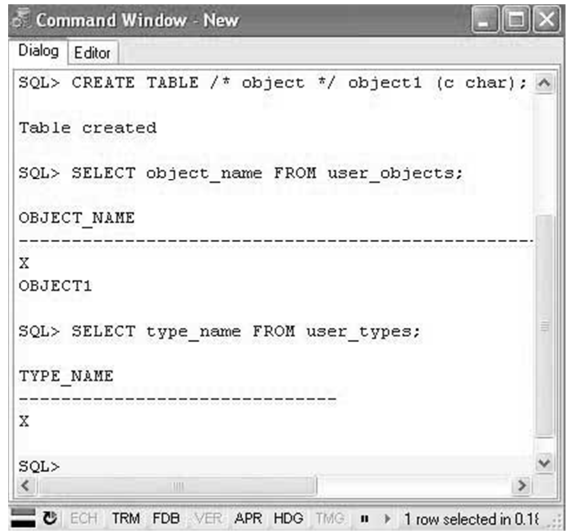


Fig 18 Incorrectness of Type = Object; Type ≠ Object.

2.6 Conceptual disintegrity: class equals object

Similar mistake and problem appear in Java standard [9] sec 4.3.2; asserting that "The class Object is a superclass (8.1.4) of all other classes". Object is said to be equal to class. Table II tabulates partial content of C# standard [14] subsection 8.2.1 that shows the essentially same mistake.

Table II. Class equals Object

Type	Description	Example
object	The ultimate base type of all other types	object o = null;

2.7 Conceptual disintegrity: object equals intermediate-code

The disintegrity of concepts becomes evident. We start with the notion of **object code**. The object code means **intermediate code** in the code translation textbooks such as [16]. The intermediate code is the result of the compilation of source code.

According to [17] there are two possible definitions for object:

- object = intermediate code
- object code = intermediate code

Both equations are proved to be false, shown below through modus Tollens. The modus is formalized as follows:

if *p* then *q*
~*q*
then ~*p*

Fig 19 Modus Tollens

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Let us assign that **object is the outcome of compilation process** to *p* and **object is intermediate code** to *q*. To show the imprecision in IEEE definition, we must show that ~*q* holds. Consequently, ~*p* holds too. While the ~*q* is read as **object it is NOT intermediate code**. It is the fact that object is not intermediate code. We can then conclude that **object is NOT the outcome of compilation process**.

2.8 Incorrect semantics

Theories within Object-Orientation literature can be semantically incorrect. The repetition of words is evident from the IEEE definition and some other example texts in the standard. We explore the second possibility of equality in the previous section: object = intermediate code. If that equality holds, the second equality cannot hold. Linguistically, the two equalities must be presented like the ones below.

- object = intermediate code
- object code = intermediate code code

Notice the repetition of the word code. Fig 21 shows another example exhibiting similar problem.

A possible improvement is done by writing two equalities for IEEE Standard Glossary of Software Engineering Terminology which are shown below:

- object = intermediate
- object code = intermediate code

The last proposed equation is free from the repeated words problem. But the equality of object = intermediate is in conflict with any English dictionary [18]. No English dictionary equates object with intermediate or intermediate code.

Incorrect semantics is also shown through substitutions test [19]. Let us take one example from Java standard [9].

4.3.1 An *object* is a **class instance** or an array. 4.3.2 A **Class object** exists for each reference *type*.

Fig 20 Two sentences that will be tested for semantics correctness

If the semantics is correct, the semantics of this statement below should be correct. We substitute object with 'class instance'.

A **Class class instance** exists for each reference *type*.

Fig 21 Example of Incorrect Semantics Found through Words Substitution

However, the sentence is semantically incorrect due to the double word 'Class class'. Applying the substitution test for the entire text of the standard will result in more occurences of semantic error.

2.9 Confusing word order

The following boxed text is contained in subsection 17.1.2.1 of C# Standard [14]. Note there are two phrases in which the difference is only on the word order: class object versus object class.

Except for **class object**, every class has exactly one direct base class. The **object class** has no direct base class and is the ultimate base class of all other classes.

Fig 22 Confusing Word Order

2.10 Difficult to understand concepts

We argue that the concept like instance is hard to understand. Consequently, the concepts such as 'instance variables' are even harder to understand. Reference [20] mentioned the difficulty in explaining the concept of instance.

Järvi, Marcus, and Smith have offered strikingly different programming concepts that are limited to C++ (one of Object-Oriented programming-languages). They created a class named concept, like this one from ref [21].

```
concept LessThanComparable<typename T>
{
 bool operator < (const T& a, const T& b);
 bool operator > (const T& a, const T& b)
 {return b < a;}
 bool operator <= (const T& a, const T& b)
 {return !(b < a); }
 bool operator >= (const T& a, const T& b)
 {return !(a < b); }
}
```

Fig 23 The Class Implemented as Concept

2.11 Difficult to understand concepts

The claim that everything is an object was made by Adele Goldberg and David Robson in their book about Smalltalk [22]. It has been rejected by TTM community [23] but favored by [24]. Interestingly [24] listed one step "Acquire the Class Concept by Abstraction of Many Common Objects". It is a contradiction to "Everything is an object". If everything is an object, we do not need class. In a previous book, Adele Goldberg (with Alan Kay) equated value with object (ref [25] page 12).

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

2.12 No concept is defined

Some research papers (e.g. [26]) do not define any concept). The authors in [26] attempted to explain Object-Orientation using logic. There is no definition of object, type, class, methods, and the usual terms in Object-Orientation. The author of [27] refers **properties** in C# as syntactic sugar. If property is a **syntactic sugar**, it deserves neither a notion of Basic Concept nor definition.

3. PROPOSED SOLUTIONS

3.1 Integrity: irreducibility and unity

Brooks in [28, 29] has written about **conceptual integrity** but he does not define it. References [28, 29] only wrote "*Conceptual integrity is the most important consideration in system design*".

We define conceptual integrity as the *integrity of concepts*, and consequently include the **unity** of concepts. The unity of concepts *precludes the redundancy of concept*s. The system that lacks conceptual integrity has conceptual disintegrity. As the systems lack the unity of concepts, it embodies redundant and incoherent concepts.

References [28, 29] wrote "It *is better to have a system that omit certain anomalous features and improvements, but it still reflects one set of better ideas, than to have one that contains many good, yet providing uncoordinated ideas*". If we remove "good but" from the original sentence; and replace *features*, *improvements*, and *ideas* with **concepts** we get this slightly paraphrased sentence:

> It is better to have a system omit certain anomalous **concepts**, but to reflect one set of good **concepts**, than to have one that contains many uncoordinated **concepts**.

Fig 24 Proposed Theory About the System Having Conceptual Integrity

That statement will be the basis of this paper. We hypothesize that **class** and **instance** are anomalous and redundant concepts. We hypothesize that *object* is also an anomalous concept in Object-Oriented literature, but not anomalous if defined precisely. We propose a precise definition for the *object* concept in this paper.

3.2 Unique Basic Concepts and Their Informal definitions

We introduce four basic concepts along with their informal definitions: VOTO, abbreviation for Value Operation Type Object in [30]. Our proposed basic concepts are similar to the four core concepts proposed in [23]. We use 'object' (instead of variable AS in [23]) because

(1) Not all objects are variables; some objects are constants [30].

(2) Even further, not all objects are value-assignable [30].

Some objects cannot be assigned to values; hence some objects are neither constants nor variables. A good informal definition of object can be found in C standard [31] that in sec 3.14 defines object as:

> region of data storage in the execution environment, the contents of which can represent values.

The following subsections contain informal definitions for basic concepts that were partially written in ref [30].

3.2.1 Type

Type is defined as follows:

- Types are first categorized into metatypes (MT) and nonmetatypes (NMTs)
- A type may or may not have identity.
- TypeCategories := {General-types, Special-types}
- General-types contain values.
- Special-types contain no values.
- General-types := {Basic-types, Record-types, Collection-types}
- SpecialTypes := {void, Module, Program}

3.2.2 Object

Object is defined as follows:

- An object has identity.
- An object is of some type.
- Objects of Special-types cannot have value
- Objects of General-types are General-objects
- Objects of Special-types are Special-objects
- General-objects have value
- Special-objects do not have value

3.2.3 Disjointness of types and objects

The disjointness of types and objects are formulated formally as follows:

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- Object is not type.
- Type is not object.

Therefore

- ➢ The concepts of object and type are disjoint (see Fig 25).
- ➢ Individual objects are disjoint from individual types.

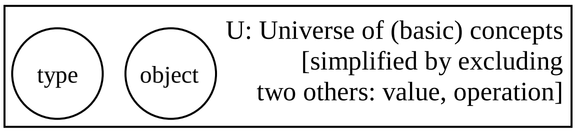


Fig. 25 Type and Object are Disjoint Concepts

3.3 Categorization of Basic Concepts into Basic, Collection, and Record

The author of [30] proposed orthogonal categorization toward basic concepts that implied the presence of twelve derived concepts (Fig. 26):

- Basic-type
- Basic-object
- Basic-value
- Basic-operation
- Record-type
- Record-object
- Record-value
- Record-operation
- Collection-type
- Collection-object
- Collection-value
- Collection-operation

Those derived concepts will prove to be sufficient and useful for the rewritten theories and sentences in section 4.

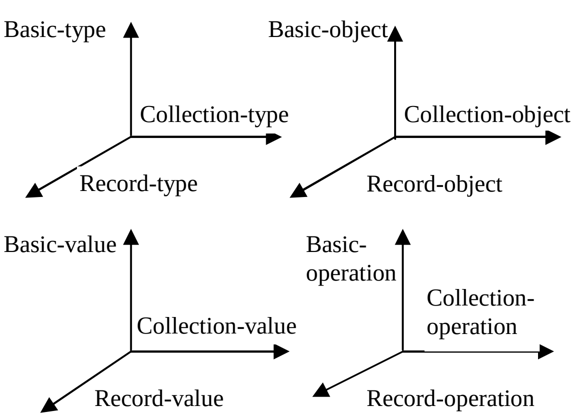


Fig 26 Formal Definition for Basic Concepts; Fulfill the Principle of Irreducibility and Conceptual Integrity.

3.4 Unique Basic Concepts and Their Formal definitions

The concepts are formal if they only have one interpretation.

Basic_Concepts := {Value, Operation, Type, Object} (1)

$\forall C_i \in$ Basic_Concepts (unique(C_i)) (1)

The basic concepts of programming introduced here have two important properties:

- Formal (Hence there is no *usually*, like the one in Booch).
- Nonredundant (type only, no redundancy with other concept like class). (2)

3.4.1 Type

These are the itemized definitions for type.

- $\forall$ t $\in$ Types HasId(t) $\wedge$ HasNoId(t) (3)
- Types = MetaType $\cup$ NonMetaTypes (4)
- Metatype $\cap$ NonMetaTypes = $\varnothing$ (5)
- NonMetaTypes = GeneralTypes $\cup$ SpecialTypes (6)
- $\forall$ t $\in$ General-types SetOfValues(t) $\neq \varnothing$ (7)
- $\forall$ t $\in$ Special-types SetOfValues(t) = $\varnothing$ (8)
- General-types = Basic-types $\cup$ Record-types $\cup$ Collection-types (9)
- Basic-types $\cap$ Record-types = $\varnothing$ (10)
- Basic-types $\cap$ Collection-types = $\varnothing$ (11)
- Record-types $\cap$ Collection-types = $\varnothing$ (12)

3.4.2 Object

These are the itemized definitions for object.

- $\forall$ o $\in$ Objects (has_id (o)) (13)
- $\forall$ o $\in$ Objects $\exists$ t $\in$ Types (IsOfType (o, t)) (14)
- Objects = SpecialObjects $\cup$ GeneralObjects (15)
- SpecialObjects $\cap$ GeneralObjects = $\varnothing$ (16)
- GeneralObjects = Basic-objects $\cup$ Record-objects $\cup$ Collection-objects (17)
- Basic-objects $\cap$ Record-objects = $\varnothing$ (18)
- Basic-objects $\cap$ Collection-objects = $\varnothing$ (19)
- Record-objects $\cap$ Collection-objects = $\varnothing$ (20)

3.4.3 Disjointness of types and objects

The disjointness is formulated as follows:

- $\forall$ t $\in$ Types $\forall$ o $\in$ Objects (o $\neq$ t) (21)
- Types $\cap$ Objects = $\varnothing$ (22)
- $\forall$ t $\in$ Types unqouted (lowercase(id(t))) $\neq$ object (23)

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- ∀ o ∊ Objects unquoted (lowercase(id(o))) ≠ type (24)

3.4.4 Category Relation

The category is defined as a relation that is transitive. The relation is denoted by symbol ⩽. The words denoting the operands to the operation ⩽ is singular. The application of the category relation to types is listed below:

- Metatype ⩽ Type (25)
- NonMetaType ⩽ Type (26)
- General-type ⩽ NonMetaType (27)
- Special-type ⩽ NonMetaType (28)
- Basic-type ⩽ General-type (29)
- Record-type ⩽ General-type (30)
- Collection-type ⩽ General-type (31)

The transitivity makes for these relations for types

- General-type ⩽ Type (through NonMetatype) (27a)
- Special-type ⩽ Type (through NonMetatype) (28a)
- Basic-type ⩽ Type (through General-type and NonMetaType) (29a)
- Record-type ⩽ Type (through General-type and NonMetaType) (30a)
- Collection-type ⩽ Type (through General-type and NonMetaType) (31a)

Formulas #4 through #12 are partially captured in the six formulations below

- A general-type is a type
- A special-type is a type
- A basic-type is a type
- A record-type is a type
- A collection-type is a type

Concerning object and value we can write

- General-object ⩽ Object (32)
- ∀ GO ∊ General-objects, has_value (GO) (33)

Proof:

∀ o ∊ Objects ∃ t ∊ Types (IsOfType (o, t)) (14)

∀ t ∊ General-types SetOfValues(t) ≠ ∅ (7)

3.5 Hypothesis: equivalent synonyms

In this section we list synonyms of one word in original text. The presence of multiple equivalent terms is due to the careless wording in textbooks and international standards.

Table III Equivalent Synonyms

No	Original	Equivalent
1	Class	Type
2	Class	record-type
3	Class	Record
4	Instance	Object
5	Subclass	derived-type
6	Property	Value
7	Property	Operation
8	Member	Column
9	Variable	Object

3.6 Hypothesis: essetially equivalent phrases

It is impossible to list all equivalent phrases. Sample equivalent phrases are listed in Table IV.

Table IV Equivalent Phrases

No	Original	Equivalent
1	data type	Type
2	data type	record-type
3	class type	record-type
4	base class	base-type
5	sub object	Column
6	Subobject	Column
7	member subobject	Column
8	data member	Column
9	function member	Operation

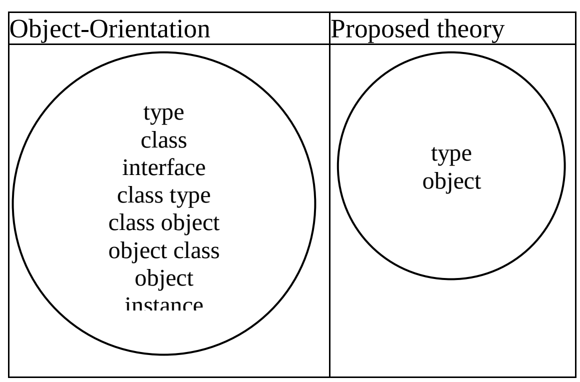


Fig 27 Object-Orientation is Theory with Highly Redundant Terms. Proposed Theory Contains No Redundancy

Fig 27 and fig 28 summarize the comparison of theories in graphical way (sec 6 details the comparison of theories). While there are derived terms (like record-type) in the proposed theory, the most important concepts are type and object.

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

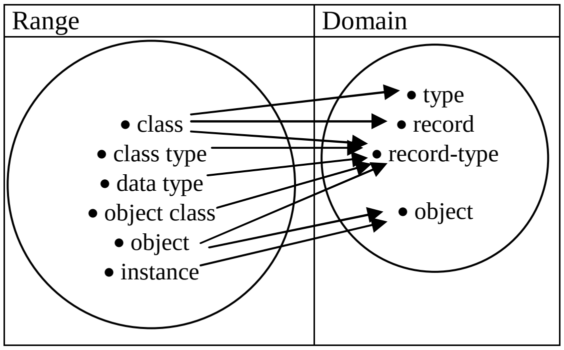


Fig 28 Partial Mapping of Terms; Graphical Version of Table II and Table III

3.7 Scope of the solutions

Literatures about Object-Orientation often relate class and object with **method**. Method is operation. The first author of this paper explained methods in [32], contrasted Module-based Encapsulation versus Type-based Encapsulation in [33], and related module to namespace in [34]. Methods have also been explained in the light of orthogonality in [35]. NUSA is a programming-language that adopts specific approach of Module-based Encapsulation. In that approach, type cannot contain operation or operator [33]. This approach is also adopted by TTM in Other Orthogonal Very Strong Suggestion number 2 “Types and operators unbundled" [36].

Solving the problems listed in the previous section is a prerequisite requirement before solving the problems of defining methods in the precise way. All of these mount to the decision of excluding the treatment of methods (in Object-Orientation folklore) in this paper.

3.8 Column and Arity

In this paper, the term column is used instead of **field**, **attribute**, and data member. A column is an object, but an object is not necessarily a column.

A C++ source-code below shows that object1 is a column, and necessarily an object. On the other and, object2 is an object; but not a column.

```
typedef struct Type1
{
  public: char object1;}}
};

void main()
{
  struct Type1 Object2;}}
  Object2.object1 = 'y';
  // there are two objects
}
```

We denote the number of columns in record-type as well as record-object using the function Arity.

- $\forall$ RT $\in$ Record-types Arity(RT) $\geq$ 0 (34)
- $\forall$ RO $\in$ Record-objects Arity(RO) $\geq$ 0 (35)

3.9 Hypothesis on some common conventions

Programmers and authors have some conventions on writing the source-code. Some of the conventions are prefixing the type name with T (for Type) or C for (Class). Programming library Turbo Vision from Borland use prefix T. Microsoft in its .NET programming library use prefix C (msdn.microsoft.com/en-us/library/20t753se.aspx , accessed 2012-05-03, is an example).

4. RESULTS (PROVING THE SOLUTIONS)

4. 1 Replacing instance by object

We take an example repeated from sec II.B and name it as ***Sentences 1a*** to serve as an example how we can make better explanation.

An *object* is created by creating a new **instance** of a **class**. *Object*s of the same **class** have exactly the same functionality, but the **properties** within the *object*s are what make them different.

The following ***Sentences 1b*** are the result of replacing instance by object, with the mapping #4 in Table II.

An *object* is created by creating a new *object* of a **class**. *Object*s of the same **class** have exactly the same functionality, but the **properties** within the *object*s are what make them different.

The mapping shows there is no new information in the first sentence "An object is created by creating a new object". It is redundant.

4. 2 Replacing class by type

We start this section by improving the rewritten sentences in the previous section. We apply mapping #2 in Table II: replace class by type for all rewritten sentences within this subsection. ***Sentences 1c*** are the result of the first rewrite.

An object is created by creating a new *object* of a *type*. *Object*s of the same *type* have exactly the same functionality, but the **properties** within the *object*s are what make them different.

The rewritten text is clearer. The concepts are becoming integrated without redundancy. Here is

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

another example, ***Sentence 2a***, that is copied from www.delphibasics.co.uk/Article.asp? Name=OO.

We have defined a **class** called TSimple as a new data *type*.

We use mapping #2 in Table II to replace class by type. ***Sentence 2b*** is the result having the conceptual integrity.

We have defined a *type* called TSimple as a new data *type*.

We refer to the sentence below as ***Sentence 3a***. It is written in Sec 9.2 of C++ standard.

A **class** definition introduces a new *type*.

This is ***Sentence 3b*** obtained by replacing class with type, and introduces with defines.

A *type* definition defines a new *type*.

4. 3 Using equivalent synonyms

We start this section by improving the rewritten ***Sentences 1c*** in the previous section. We replace properties with values using the mapping #6 in Table II. The result is ***Sentences 1d*** below.

An *object* is created by creating a new *object* of a *type*. *Object*s of the same *type* have exactly the same functionality, but the *value*s within the *object*s are what make them different.

Sometimes replacing the words by means of equivalent phrases is better. Using the mapping #3 in Table III we replace **data type** in ***Sentence 2b*** with *record-type*. The result is ***Sentence 2c*** below.

We have defined a *type* called TSimple as a new *record-type*.

The resulting text will be compared against the improvement of source-code in sec 4.6. In the subsequent sentences, we cover the more complex sentences.

The next sentence is taken from point 1 within Chap 9 (chapter about Classes) in C++ Standard [7]. We call it ***Sentence 4a***.

An *object* of a **class** consists of a (possibly empty) sequence of **member**s and base **class** *object*s.

We use the mapping #2 in Table II to replace class by record-type, mapping #6 in the same table to replace member with column, and replace one of the word object with column. We add - after the word base and remove the space before the word type (the word type that substitutes the word class). The result is ***Sentence 4b***.

An *object* of a *record-type* consists of a (possibly empty) sequence of *column*s and base-*type* *column*s.

In the conversion of two subsequent original sentences, we replace class with record-type (mapping #2 in Table II). The original sentences are taken from chapter 3 point 3 in C++ standard [7]. We call the first one as ***Sentence 5a*** (see below).

Note: **class** *object*s can be assigned, passed as arguments to functions, and returned by functions.

We rewrite the previous sentence by replacing class with record-type. The result is ***Sentence 5b***.

Note: *record-type* *object*s can be assigned, passed as arguments to functions, and returned by functions.

This is another sentence from the C++ standard [7] chapter 9 and the same point (3). We call it ***Sentence 6a***.

(except objects of **class**es for which copying has been restricted; see 12.8).

We apply the same mapping to replace class with record-type. The result is ***Sentence 6b*** below.

(except objects of *record-type*s for which copying has been restricted; see 12.8).

4. 4 Using equivalent phrases

The rewriting of sentences can be complex. In this section we convert the phrases of sentences. Point 3 in chap 9 of the C++ standard [7] is written as what we call ***Sentence 7a*** below.

Complete *object*s and **member subobject**s of a **class** *type* shall have nonzero size.

We apply the mapping #3 (class type with record-type) and mapping #7 (replace **member subobject** with column) for rewriting; both are from Table III. The result is ***Sentence 7b*** below.

Complete *object*s and *column*s of a *record-type* shall have nonzero size.

In which they can be rewritten as two sentences ***Sentence 7c*** and ***Sentence 7d*** to make explanation more explicit about C++.

Complete *object*s of a *record-type* shall have

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

nonzero size. *Columns* of a *record-type* shall have nonzero size.

Similar rewriting (replacing class type with record-type, mapping #3 in Table III) can be applied to another sentence within C++ standard. This is the Point 4 in chap 9 of the C++ Standard. We call it ***Sentence 8a***.

Note: aggregates of **class** *type* are described in 8.5.1.

Using the mapping #3 in Table III (replace class type with record–type) we obtain ***Sentence 8b***.

Note: aggregates of *record–type* are described in 8.5.1.

The next example for this subsection comes from point 4 of chap 9 of C++ standard which we refer to as ***Sentence 9a***. POD is short for Plain Old Data. That term is unnecessary.

A **POD-struct** is an aggregate **class** that has no **non-static data members** of type **non-POD-struct**, **non-POD-union** (or array of such types) or **reference**, and has no user-declared copy assignment operator and no user-declared destructor.

We rewrite ***Sentence 9a*** by replacing POD-struct with struct, class with record-type, non-static with dynamic, data member with column, reference with address, and operator with operation. The result is ***Sentence 9b*** below.

A struct is an aggregate *record-type* that has no *dynamic columns* of type **non-struct**, **non-union** (or array of such types) or *address*, and has no user-defined copy assignment *operation* and no user-defined destructor.

4. 5 Paraphrasing

Rethinking further, ***Sentence 9b*** can be improved by paraphrasing to be ***Sentence 9c*** below. C++ has two record-type-qualifiers: `struct` and `class`. The word aggregate is not needed.

A ***record-type with qualifier struct*** has no *dynamic columns* of type *non-struct*, *non-struct-union* (or array of such types) or *address*; and has no ***user-defined*** copy assignment *operation* and no ***user-defined*** destructor.

Other paraphrasing techniques that results in better explanation are discussed in the following two subsections.

4.5.1 Changing the word order

The words constituting phrases 'record-type objects' are paraphrased into 'objects of record-type'. We apply the rule to rewrite the end of ***Sentence 4b*** ('record-type objects') to be 'objects of record-type' in ***Sentence 4c*** below.

An *object* of a *record-type* consists of a (possibly empty) sequence of *columns* and base-*type columns*.

We can apply a similar rule to rewrite ***Sentence 5b*** to be ***Sentence 5c*** below.

Note: *objects* of *record-type* can be assigned, passed as operands to functions, and returned by functions.

4.5.2 Changing "objects of <x-type>" into "x-objects"

Objects of record-type can be paraphrased into record-objects. This paraphrasing technique allows us to rewrite ***Sentence 5c*** into ***Sentence 5d*** below.

Note: *record-objects* can be assigned, passed as operands to functions, and returned by functions.

Finally, we can also paraphrase ***Sentence 6b*** into ***Sentence 6c*** below.

(except *record-objects* for which copying has been restricted; see 12.8).

4. 6 Removing unnecessary phrases or sentences

The further result from the proposed theory is that we can remove unnecessary phrases or sentences. ***Sentence 3b*** can be removed altogether from the (C++) standard.

4. 7 Reversing the conventions, applying the theory to source-code

We apply the refinement of the theory to the refinement of source-code. We start with the theory reformulated as ***Sentence 2c*** in sec 4.3.

We have defined a *type* called TSimple as a new *record-type*.

Secondly, we check the correctness of theory (rewritten sentences) to the accompanying source-code. Here is the code obtained from www. delphibasics.co.uk/Article.asp? Name=OO.

```
type (* Define a simple class *)
  TSimple = class
```

Journal of Theoretical and Applied Information Technology
30th June 2013. Vol. 52 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
  simpleCount : Byte;
  property count : Byte read simpleCount;
  procedure SetCount (count : Byte);
end;
```

The proposed theory matches the source-code. We apply the replacement of text according to the proposed theory.

- replace the word class with record-type in the comment
- replace the word class with record (because record is record-type) in Line 2
- replace the word property with function
- replace type name TSimple with Simple (reverse the conventions, drop the prefix T from type-name)

In addition we replace = by := as assignment operation. Delphi uses the symbol = inconsistenly, it can mean comparison-operation and assignment-operation. The result of converted source-code is as follows:

```
type (* Define a simple record-type *)
 Simple := record
  simpleCount : Byte;
  function count : Byte read simpleCount;
  procedure SetCount (count : Byte);
 end;
```

The converted source-code is no longer a Delphi source-code. But the source-code reflects the good theory. The source-code can now be explained with integrated concepts, not by disintegrated concepts.

This is the refined explanation for the source-code called ***Sentence 2d***, in which the type-name TSimple has been replaced by Simple.

> We have defined a *type* called Simple as a new *record-type*.

5. APPLICATIONS IN NUSA PROGRAMMING-LANGUAGE

5.1 NUSA: language that conforms to the proposed theory

The theory that underlies NUSA programming-language conforms to the proposed theory and conceptual integrity. There is no concept of class, data member, property, and instance in NUSA. The original source-code inside sec 4.6 that is written in Delphi can be rewritten in NUSA as follows:

```
type Simple := Thing +
  record { word count; };

word count (Simple self)
{ return (self.Count); }

void SetCount (Simple& self; word count)
{ self.Count := count; }
```

That source-code can be explained and theorized without involving the concepts of class, data member, property, and instance. ***Sentence 2e*** below (rewritten from ***Sentence 2d***) theorizes the source-code concisely.

> We have defined a new *record-type* called Simple.

5.2 Type, object, name for metatype and types

NUSA uses `type` for the name of metatype. There is only one metatype in NUSA. The name of objects cannot intersect with the name of types. NUSA adheres to the formulas #1 through #33.

By conforming to formula #23 and #24 NUSA avoids two problems: confusing type with object, confusing phrases **class** *object* versus *object* **class**.

In NUSA the system-defined root record-type is named `Thing`, not `object`. It removes the possibility of mistaking type with object. Table IV explains `Thing` in NUSA. The explanation can be compared to the theories contained in subsec 8.2.1 of C# standard and subsec 4.3.2 of Java standard.

Table IV Partial table of types in NUSA

Type	Description	Example
Thing	Root system-defined base record-type for other record-types	Thing Object1;

The design also removes the confusing terms. In sec 2.9 we show example of confusing word order in C# standard. In this section we show how the confusion can be removed.

> Except for *type* Thing, every *record-type* has exactly one direct base-*type*. The *type* `Thing` has no direct base-*type* and is the ultimate base-type of all other *record-types*.

5.3 Support for inheritance and polymorphism

NUSA is similar to Tutorial D [23] in terms of unbundling the operations. The most significant difference between the two is the explicit notion of module and modular programming [32]. Unbundling the operations from record-types does not prohibit the support of polymorphism, due to the usage of namespace within NUSA [34].

Reference [34] shows that NUSA can handle inheritance. Indeed, the boxed sentence in the previous subsection implies the support of inheritance in NUSA. In this subsection we present

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

an overview of how NUSA handles polymorphism related to the inheritance.

```
Program Demo; // inheritance, polymorphism
type Type1 := Thing +
   Record { boolean column1; };
type Type2 := Type1 +
  Record { char column2; };

void operation1 (Type1 this)
// polymorphic operation, accepts operand
{ // whose type is derived from Type1
  writeline (this.column1);
}

void main ()
{
  Type2 Object2;
  // call the polymorphic operation
  operation1 (Object2);
}
```

6. COMPARISON OF THEORIES

In this section we compare the existing theories versus the proposed theory for type and object. Table V summarizes the comparison of theories. The proposed theory eliminates 12 problems associated with the existing theories.

Table V Comparison of theories

No	Existing concept/theory	Proposed concept/theory
1	Informal	Formal
2	Not universal	Universal; independent of the programming-language
3	Contains the redundancy due the term object and instance.	The term instance (and the redundancy) is removed.
4	Contains the redundancy due the term class and type.	The term class (and the redundancy) is removed.
5	Conceptual disintegrity: type equals object	Conceptual integrity: type ≠ object
6	Conceptual disintegrity: class equals object	Conceptual integrity. The term class is removed (corollary of solution #4).
7	Conceptual disintegrity: object equals intermediate-code	Conceptual integrity. Intermediate-code is not a basic concept
8	Incorrect semantic when we apply the substitution principle.	Correct semantic when we apply the substitution principle.
9	Confusing word order (e.g., of **class object** and **object class**)	No confusing word order
10	Difficult to understand concepts (e.g., the term /concept instance)	No difficult to understand concepts (e.g., **instance** and **class** are eliminated)
11	Everything is an object.	Not everything is an object (see the 4 basic concepts).
12	No concept is defined (due to informality, #1)	All concepts (type and object) are defined.

The following table summarizes the sections introducing the problems with existing theories and the solutions. We use the abbreviation sec to refer to subsection.

Table VI Solutions (and formulas) for the problems

No	Sec	Solution
1	II.A	Formal theory for type, object, and differences between type and object. Formula #1-3, 13-14, 21-24
2	II.B	Universal (language-independent) theory using the concept type even if the programming-languages use the word class. The solution is within all formulas and mappings in sec III.
3	II.C	Basic Concepts and The Formal Definitions remove the need for the term instance.
4	II.D	Basic Concepts and The Formal Definitions remove the need for the term instance.
5	II.E	Formal theory for differentiating type versus object. Subsection III.D.3.
6	II.F	Basic Concepts and The Formal Definitions remove the need for the term instance.
7	II.G	Principle solutions: (a) Basic Concepts and (b) Code-translation theory [14] that uses the term 'intermediate-code' instead of 'object-code'.
8	II.H	Principle solutions: (a) Basic Concepts and (b) Code-translation theory [14] that uses the term 'intermediate-code' instead of 'object-code'.
9	II.I	Basic Concepts and The Formal Definitions remove the confusing word

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

		order, especially subsection III.D.1. Additional solution can be inferred from section IV.B.
10	II.J	Basic Concepts and The Formal Definitions remove the need for the term instance.
11	II.K	Formal theory for type and the formal differences between type and object prove not everything is an object.
12	II.L	Basic Concepts and The Formal Definitions.

Solution for the problems #7 and #8 needs code-translation theory as described in [30] (see 'intermediate-code' in fig 29 below).

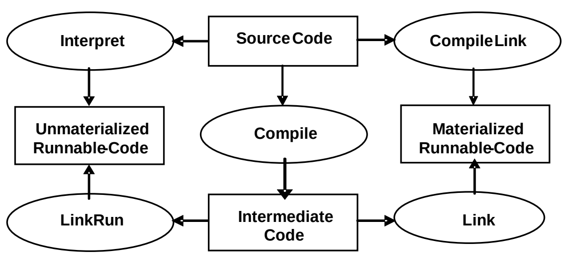


Fig 29 Code-translation processes

Sentences in the textbooks, specifications, or theories in the webpages can be perceived as theories. Table VII shows that using the theory proposed in this paper; explanations in textbooks and specifications, and more detailed theories can be written using consistent and very limited terms.

Table VII Tabulation of rewritten sentences with conceptual integrity, consistent and limited terms

No	Initial sentence(s)	Final sentences, with reference to formula
1	An *object* is created by creating a new **instance** of a **class**. *Objects* of the same **class** have exactly the same functionality, but the **properties** within the *objects* are what make them different.	*Objects* of the same *type* have exactly the same functionality, but the identities and *values* are what make them different. ***Formula (13, 14, 33)***

No	Initial sentence(s)	Final sentences, with reference to formula
2	We have defined a class called TSimple as a new data type.	We have defined a new *record-type* called Simple. ***Formula (9)***.
3	A class definition introduces a new type.	A *record-type* definition defines a new *type*. ***Formula (30a)***.
4	An object of a class consists of a (possibly empty) sequence of members and base class objects.	A *record-object* consists of a (possibly empty) sequence of *columns* and base-*type columns*. ***Formula (35)***.
5	Note: class objects can be assigned, passed as arguments to functions, and returned by functions.	Note: *record-objects* can be assigned, passed as arguments to func tions, and returned by functions ***Formula (17)***.
6	(except objects of classes for which copying has been restricted; see 12.8).	(except *record-objects* for which copying has been restricted; see 12.8). ***Formula (17)***.
7	Complete objects and member subobjects of a class type shall have nonzero size.	*Objects* and *columns* of a *record-type* shall have nonzero size. ***Formula (17, 35)***.
8	Note: aggregates of class type are described in 8.5.1.	Note: aggregates of *record-type* are described in 8.5.1. ***Formula (9)***.
9	A POD-struct is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-declared copy assignment operator and no user-declared destructor.	A *record-type* with qualifier struct has no dynamic *columns* of *type* non-struct, non-union (or array of such *types*) or address, and has no user-defined copy assignment *operation* and no user-defined destructor. ***Formula (9, 34)***.

Rewritten sentence #3 (using the word record-type) can be removed because it is redundant. Our proposed theory can be used to reduce the explanations in the specifications.

7. CONCLUSIONS

In this paper we have described the problems with existing theories underlying the Object-Oriented Programming. Existing theories lack conceptual integrity among the concepts of type, object, instance, and class. Class and instance are

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

redundant and anomalous concepts (the two appendices offer proper explanation of class).

This paper proves that the concept of object and type can be mathematically formulated. Prior to this paper, the concept of type and object are not formulated formally and uniquely. To the best of the authors' knowledge, the mathematical formula for type and object are either informal, or formal but redundant with the term class and instance.

We have proposed a theory that fulfills the conceptual integrity principle. There is neither redundant nor isolated concept. We focus on informal and formal definition of the concepts of type and object. Both concepts are disjoint, shown through 35 mathematical formulas.

To aid understanding of the proposed theory, NUSA programming-language is designed and its code-translators are created. NUSA is not an OOPL, but a general-purpose with conceptual integrity; providing encapsulation, inheritance, and polymorphism without resorting to class. Class is not required for encapsulation, inheritance, and polymorphism. Class is not a basic concept.

The independence of basic concepts formulation can be used to increase the maturity level of software engineering. One day all software engineers understand "Concepts Every Software Engineer should know". The concepts will be few, universally agreed, matematically and linguistically precise, integrated, and comprehensible like the concepts in physics.

APPENDIX 1: CLASS AS RECORD-TYPE FOR DYNAMICALLY-ALLOCATED OBJECTS

While there are thousands of OOPLs, from the memory-allocation perspective there are essentially two allocation strategies: static and dynamic. C# and Java require record-objects to be dynamically allocated. The term **reference type** is added for class, introducing more difficulty.

The term may seem to invalidate the concept of record-value, as can be seen in [27]. But here we prove that the dynamic allocation like in C# and Java does not invalidate the concept of record-value. The operational-semantic of

```
an_object := type_name (arguments_list) ;
```

is

```
for all columns in an_object
an_object.columns[i] := columns[i]
(type_name (actual_operands_list))
```

Mathematically we write:

$\forall$ $Columns_{[i]} \in Columns$
$An\_object.Columns_{[i]} := type\_id\ (arg_1\ [,\ arg_k])_{[i]}$

That the record-object (An_object) is a dynamically-allocated object does not change the fact that the value of arg_i is assigned to Columns[i]. Thus, the concept of class in C# and Java is correctly referred to as record-type.

APPENDIX 2: CLASS AS MODULE

In C#, Java, and similar programming-languages, class is mapped not only to record-type but also a module. Thus, M is a name for record-type and module-object.

NUSA helps understanding class as module. Fig 30 shows the translated source-code in NUSA.

```
Module M;

interface

type M := Record { };
M M ();
char object2 := 'a';
void operation2();

implementation

M M ()
{
  M this;
  return (this);
}

integer object1 := 2;

void operation2()
{
  Object2 := 3;
}

void operation1()
{
  Object1 := 'b';
}
```

Fig 30 Equivalent source-code in NUSA

Classes in C# and Java are both record-types and modules. This still confirms the theory that the common denominator for class is: record-type. Classes in other OOPLs (notably C++) are not modules.

APPENDIX 3: MODELING CLASS AS DERIVED CONCEPTS

This appendix explains the similarity of physics' base and derived dimensions with the proposed basic and derived concepts. Table VIIII shows two base dimensions and two derived dimensions in

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

physics. All derived dimensions must be based on base dimension(s).

Table VIII. Partial list of base dimensions and derived dimensions in physics; for various classic engineering

Base dimension	Derived dimension
Length (L) Time (T)	Area (L^2) Speed (L^1T^{-1})

Table IX tabulates class as derived concept, not a basic one; based on the explanation in Appendix 1 and Appendix 2. Class in C++, Delphi, and alike belong to the class as T^1 (merely as record type). Class in C#, Java, and alike belong to the class as T^1 Ob^1; a class is a type as well as an object (of type module).

Table IX. Partial list of basic concept and derived concept for software engineering

Basic concept	Derived concept
Type (T) Object (Ob)	Class (T^1) Class (T^1Ob^1)

ACKNOWLEDGMENT

The first author thanks The Ministry of Communication and Information of Indonesia for financial support in the making of NUSA code-translator.

REFERENCES:

[1] Saeed Moaveni. *Engineering Fundamentals: An Introduction to Engineering*, 2nd ed, Thomson Engineering. 2005.
[2] ISO. ISO/IEC 2382-15:1999 *Information technology -- Vocabulary -- Part 15: Programming languages*. ISO. 1999.
[3] Elisa Bertino, Lorenzo Martino. *Object-Oriented Database Systems: Concepts and Architectures*. Addison Wesley, 1993.
[4] Grady Booch et al. *Object-Oriented Analysis and Design*. 3rd edition. Addison Wesley, 2007.
[5] OMG. *OMG Unified Modeling Language Infrastructure*. Object Management Group. 2011.
[6] OMG. *OMG Unified Modeling Language Superstructure*. Object Management Group. 2011.
[7] ISO. *ISO/IEC 14882:2003 Programming Languages -- C++*. ISO. 2003.
[8] Twan Basten, Wil M. P. van der Aalst. "*Inheritance of behavior*", in *The Journal of Logic and Algebraic Programming*. Elsevier. 2001.
[9] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley. *The Java Language Specification*. Oracle Corp. 2012.
[10] Martin Sulzmann, Meng Wang. "*Modular Generic Programming with Extensible Superclasses*", in *ACM SIGPLAN*, September 16. 2006.
[11] David Greenfieldboyce and Jeffrey S. Foster. "*Type Qualifier Inference for Java*", in *OOPSLA October 2007* pp 21-25. ACM 978-1-59593-786-5/07/0010. 2007.
[12] Martin Plümicke. "*Typeless programming in Java 5.0 with wildcards*", in *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM pp 73-82. 2007.
[13] Norbert Schirmer. "*Analysing the Java Package/Access Concepts in Isabelle/HOL*", in *Concurrency and Computation: Practice and Experience*. 2003; 0:1-10. John Wiley & Sons. 2003.
[14] ECMA International. *ECMA-334 Standard. C# Language Specification*. 2006.
[15] Antero Taivalsaari. "*On the notion of Object*", in *Journal of Systems Software*; Vol 21:3-13 pp 3-16; 1993.
[16] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*, 2nd edition. Addison-Wesley, 2006.
[17] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE. 1990.
[18] Oxford. *Oxford American English Dictionary*. Oxford. 2005.
[19] Geoffrey Leech, Margaret Deuchar, Robert Hoogenraad. *English Grammar for Today*. MacMillan. 1982.
[20] Josë de Oliveira Guimaräes. "*The Green language type system*", in *Computer Languages, Systems & Structures*. Elsevier. 2009.
[21] Jaakko Järvi, Mat Marcus, Jacob N. Smith. "*Programming with C++ concepts*", in *Science of Computer Programming*. 2009.01. Elsevier. 2009.
[22] Adele Goldberg, David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley. 1983.
[23] C. J. Date, Hugh Darwen. *The Third Manifesto: Foundation for Object/Relational Databases*, 1998.
[24] Haibin Zhu, MengChu Zhou. *Methodology First and Language Second: A Way to Teach Object-Oriented Programming*, in *OOPSLA 2003, October 26-30.* ACM 1-58113-751-6/03/0010. 2003.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[25] Adele Goldberg, Alan Kay. *Smalltalk-72 Instruction Manual*. Xerox Palo Alto Research Center. 1972.
[26] Derek Rayside, Gerard T. Campbell. "*An Aristotelian Understanding of Object-Oriented Programming*", in *OOPSLA 2000, 10/00*. 2000 ACM ISBN 1-58113-200-x/00/0010. 2000.
[27] Nicu G. Fruja. "*Towards proving type safety of C#*", in *Computer Languages, Systems & Structures*. Elsevier. 2009.
[28] Frederick P. Brooks. *The Mythical Man Month*. Addison Wesley. 1975.
[29] Frederick P. Brooks. *The Mythical Man Month*, 2nd edition, Addison Wesley. 1995.
[30] Bernaridho I. Hutabarat. *Programming Concepts: with NUSA Programming-language*. Ma Chung Press. 2010.
[31] ISO. *ISO/IEC 9899:1999 Programming Languages --C*. ISO. 1999.
[32] Bernaridho I. Hutabarat. *Modular Programming: A Revolutionary Approach*. Ma Chung Press. 2010.
[33] Bernaridho I. Hutabarat, Mochamad Hariadi, Ketut E. Purnama, and Mauridhi H. Purnomo. "*Module, Modular Programming, and Module-based Encapsulation: Critiques and Solutions*"; in *The 5th International Conference on Information & Communication Technology and Systems* (*ICTS*). pp 233-240. ISSN 2085-1944. 2009.
[34] Bernaridho I. Hutabarat, Lucky Irawan. "*Simple and Universal Theory of Namespace and Its Relationship to Repetition using For(): How it affects the design of NUSA programming-language*"; in *Journal of Computer Science*, pp 167-176. ISSN 1412-9523. University of Pelita Harapan. 2011.
[35] Bernaridho I. Hutabarat, Mochamad Hariadi, Ketut E. Purnama, and Mauridhi H. Purnomo. "*NUSA (Neat Uniform Simple Architecture): A Highly Orthogonal Programming Language*", in *Proceedings of the World Congress on Engineering and Computer Science* 2011 Vol I WCECS 2011, October 19-21, 2011, San Francisco, USA. 2011.
[36] Hugh Darwen. The Third Manifesto, ACM SIGMOD Record, vol. 24, no. 1, pp. 39-49, March 1995.