40 Years of Super-Object-Oriented Programming

Eugene Kindler
Department of Informatics and Computers, Faculty of Science, Ostrava university
Street 30. dubna 22, CZ–701 03 Ostrava, Czech Republic
ekindler@centrum.cz

Abstract

The year of 2007 was a jubilee year, namely the anniversary of 40 years of the object-oriented programming (OOP). Nevertheless, when the first tool allowing application of the OOP paradigm was presented to the world professional community in 1967, it offered much more than the mentioned paradigm. The whole system of the offered tools was later called Super-Object-Oriented programming tools (SOOP). Its origin and its abilities have several intimate relations to computer simulation and to computing anticipatory systems. The properties of the OOP and of SOOP and their differing are presented in the paper and the relation to the computing anticipatory systems as well.
Keywords: object-oriented programming, anticipatory systems, SIMULA, simulation, super-object-oriented programming

1 Simulation and Computing Anticipatory Systems

When a human thinks on something that will operate in future, he/she is an anticipatory system. In such a situation, human often imagines the consequences of a present doing, which could appear in future. Human imagining is limited by its fuzziness and by a low capacity of abilities that check against possible inconsistencies made during the process of imagining. Computer simulation is an efficient tool for amplifying the imagining abilities of humans, because the computer storage ability allows evidence of many values figuring in the "imagined" process, the operation speed of computer allows taking millions of events into "imaginings", and the rational checking of consistency can be programmed by simple algorithmic techniques. When a human or a team applies computer simulation for getting information on the possible future consequences of the instantaneous decisions he/she/it is a real computing anticipatory system. Let us speak on ***anticipatory system of type S*** or shortly on ***S-system*** in such a case (*S* causes an association with *simulation*).

Note that the consequences can concern a large spectrum of the future, from that very near (e.g. those of pressing a certain button at a power plant) up to a very distant (e.g. those of releasing a given machine into production and then into operation). And note that computer simulation is not limited to inform on future, as it is applied e.g. to help determining parameters of some bygone processes (in medical or industrial diagnostics), but such applications are not interesting for the present paper.

Simulation enables to take many details into account and therefore the computer models used for it are often complicated, demanding a hard programming work. Application of conventional algorithmic technique of programming leads often to bad

events: an unexpected need of a rather small change of the simulated system comes, but its reflecting in the algorithm appears as its total rewriting.

The communities of simulation specialists solved those obstacles by inventing so called ***simulation languages***. The substance of their behold consists in the offer to their users: they do not need to algorithmize the simulation model, as they only need to described the simulated system itself and such a description is automatically either interpreted at running computer as real simulation model, or automatically translated into an algorithmic form, usually expressed very near to the proper machine language of the given computer. In the first case an *interpreter* and in the other case a *compiler* should be programmed; they are rather complicated and sophisticated program products, demanding more programming effort than a single simulation model without help of a simulation language, but they need to be programmed only once for a simulation language and that is their advantage.

Naturally, a simulation language can be applied only for description of a certain limited class Γ of systems (it is difficult to define a language in that one could describe e.g. systems of biological cells and production systems in machine industry, or a system of maritime transport). The conventional task of a component H of human civilization, which decides defining and implementing a simulation language λ has to anticipate the spectrum of possible applications and, according to it, to set bounds for Γ in a suitable manner: if λ were too particular some systems on that one expects to belong to Γ could not be described by λ in a suitable way, while in case λ were too general the description of all systems belonging to G would be obliged contain some components repeated for all of them. Therefore H appears an anticipatory system. Nevertheless, to propose a – may be excellent – simulation language without implementing it at computing technique (i.e. realization of its interpreter or its compiler) has no effect. H must be able to implement λ, and so H becomes a computing anticipatory system; let us call it ***anticipatory system of type L*** or shortly ***L-system*** (*L* like language).

2 Process Oriented Simulation Languages

To describe a system needs to present something that could be algorithmized (even by a complicated manner when an interpreter or complier runs, making millions operations with the description D of the given system). Soon after the first Von Neumann's digital computers have appeared (namely at the beginning of the 60ies of the XX. century) one has discovered that there is a lot of discrete event dynamic systems that are formed by a constant network of permanent elements and by a variable set of ***transactions*** that enter the system, move along the network, interact with the permanent elements and – through that interaction – interact also mutually. An example is a department store with fixed investment and employees, among which customers/transactions move; when a customer enters to cash desk, he may interact with another customer who has the same intention bus has to wait in the corresponding queue for some time. Another example is a biological cell system, viewed as having several sets of cells that are in the same state; each of the cells moves from one state-set into

another, dwells there some time and in certain states dies or multiplies. Another example is a transport system (railway network, highways, harbor, airport etc.).

A holistic description of such a system (e.g. by a conventional algorithm describing the time flow of the event occurring in it) is difficult but it is possible to view the system as composed of transactions so that the set of all possible transactions can be decomposed to a small number of sets called ***classes***, so that each of them contains transactions that are "similar". The similarity consists in the same formulation of ***life rules*** for every transaction of the given class, and in a certain list of data, the instance of which is handled by every transaction of the class. Thus, when any transaction of the class enters the described system it behaves according to the given life rules and handles its own instance of data. These data are called ***attributes*** of the transaction and respecting the life rules is metaphorically called ***life*** of the transaction. The life rules can be expressed by means of the components of conventional algorithmic languages (like assignments, branchings, cycles,...) completed by expressions with pseudorandom numbers and by so called ***scheduling statements*** that express that when the life of a transaction accesses such a statement it remains at it either some time, or until it gets some signal to continue, or until it recognizes that a certain condition is satisfied (according it, the scheduling statement are classified into several sorts).

It is known that a computing process following the algorithm can proceed in different manner, e.g. in dependence on instantaneous values of its variables. Similarly the lives of the transactions of the same class can differ, also in dependence on the values of attributes of other transactions.

The simulation languages that allow decomposing the description of a given system into transactions with attributes and life rules where both are grouped into classes are called ***process oriented (simulation) languages*** [1] and instead of term transaction one uses also ***process***. The first language called GPSS introduced in 1961 [3] was so primitive that some authors do not recommend to class it into the process oriented, but soon the languages of that sort were improved, e.g. by accepting all algorithmic tools of ALGOL 60 (a programming language recommended in the sixties as an international standard of algorithmization [4]) among the life rules. The culmination is the simulation language called SIMULA [5], [6] in the first half of the sixties and then SIMULA I, in order to distinguish this language from the new, object-oriented, SIMULA (see further). In the present paper, let that simulation language be called old SIMULA. For it the following important properties can be expresses:

(P1) a class has its name, its set of attributes and its life rules;

(P2) a class itself cannot influence the computing, but is a pattern for an arbitrary number of its ***instances***, i.e. elements, each of which has its own set of attributes and it own "life" according to life rules introduced in the class;

(P3) each of the members of the attribute set has its name and its type;

(P4) an instance can interact with other instances (of any class) by sending signals on the scheduling and by so called ***connection statements*** of form *inspect R do S* where *R* points to a certain instance of (an arbitrary) class and *S* is a statement; it is interpreted as belonging to the life rules of *R*, i.e. handling with attributes of *R*. (because of its form, the connection statements are often called ***inspections***).

3 Hoare's Record Handling

In 1966 a NATO advanced summer school on programming languages were organized in France [2] and O.-J. Dahl, one of the authors of the old SIMULA, was invited there to take a lecture on discrete event simulation languages [1]. Another lecturer at that school was C. A. R. Hoare who spoke there on record handling and introduced concept of record according to the following principles [7]:

(H1) Class (of similar records) has its name and a list of its components; each of them has a name and a type; the types can be those of conventional languages (numerical, textual,...) or ***pointers*** to other records. Any pointer has its ***qualification***, informing to which class the pointed record should belong.

(H2) Class is open for generating arbitrary number of its instances.

(H3) Class can be specialized by adding further components; so a new class arises, called ***subclass*** of the source class, which is called ***superclass*** or ***prefix*** of the subclass; if the prefix *C2* of class *C1* is also a subclass then the superclass *C3* of *C2* is also viewed as a superclass of *C1* and *C1* is viewed as a subclass of *C3*. The specialization can be iterated in an arbitrary number of steps and so a class *C1* can has its ***prefix sequence*** of classes beginning by *C1* and ending by a class *C* that has no prefix.

(H4) A class is open for any number of specializations.

(H5) Principles (H3) and (H4) cause possibility of trees of its classes ordered by means of prefix sequences.

(H6) An instance is generated by expression *new C* where *C* is a class, called ***native class*** of the given instance; such an instance is also an instance (but not a native instance) of prefix of *C*.

(H7) If *X* is a pointer qualified into class *C* and *Y* is a component introduced for class C, then *X.Y* represents the component *Y* of the record pointed by *X*.

(H8) If component *Y* of (H7) is also a pointer and if it is qualified into class *CC*, for which component *Z* is introduced, then *X.Y.Z* is a legal expression representing the component *Z* of the instance to that *X.Y* points.

(H9) If *X* is a pointer qualified (= with qualification) to *C* then *X* can point to any instance the native class is *C* or any subclass of *C*.

(H10) If *D* is a subclass of *C*, *Z* is a component of *D* that is not introduced for *C*, and *X* is a pointer qualified to *C*, than *X.Z* is erroneous (as it is in contradiction with the qualification of pointer *X* – note that *X* can point also to other classes than *D*).

The expression introduced in (H7) is called ***remote identifying*** and more popularly ***dot notation***; according to (H8), it can be iterated. Note the records are passive elements according to Hoare's conception; they had to be elaborated by statements organized in a conventionally structured algorithm. Rule (H10) is often called ***type discipline***

The old SIMULA had attributes, i.e. something like components of the records according to Hoare, it had life rules that were not considered in Hoare's contribution, but it had not dot notation, pointers and subclasses introduced by Hoare. In this situation, Dahl's idea arose to accept all that Hoare had carried in his lecture and the old SIMULA was enriched according to the following principles:

(PH1) class has its name, attribute set and life rules and can serve for generating as a pattern for any number of instances;

(PH2) the attributes of an instance are accessible for another instance in the same manner as it was introduced by Hoare and formulated in (H7) and (H8). The dot notation is not in contradiction with the connection statements (see (P4)) and can remain as another tool for expressing the interactions among the instances;

(PH3) a class can be specialized to any number of subclasses by adding further attributes and life rules; the terms subclass, superclass and prefix are used in the same manner as in (H3) and a subclass can serve as prefix for any further specialization.

4 Procedures as Components of Classes

Although Dahl often expressed his gratitude to Hoare and his ideas the synthesis just described in (PH1) and (PH2) is far from the OOP. The next step consists in that Dahl and his collaborator K. Nygaard included procedures into the class concept and allowed the dot notation for calling procedures. So (PH1) was enriched to

(SOOP1) class has its name, attribute set, procedure set and life rules and can serve for generating as a pattern for any number of instances;

while (PH2) was enriched to

(SOOP2) the attributes and procedures of an instance are accessible for another instance by the dot notation. In the connection statements of form *inspect R do S*, the procedure calls occurring in *S* are in the first level interpreted according their possible declaration for class to that *R* is qualified.

And the first part of (PH3) was enriched to

(SOOP3) a class can be specialized to any number of subclasses by adding further attributes, procedures and life rules.

The qualification and type discipline are generalized for procedures, too. For the next analysis, let us formulate other three principles, namely those arising from (SOOP1)-(SOOP3) by omitting life rules from them:

(OOP1) class has its name, attribute set, and procedure set and can serve for generating as a pattern for any number of instances;

(OOP2) be equal to (SOOP2);

(OOP3) a class can be specialized to any number of subclasses by adding further attributes and procedures.

OOPi should recall term Object-oriented programming, while SOOP should recall term Super-object-oriented programming. Evidently, if a language offers (SOOP1)-(SOOP3) it offers (OOP1)-(OOP3), too.

5 Up to the Object-Oriented Anticipation

In the middle of the 60ies of XX century the first symptoms of a so called software crisis appeared. One of the aspects of that crisis was a need of many new programming languages and their implementations, i.e. compilers and interpreters. That situation became especially evident in computer simulation where new domains of application ori-

ginated, i.e. new sets of similar systems demanded their proper simulation languages and their implementation. The desire arose to implement the new languages without dreadful programming of compilers or interpreters. The OOP appeared to be a good technique for it. The full definition of it is presented in the next section, but already now, i.e. with the knowledge of (OOP1)-(OOP3), it is possible to explain its role in the mentioned situation.

Let an expression like *X.F*, where *F* is a procedure, occurs in the life rules (or in a procedure declaration) in the declaration of class *C* and let *Y* be an instance of *C*; when *Y* meets this expression it models something like "*X*, be kind to perform a service for me, namely to perform the procedure *X*". Because of that interpretation, *X.F* is called a ***message*** addressed to *X*, while *X* is called ***addressee*** of this message. If instead of *F* a verb is applied for the given procedure, then the message may be interpreted as a command to the addressee. If the verb is especially well chosen, the contents of the procedure can be remembered and the procedure may be accepted as representing a new sort of statement, moreover – as a new sort of a statement, which can be accepted by the computer without programming a new compiler/interpreter.

Such a procedure may be introduced to have one or more parameters and then the form of the message *X.F(Y)*. In case of a suitable choice of name *F*, such a message can recall a command with verb *F* and object *Y* (e.g. *X.accept(Y)* or *X.draw(Y)*) or a real military command where *F* is something like preposition or conjuction (e.g. *X.into(Y)*). If such a procedure is a function, i.e. if it has a result, the construction like *X.F(Y)* may represent a mathematical operation (like *X.plus(Y)* where *X* and *Y* may be vectors, matrices, texts etc.) or a grammar clause that can figure as object or subject in the command mentioned above (e.g. *X.plus(Y).assign_for(equation1.left_hand_side)*).

If one has a programming tool for disposal, which satisfies (OOP1)-(OOP3), a possibility offers to him to analyze a certain domain of interest (a – may be infinite – set *D* of systems) and to anticipate the statements that will be useful for exact description of such systems. He usually starts by analyzing the language common among the specialist in the domain for exchange the information about the systems belonging to *D*, and analyzes what of its language structures will be suitable as tools for computer models of the systems occurring in *D*. Some authors call that activity ***domain analysis*** [8]. In fast, he anticipates what (simulation) programming language will be necessary in the future, and so he becomes an anticipatory system. And when he formulates declarations of considered classes, debugs them at a computer and tests them at some possible future models, he really becomes a computing anticipatory system that we will call ***anticipatory system of type OOP*** or simply ***OOP-system*** (that should be an association with the object-oriented programming). He makes the same work as if he anticipates on one or more anticipatory systems of type L. Naturally, in place of the mentioned individual person a team can exist.

6 With Virtuality up to the Object-Oriented Programming

More classes can contain procedures with the same name, i.e. such procedures are homonymous. Procedures with the same names can be introduced even for more

members of a prefix sequence. When an instance *A* sends a message *B.F* to addressee *B* (no matter whether with or without parameters) and *F* belongs among homonymous one, then – according to the type discipline – *A* determines unambiguously what *F* represents (in fact, the meaning of *F* is determined by the qualification of *B*). When Dahl and Nygaard thought over the block orientation, they came to the following conclusion.

Suppose *C* is a class and *C1*, …, *Cn* classes of the tree of subclasses of *C*. It is possible to declare a procedure *F* introduced in *C* as ***virtual*** so that its contents can be re-declared (in different ways) in arbitrary classes of *C1*, …, *Cn*. It may be re-declared by different ways for two classes even in case one of them is a subclass of the other one. When the name of the virtual procedure occurs in a message like *B.F* then the contents of *F* is not chosen according to the qualification of *B* (as that would be in case *F* is not specified as virtual) but according to the native class of *B* (see principle (H6) in section 3). A difference between calling virtual and non-virtual procedures is illustrated in Fig. 1, where the circles represent classes and arrows specialization. Assume a circle with digit *k* represents class with name *Ck*. The circles in full line represent classes with declaration of procedure called *F*, the circles in dashed line represent the classes without a declaration of *F*. Suppose that *B* points to instance *K*, is qualified into *C1* and used in the message *B.F*; if *F* is not specified as virtual this message is always performed according to the declaration formulated for *C1*, independently of the native class of *K* Let *N* be the native class of *K* and *F* be specified as virtual in class *C1*. Then the message *B.F* is performed according to the declaration occurring in *C1* if *N* is *C1*, *C2* or *C3*, according to the declaration occurring in *C4* if *N* is *C4* or *C5*, according to declaration occurring in *C6* if *N* is *C6* and according to the declaration occurring in *C7* if *N* is *C7* or *C8*.

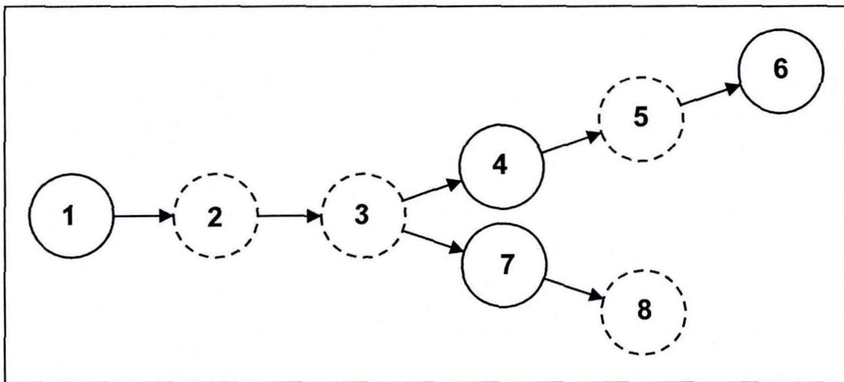

Figure 1: Arrows lead from classes to their subclasses

Dahl and Nygaard came to the importance of virtual procedures after profound analyzing the consequences of the block structure (see section 8). Nevertheless, the behavior of virtual procedures is very impressive for the users, because it is a model of a certain initiative behavior: in case a message contains a virtual procedure *F*, then its

addressee does not interpret F according to the qualification of its own, but – initiatively and independently of the sender of the message - it determines itself the rules for the answer to the message (according to its own native class). The popularity of virtual procedures became so high that the authors of some later OOP tools (e.g. SmallTalk, Eifel) stated that every procedure has to be taken as virtual. On the other side, saving non-virtual procedures and letting the user to be free in determining what procedures he wants to have as virtual, has value and so not only the new SIMULA but also e.g. C++ allow the choice.

Nowadays, a possibility to use virtuality of procedures is understood as an integral part of OOP. Virtuality without subclasses and procedures as components of class instances is meaningless. Interesting is that the life rules (including scheduling statements) were not accepted as constituent part of OOP – such popular OOP tools as C++, SmallTalk 80 and later versions of Pascal do not support it. In other words, the OOP can be characterized by principles (OOP1)-(OOP3) formulated in section 4, and the following principle of virtuality:

(OOP4) procedures can be virtual.

The lack of life rules makes the related languages detrimental for making simulation models in a similar way like by using process oriented languages.

By assuming life rules and scheduling statements, the development of SIMULA exceeded the OOP many years before the world professional community accepted it as the best paradigm of programming. The influence of simulation to attend that matter is evident. But the life rules were not all that the development of SIMULA carried in 1967. The other contributions are explained in the next three sections.

7 Life Rule Sequencing by Virtuality and by Sequencing Statements

What was presented under title SIMULA as its new version was soon called SIMULA 67 in order to distinguish it from the old SIMULA. All essential ideas of SIMULA 67 [9] were presented at the IFIP Working Conference on Simulation Programming languages held is Oslo in May 1967. In the next months, the proposal was refined in certain details (e.g. in manipulation with files) and published as [10]. That definition remained constant until 1986 when the language was declared as an international standard under ISO. At this occasion the language title was returned to SIMULA (without affix 67, because the old SIMULA was completely forgotten and its former users went over to SIMULA 67) and the language itself was completed by standard procedures corresponding to the use of personal computers and workstations [11].

The life rules offered two complements to SIMULA 67, which would be meaningless for OOP tools that have no life rules:

Already the old SIMULA took from ALGOL 60 possibility to transfer the life inside the sequence of life rules. The new version of SIMULA offered a possibility to introduce ***virtual labels***, according to the following principle:

(SOOP5) among the life rules introduced for a class, a transfer to a virtual target can occur; when an instance of a subclass of the class performs the mentioned life rules it

looks for the target among the life rules of the subclass; so the virtual labels enable transfer of the life among life rules of different classes.

A general scheme of the realization of a scheduling statement is, that the process P that interprets the own life as a phase of the model run (i.e. is performing its own life rules) interrupts it at the scheduling statement, switches the computing run to the life of another process and into its own evidence saves the place that immediately follows the scheduling statement; the place is called ***reactivation point***; later, when the computing should return to perform the life of P, it starts from the reactivation point.

This scheme can be generalized to so called ***sequencing statements*** so that the scheduling statements can be introduced as standard procedures defined with use of the sequencing statements. SIMULA 67 was equipped by the following three sorts of sequencing statements:

(SOOP6) *call(X)*: when the life of Y meet such a statement its performing is interrupted, the reactivation point is assigned by the place immediately following this statement and the computing process is switched to the reactivation point of X; X includes into its evidence Y as its "caller" and enters into *attached* state;

(SOOP7) *resume(X)*: it behaves similarly as *call(X)* but nothing like a "caller" is put into evidence; X enters into *resumed* state and forgets what entity stimulated it to continue its life;

(SOOP8) *detach*: when X performs its life and is in attached state, *detach* returns the control of computing of X to the caller of X; the life of the caller continues from the reactivation point, while the reactivation point of X is set immediately after the detach statement; explaining the effect of detach performed by an element that is in resumed state is behind the scope of this paper, as it needs to know much on the so called quasi-parallel systems, i.e. systems of elements that switch their lives by iterating *resume*.

When one makes some software that does not concern simulation, he does not need using the scheduling statements offered by SIMULA. So SIMULA left a position of a language determined only for simulation, and become general purpose programming language with excellent tools to define special purpose programming languages without exigence to implement their compilers or interpreters. The sequencing statements represent the last reason against any opinion that SIMULA might be limited to simulation.

8 Block Nesting and its Consequences

Already the old SIMULA perfectly followed the block structure designed for ALGOL 60 [4] and SIMULA 67 as well. The ***textual block*** is a part of program for which ***local entities*** are introduced; let π be a computing run according to a program p in that a textual block b occurs. When π enters into the place corresponding to the beginning of b, a ***block instance*** β corresponding to b arises so that the local entities introduced in b exist for β and can be used by it. When π passes the place corresponding to the end of b over, the block instance β disappears and the local entities introduced in b as well. Any textual block b is in principle like any other statement (or life rule) and

contains statements that are performed when π is in the corresponding block instance β. Among these statements (life rules), another textual block *bb* can occur and in such a case a block instance $\beta\beta$ corresponding *b* arises when β enters the place corresponding to the beginning of *bb*. So ***subblocks*** and nesting of blocks can occur, which causes a contemporary existing of more block instances.

It is also possible that when π is in β (i.e. before leaving β), a new entrance of *p* into the same textual block *b* is met; in such a case two (or, in general, more) instances of the same textual block exist at the same moment. The theoreticians of programming like to illustrate that possibility by recursive calling of procedures, but simulation (or object-oriented) programming languages that allow life rules and scheduling or sequencing statements offer more natural examples: assume a class *C* has a textual block *b* in its life rules and that textual block contains a scheduling statement; such a statement can cause an interruption of the life of an instance *A* of *C* for a certain phase *f*; but during phase, another instance *B* if *C* can perform its life rules, can also enter *b* and be in phase like *f*. In other words, each of the instances *A* and *B* can occur in its own instance β of the same textual block *b*.

In case of ALGOL 60, the entities local in blocks can be namely variables and procedures. Suppose inside life rules of class *C* a textual block *b* occurs and variable *x* and procedure *g* are introduced as local in it; when the life of an instance *A* of *C* enters *b* it models a situation that *A* becomes be able to "know what means *x* and what should make *g*". When the lives of two instances *A* and *B* of *C* enter *b* then they are modeled like each of them has its own mentioned knowledge that may have different contents and meanings for *A* and for *B*, namely in case it depends on the attributes and (virtual and non-virtual) of the corresponding instance.

The authors of SIMULA 67 formulated a very important decision that could be formulated as

(LC) to introduce somewhere a class is like to introduce somewhere a procedure

and, according it, they designated the standpoint that

(SOOP9) a class can be introduced as local in a block.

A class is a computer image of a concept (see [12]). Therefore, when the life of an instance *A* of a class *C* enters a block in that a class *D* is introduced as local, *A* is modeled as becoming an "expert" who understands the concept mapped by *D*. When the life of *A* leaves that block, *A* is modeled like forgetting what is *D*. In other words, when the life of the instance is being in that block it may model a "professional" phase of the instance. When the lives of two instances of class *C* are in the mentioned block they may be viewed as models of two experts that can communicate about the same abstract concept but each of the instance has its own image of the concept in his mind and – relating to the other entities owned by the instances, the images of the minds may more or less differ.

Let *C* be a class and *A* its instance. It is possible that among the life rules of this class a block *b* occurs in that a local class with the same name *C* is introduced. Both the classes can be formulated in a more or less different way. If the difference is small then block *b* represents a phase that the *A* can be in a certain sense an image of an entity with reflection of its own or of several entities. If the difference is rather great then the class

introduced in block *b* is an image of a certain name conflict that causes no contradiction (e.g. *weight* in sense of physics and in sense of statistical computation).

Note that the common understanding to the concept of OOP takes as sufficient if classes can be declared at only one level, namely as global entities for the whole program or program module (Pascal, C++, SmallTalk, Eifel, Modsim,...).

9 Main Classes and Class Nesting

Already in ALGOL 60 and similarly in SIMULA 67, textual block was introduced as a part of the source code, bordered by certain parentheses and containing "declarations" of the local entities, followed by statements. The declaration of a class was introduced as a part of source code beginning by a certain heading containing the information on the name, prefix, virtual entities and parameters of the class, followed by class body, the general form of which was like that of textual block, only the interpretation was slightly different:

(1) in class body, the declarations of attributes corresponds to the declarations of local variables in textual block,

(2) class body can contain declarations of procedures similarly as textual block,

(3) in class body, the life rules correspond to the statements in a textual block.

And further analogies exist: textual block is a "pattern" for block instances similarly like class declaration is a "patter" for class instances, and a possibility of a greater number of contemporaneous instances of the same class exists analogously to more contemporaneous instances of the same block. The differences consist in the possibility of giving names to the class instances (for the blocks it is prohibited in SIMULA 67), and in generating the instances: a block instance can arise when the computing process enters in the corresponding, exactly given place, while an instance of class *C* can arise whenever the class generator (*new C*) is applied.

A simple synthesis of the analogy expressed by (1)-(3) with the principle (LC) expressed in the preceding section leads to another offer of SIMULA 67:

(SOOP10) A declaration of a class *C* can contain a declaration of another class *D*.

In such a situation, *C* is called ***main class*** and *D* ***nested*** class (or class nested in *C*). *C* is a model of a certain formal theory (or world viewing, formal language,...) while D is a model of a concept used in this formal theory (or of in this world viewing, or a term in this formal language). In a main class *C*, classes *D* and *E* may be nested so that they are independent or one of them can be a subclass of the other; but none of them can be subclass of *C*. The main classes can be used similarly as other classes, among other they can be specialized. If *D* is nested in *C* and *C* is used for formulating its subclass *E*, then *D* is automatically available in *E*; therefore also subclasses of *D* can be introduced into *E*. If a declaration of a class with the same name *D* occurs is *E* it represents another concept that the declaration of *D* accessible in *C*.

Of course, the main class usually contains more nested classes in practice. Principle (SOOP10) offers a possibility to anticipate the whole theories (based upon more than one concept – may be results of analysis of realistically understood domains), i.e. the whole formal languages manipulating more than one words with different meaning.

They could be applied by using a simple prefix, concretely name of the suitable main class. It is allowed by the following principle of SIMULA 67:

(SOOP11) If *C* is a class and *B* a block or a sequence of statements bordered by parentheses used for blocks, *C* may be put before and so called ***prefixed block*** arises that behaves like an instance of a fictitious subclass *F* of *C*, for which the statements of *B* represent the life rules and the possible declarations occurring in *B* represent those of attributes, procedures and classes.

In practice, *C* is a main class representing a formal language *L* and a block prefixed by *C* can be programmed with use *L* (and – naturally – with use of all tools offered by SIMULA 67). It is possible that a prefixed block is nested in another block.

Note a nested class can be a main class in that another nested class is nested. In other words, the class nesting can be iterated. As an example an instance *K* of a class *C* that is nested in a main class *M* and that contains a main class *m*; when *K* enters in its life to use *m* it behaves like an image of an element that models a certain situation using *m*. It is illustrated in Fig 2, where

the "scene" in double-line represents a reflection of a world viewing by means of *M*,
the circles in double-line correspond to objects identified in that world viewing,
κ represents one of such objects, namely *K*,
the bow inside κ with an arrow in the center represents the life of *K*,
the "scene" in dashed line, which hangs down from the bow, represents a model μ of a certain part of the world, based on world viewing *m* handled by *K* during its life, and
the circles in dashed line represent the components of μ.

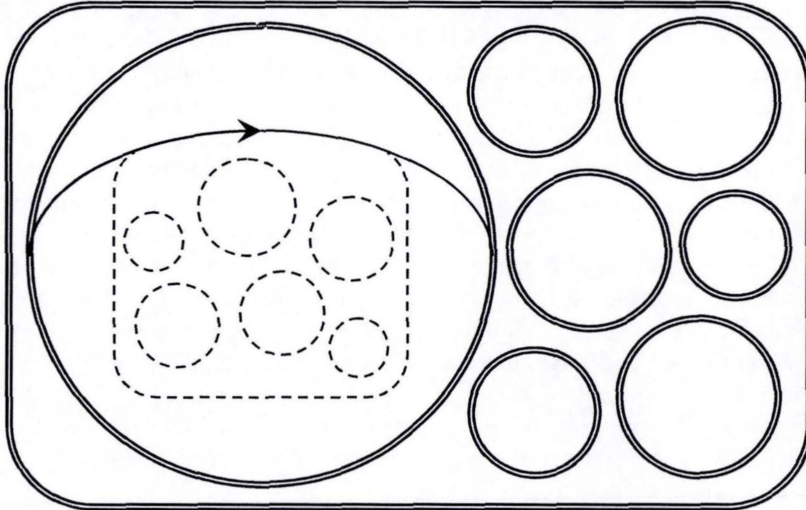

Figure 2: A model of a system in phase of reflection

Such a multiple class nesting occurs namely in simulating anticipatory systems, i.e. in simulating systems that contain elements that simulate or imagine and according to that activities they decide of their own instantaneous steps (see e.g. [13]-[15]).

The active users of OOP paradigm that do not pass beyond to the SOOP often prepare several classes; use terms “class package”, “class library” or “programming module composed of classes”. Note that such terms are related more to the computers that to the possible reality that should be reflected in the corresponding computer modeling. A modeler (or a team of modelers) who formulates and implements a main class with a target to be applied for computer models

(α) can be free from the ”computing machines sociology and viscera” and completely concerned with the modeled reality, and

(β) can formulate the main class so that it contains more that nested classes – also a main class can have its procedures, attributes and life rules.

For illustrating (β), an almost copybook example cane be presented. The main class *G* reflecting plane geometry need not to be a mere collection of classes of points, lines, squares, circles and many other representations of geometrical concepts; the names of the co-ordinate system center and axes can be introduced as attributes of *G* and in the life rules of *G* these names can be “filled in” by an instance of point and by two instances of lines. In such a state, when a user of *G* wishes to apply it as a bases for a certain more special world viewing (e.g. for viewing at a map of a given array), he can count with the existing co-ordinate system and enrich it to the static configuration of objects of the reflected array by further attributes and life rules. Similar praxis is admitted in case of procedures of *G* – e.g. one knows that a line *q* can be introduced as a join of two points *a* and *b*; that custom can be mapped in *G* as its procedure (namely function that gives a result), which can be called as e.g. *join(a,new point(3,4))* in SIMULA can be then used e.g. in a condition *if joint(a,new points(3,4)).contains(O)*, which uses a procedure contains *introduced* for the class of the lines and which represents a test whether the given line comes through the co-ordinate center.

So the concept of main class leads to the further stage of anticipatory systems: an author of a main class anticipates and formalizes not only isolated concepts but their logical systems that assembly the concepts to a given formal language and even to a given scene (or basic structure of such a scene). The authors of such main classes anticipate that one will use such a formal language not only for describing computer models but for defining more specialized formal languages, “tailored” to less fuzzy horizons (e.g. the tailoring the language on plane geometry, mentioned above, to languages of geographical maps, then further to a given sort of maps etc., but on the other side e.g. for the language of domestic architecture, of electronic circuits etc.). The authors of main classes could be called ***anticipatory systems of type SOOP*** or ***SOOP-systems***.

10 Conclusion

Although the principles (SOOP1)-(SOOP11) were formulated 40 years ago, only a small part of them, namely (OOP1)-(OOP4) were projected in the OOP paradigm and only a small number of programming languages followed more; e.g. MODSIM [16] took over life rules and scheduling statements, C-FLAVOURS]17] took over life rules and sequencing statements. May be Java and especially BETA [18] allow life rules,

sequencing statements and class nesting – at the present time, the real possibilities of Java are profoundly tested at Ostrava University). Note that the next OOP languages that followed SIMULA 67 were implemented not sooner than in the eighties of the XX. century and the languages mentioned above, which more or less passed over OOP in direction to SOOP, arose even ten years later.

The humans and human social communities are anticipatory systems that reflect themselves, i.e. that anticipate also their anticipation, their abilities of anticipation etc. The application of computer models has to include the mentioned abilities. The first experiments have been done and classified, the SOOP programming paradigm promises a good way to implement the computer models without essential obstacles, but there is a wide horizon for developing further works, to go beyond the four stages of S-systems, L-systems, OOP-systems and .SOOP-systems.

Acknowledgement

This work has been supported by the Grant Agency of Czech Republic, grant reference no. 201/060612, name "Computer Simulation of Anticipatory Systems".

References

1. Dahl, O.-J. (1966). Discrete Event Simulation Languages. Norwegian Computing Center, Oslo. Reprinted in [2], pp. 349-394.
2. Genuys, F., ed. (1968). Programming Languages. Academic Press.
3. Gordon, G. (1961) A General Purpose Systems Simulation Program. Proc. 1961 EJCC, Published by MacMillan, New York, pp. 81-91.
4. Backus, J. W. et al. (1960) Report on the Algorithmic Language ALGOL 60. Numerische Mathematik, 2, pp. 106-136.
5. Dahl, O.-J. and K. Nygaard (1965). SIMULA – a Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual. Norsk Regnesetralen, Oslo.
6. Dahl, O.-J. and K. Nygaard (1966) SIMULA – an ALGOL-based Simulation Language. Communications of the ACM, 9, no. 9, pp. 671-678.
7. Hoare, C. A. R.(1968) Record Handling. Programming Languages. Edited by F. Genuys, Published by Academic Press, pp. 291-346.
8. Hill, D. R. C. (1996). Object-oriented Analysis and Simulation. Addison-Wesley.
9. Dahl, O.-J. and K. Nygaard (1968) Class and Subclass Declarations. Simulation Programming Languages – Proceedings of the IFIP Working Conference on Simulation Programming Languages. Edited by J. N. Buxton, Published by North-Holland, pp. 158-174.
10. Dahl O.-J., B. Myhrhaug and K. Nygaard (1968). Common Base Language. Norwegian Computing Center, Oslo. 2nd edition 1972, 3rd edition 1982, 4th edition 1984.
11. SIMULA Standard. (1986). Simula a.s., Oslo.
12. Dahl, O.-J. (1970) Programming Languages as Tools for the Formulation of Concepts. (eds.). Proceedings of the 15th Scandinavian Congress, Oslo 1968. Edited

by K. E. Aubert and W. Ljunggren, Publisher by Springer, Lecture Notes in Mathematics no 118, pp. 18-29.
13. Kindler, E. (2001) Computer Models of Systems Containing Simulating Elements. Computing Anticipatory Systems: CASYS 2000 – Fourth International Conference. Edited by Daniel M. Dubois, Published by The American Institute of Physics, AIP Conference Proceedings 573, pp. 390-399.
14. Berruet, P., Coudert, T. and Kindler, E. (2004) Conveyors With Rollers as Anticipatory Systems – Their Simulation Models. Computing Anticipatory Systems: CASYS 2003 – Sixth International Conference. Edited by Daniel M. Dubois, Published by The American Institute of Physics, AIP Conference Proceedings 718, pp. 582-592.
15. Krivy, I. and Kindler, E. (2006) Synthesis of two Anticipatory Modes in Design and Life-Cycle of Hospitals. International Journal of Computing Anticipatory Systems, Vol. 18, pp. 75-85.
16. Herring, C. (1990) ModSim: A new object-oriented simulation language. SCS Multiconference on Object-Oriented Simulation. Edited by A. Guasch, Published by The Society for Computer Simulation, San Diego.
17. Kreutzer, W. and Stairmand, M. (1990) C-Flavours – a Scheme-based Flavour System with Coroutines and its Application to the Design of Object-Oriented Simulation Software. International Journal of Computer Languages Vol. 15, No. 4, 225-249.
18. Madsen, O. L., Møller-Pedersen, B. and Nygaard, K. (1993). Object-Oriented Programming in the Beta Programming Language. Addison Wesley.