Science of Computer Programming 78 (2013) 1407–1426

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

When intuition and logic clash: The case of the object-oriented paradigm

Irit Hadar [*]

Department of Information Systems, University of Haifa, Carmel Mountain 31905, Haifa, Israel

ARTICLE INFO

Article history:
Received 11 December 2011
Received in revised form 18 July 2012
Accepted 1 October 2012
Available online 12 October 2012

Keywords:
Object-oriented analysis and design
Cognitive processes
Intuition
Qualitative research

ABSTRACT

This paper presents difficulties illustrated by software development practitioners with regard to the use and understanding of concepts and principles of the object-oriented paradigm. In order to understand these difficulties and the sources from which they stem, a theoretical framework borrowed from cognitive psychology, called the dual-process theory, was used. The empirical findings, collected using tools based on the qualitative research approach and analyzed through the lens of the dual-process theory, suggest that a tension between intuitive and logical thinking modes may lead to simple mistakes software engineers exhibit when practicing object-oriented analysis and design (OOA/D). Examining the understanding of OOA/D from this perspective helps explain some of the previously documented, as well as some newly identified difficulties in learning and practicing OOA/D, as well as guide the design of future quantitative experiments to understand how prevalent these phenomena are.

1. Introduction

The object-oriented (OO) programming paradigm was created partly to help software developers cope with the ever-increasing complexity of software systems. Although this paradigm is practiced extensively in industry and emphasized in academic education, studies have repeatedly shown that object-oriented programming, analysis and design are quite difficult to learn and practice [1–6].

Consider the following dialog observed during this research in an advanced object-oriented design workshop for software engineers from industry:

Instructor: In an ambulance support system we have a class called *Doctor* and a class called *Paramedic*. The *Doctor* is authorized to do everything the *Paramedic* does, as well as some additional activities, such as declare a patient's death. Consider a case where only these two classes exist and we wish to use an inheritance relation between them. Which class should inherit from the other?
Participant: The *Paramedic* must inherit from the *Doctor*, because the *Paramedic* only has a subset of the *Doctor*'s functions.
Instructor: So who has more functionality?
Participant: Oh, sorry — *Doctor* should inherit from *Paramedic*.

Similar conversations were observed many times during this research in different contexts, all illustrating the same phenomenon of confusing the inheritance direction (see another example in [7]). This phenomenon is one of several identified difficulties that involve basic concepts and principles of the OO paradigm. In fact, professional software developers were observed having difficulties with the most basic concepts and principles even at the preliminary, exploratory phases of this study. These observations led to the direction taken in this research to investigate and understand the type of difficulties professional OO developers encounter and their underlying causes.

[*] Tel.: +972 48288508; fax: +972 48288522.
E-mail address: hadari@is.haifa.ac.il.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.10.006

In search for an explanation, the psychological literature revealed a framework that seemed to shed some light onto these surprising difficulties experienced developers exhibited when dealing with basic concepts of the OO paradigm. This framework, called the dual-process theory (e.g. [8]), suggests that two separate cognitive mechanisms operate within our minds, roughly corresponding to what is commonly known as intuitive and analytical thinking. The dual-process theory explains simple mistakes intelligent people make in daily tasks even when they have the knowledge and skills needed to avoid them. According to this theory, the source of such mistakes is the clash between these two thought mechanisms. Leron [9,10] and Leron and Hazzan [11] show how the dual-process theory helps explain mistakes made when solving problems in mathematics. Analysis of the data collected in this research in light of the dual-process theory shows that experienced and capable software engineers make mistakes or misjudgments even when they have the knowledge needed to make better decisions. Examining the understanding of the OO concepts and principles from this perspective helps explain some of the previously documented difficulties in learning and practicing OO development, as well as some that were newly identified in this research, specifically in cases where developers have the knowledge required to overcome these difficulties. Understanding the sources of these difficulties can be helpful for improving education as well as state-of-the-art programming paradigms.

The main objective of this research is to explore how software engineers understand and realize concepts and principles of the OO paradigm when performing OO analysis and design (OOA/D). Thus, the following research questions were formulated:

(1) What difficulties and challenges do software engineers face when practicing OOA/D?
(2) What can be identified as possible sources of these difficulties?

Based on the findings of the initial exploratory study, the research focused on relatively basic concepts (e.g. object, class, class relations) and principles (e.g. abstraction and encapsulation) of the object-oriented paradigm.

The OO concepts and principles are analyzed here regardless of whether they are associated with OO analysis, OO design or both.[1] The research subjects were observed working on design tasks based on textual requirements rather than on OO analysis models, thus performing both OO analysis and design to complete their task, with no apparent distinction between the two activities. As this research focuses on the cognitive mechanisms when making decisions regarding basic concepts and principles of the OO paradigm in general, the distinction between analysis and design activities is not needed, and hence they will all be referred to as OOA/D activities.

The next section of the paper presents a literature review of research on cognitive processes and difficulties in object-oriented development. In Section 3, the research method and settings are presented, followed in Section 4 by a presentation of the dual-process theory, which was the theoretical framework that was found to shed light on the findings of this research. The finding are presented and discussed in Section 5 and, finally, the main conclusions of this paper are summarized and presented along with suggestions for future research in Section 6.

Section 5 that reports the subjects' quotations is admittedly heavy reading. However, showing these in full is necessary to show how the conclusions are supported. The author believes that the results are worth the reader's patience.

2. Cognitive processes and difficulties in object-oriented development

Researchers began studying the cognitive processes associated with programming and software development in general as early as the 1970s [13–16]. In the 1980s, the focus shifted to the more specific context of software design [17–20] and, from the 1990s, to OO development [5,6,21–25].

The OO paradigm aims to help handle the high complexity of software systems and reduce the cognitive effort and load required from the developers and maintainers of software systems. The idea is to exploit the human mind's natural capabilities for thinking about the world in terms of objects and classes, thus recruiting our intuitive, natural thinking in order to build formal software systems.

Researchers have commonly assumed that the intuitive and formal systems of objects and classes are similar and that fluency in the former helps deal efficiently with the latter. Moreover, some have claimed that OO naturalism stems from the fact that system constructs are abstract versions of items in the real world, i.e. they do the same things to solve their problems in the cyber-world that their corresponding items do to solve the same problem in the real world [26]. Although knowing how a problem is solved in the real world helps us understand how the OO system would solve the problem [21,27,28], studies have shown that object-oriented programming, analysis and design are quite difficult to learn and practice [3,4,6,29,30], and yet other researchers claim that object technologists have underestimated the complexities associated with objects [3,31]. It seems that naturalness does not necessarily mean easy to understand and learn [26].

Over the past three decades, understanding the cognitive activities of software developers has evolved as an area of interest and research. Models describing the cognitive mechanisms of software development and maintenance have been

[1] OO analysis is "a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the domain" and OO design is: "a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design" [12, p. 36].

proposed over the years [15,17,21,32]. Cognitive models and difficulties associated with the OO paradigm, and with several of its basic concepts and principles, are presented next.

2.1. Object/class

One of the claims for the naturalness of OO is that mapping between the problem domain and the programming domain should be more straightforward in the OO paradigm than in the procedural paradigm [28]. The theoretical argument in support of the OO approach is that objects are clear and visible entities in the problem domain and are represented as explicit entities in the solution domain as well, so that the mapping between the problem and solution domains is simple and clear. According to Meyer [33], the world can be naturally structured in terms of objects. Thus, it seems particularly relevant to organize a model of design around a software representation of these objects. The OO paradigm facilitates object-based problem decomposition as well as object-based design solutions [31].

Based on the above, it seems that a basic implication of the OO approach to design is that designers may give more attention to problem understanding. Focusing on objects, the designers begin by organizing problem representations in terms of the objects that participate in problem activities rather than in terms of general functional requirements or constraints. This requires careful analysis of the semantic structure of the problem [21]. As well, the OO approach enhances the focus of problems by supporting a more sustained process of problem understanding. "[M]uch of the design process itself consists of dealing with objects in terms of their abstracted interfaces and behaviors, not with the implementations of those interfaces and behaviors. As a result, the decomposition process can be quite conceptual in nature as problem entities are identified, organized, and specialized to meet problem requirements". ([21] p. 362).

However, not all researchers agree: Baragry [34] for example claims that the belief that objects allow us to directly implement our views of reality is no more than one of the major myths of object-orientation. One reason for that is that people's conception of reality cannot be considered separately from the objective reality. As people interact with the world, they automatically and subconsciously apply their accrued concepts and theories to the observed phenomena in order to understand them. Consequently, people's explanatory theories do not capture the natural order of the world; rather, they are subjective to the person applying them.

Another challenge is the definition of classes based on identified objects. "Object-oriented languages define a class as a description of a set of objects that share the same attributes and operations. Objects cannot exist without classes; they are created from class templates." [35, p. 153]. In the real world, as well as in Bunge's ontology, for example (on which Evermann and Wand base their domain modeling approach), things (analogous to objects) exist independent of any classes.

Empirical studies, typically focusing on novices, shed additional light into the challenges of defining objects and classes. Novice OO developers tend to follow a method learned in their programming courses and identify domain entities first, then classes, and then methods [1,3]. They seem to hope that the classes thus created will be useful later on. In addition, novices, as well as experts, try to define the class model before they implement the methods in detail [3]. However, research shows that classes created in the earliest stage of design are often not useful later on.

Novices tend to think that instances of classes are created by default; thus, they tend to use instances of classes without creating them beforehand (this error is similar to forgetting to initialize variables in the procedural paradigm). This type of error is also attributed to the analogy to the real-world. Initializing a variable or creating an object is a type of prerequisite that is not necessary in the real-world situation and so is not part of the knowledge transferred from the real world. In general, novices learn by forming analogies between real-world source situations and OO target situations [31].

2.2. Functionality

An important aspect of OO problem understanding and decomposition is the assignment of functionality to the specific objects. In the OO paradigm, much of the functionality assignment is a reflection of the *responsibilities* of objects in the world. "[A]n electronic checkbook is defined to hold a balance, have money deposited and withdrawn from it, and keep a record of these transactions, because these are the kinds of interactions we have with checkbooks in the world" [21, p. 367]. Rosson and Alpert, however, also note that the behavior of entities in an OO system often goes beyond that of the problem entities being modeled. They assess that "Although the assignment of active behavior to inanimate objects may seem to be an unnatural process, humans appear to readily adopt an anthropomorphic perspective in their efforts to understand computing objects" (ibid, p. 367).

Many psychological accounts of design have focused on the decomposition process [18–20]. Decomposition refers to the breaking down of higher-level concerns (e.g., an elevator control system that tracks elevator position) into sub-problems implied by those concerns (e.g., one way of tracking position is to have a central controller that polls individual elevators on a regular basis) [21].

The OO paradigm relies heavily on decomposition. Decomposing the problem consists of identifying objects and the associations between their structures and functionalities. It is assumed that the activity of decomposing a problem into objects is natural and driven much more by a designer's knowledge about world structure than by knowledge about the design process or about particular software domains. Rosson and Alpert [21] suggest that OO may be especially valuable in new domains or when practiced by relatively experienced designers. In procedural paradigm problems, in contrast, decomposition is driven by generic programming constructs and specialized design knowledge [31].

Booch [36] describes decomposing the problem into objects and their properties and functions as a simple task, which requires to simply underline the applicable nouns and adjective–verbs respectively. According to Svetinovic et al. [3,4], however, these tasks are not that simple and are not sufficient for the production of high-quality OO models. Détienne [31] claims that one of the main difficulties experienced by novices is the articulation between declarative and procedural aspects of the solution. Some novices do not succeed in decomposing the large procedure into smaller functional units and they associate the procedure as a whole to a single class.

Novices were found to spend a considerable amount of time creating and abandoning domain entities before they actually start defining objects. The activity of finding classes of objects appears to consume the novices' attention and they pay little or no attention to functionality until late in the design activity. In fact, novices delay considering goals and defining methods until relatively late in the design process [3].

One of the main difficulties experienced by novices is the articulation between declarative and procedural aspects of the solution, e.g., the hierarchy of classes and the main procedure [37]. Based on knowledge in the problem domain, novices identify the structural properties and organization of classes; based on knowledge in the programming domain, they identify procedural aspects of the solution. As a first step, they often construct a procedure to deal with the main goal (a complex plan) as well as procedures for performing typical roles (simple plans). The complex plan represents a large procedure, not necessarily expressed at the code level yet, and not yet decomposed into individual functions, which could be attributed to different classes. "When they refine the functions by writing the code of methods, novice OO designers are able to refine some characteristics of the structure of objects already defined at higher levels of abstraction, e.g., they add attributes in a class or add parameters in a method signature. However, they also make more drastic changes to the declarative model." ([31] p. 52). The lack of appropriate knowledge in OO and the transfer of inappropriate plans from procedural languages do not allow the construction of a correct complex plan and the allocation of actions of this plan among classes. When they construct a more detailed representation of the procedure (developed in the procedural part), they revise the static aspects of the global solution, i.e., parts composing a complex plan or the association between the actions of this plan and the objects. This suggests that constructing a representation of some procedural aspects of the solution precedes the construction of more declarative aspects of the solution, e.g., the objects (ibid).

2.3. Inheritance

Additional difficulties are hidden in the so-called natural decomposition mechanisms used in the OO paradigm. Inheritance provides a convenient mechanism for capturing a natural structure of shared attributes. For example, road bikes and tandem bikes are similar in that they can both be ridden, but they differ in their usage; there are many different types of bikes, but they vary in appearance and in their mechanisms of operation. Humans are quite good at recognizing these relations, and seem to develop internal representations of this natural structure through their interactions with the world [38]. When designing an architectural tool, rather than decomposing the problem into independent objects such as tandem bikes, mountain bikes, road bikes, and so forth, designers can explicitly capture entity relations using the inheritance mechanism, to yield "bike" as a superclass, and "tandem bike" and "road bike" as subclasses of that superclass. Not only does this allow the objects to share functionality as appropriate (code reuse), but it also permits designers to align their solutions with their knowledge of the real world [21].

This cognitive perspective raises, however, some concerns over the kinds of relations analysts and designers are most likely to perceive or create among objects in the world. Rosch [38] notes that natural hierarchies possess a basic level of abstraction at which similarities among actual objects in the world are most reliably and quickly recognized. For example, a particular Labrador dog is more easily identified as a dog than as a Labrador or a mammal. This distinction, for example, does not exist in the OO approach. Thus, there are some differences between the OO inheritance and the way we naturally interpret inheritance. Such differences could explain why Cartwright and Shepperd [39] found that software components involved in inheritance contain three times more defects than components not thus involved.

Another aspect that may affect OO problem decomposition and the resulting class hierarchy is the designers' knowledge about the target system. Designers familiar with the system will organize the decomposition of the new problem according to their knowledge of existing systems' classes, thus promoting the goal of code reuse, which is one of the core principles of OO. In contrast, designers that are unfamiliar with their target OO systems might be induced by the naturalness of object-based decomposition to produce designs with unnecessary and possibly inefficient distinctions as well as erroneous inheritance structures [21].

Novice OO developers have misconceptions about the fundamental OO concepts of class and inheritance [3,21,27,28]. They tend to conceive a class as a set of objects and therefore attribute a set of properties to the class concept. For example, when they define a function that processes a set of objects of classX, they place this function in classX instead of creating a classY whose type would be "set of objects of the classX" and placing the function in classY.

A different difficulty found in the context of inheritance is that novices try to use this property as often as they can, and they often do it inappropriately. In particular, they tend to use the abstraction hierarchy to express a composition. There are two aspects of inheritance: the inheritance of static characteristics of a class, i.e. the type of the class, and the inheritance of functional characteristics, i.e. the methods of a class. Observations show that, while novices tend to have difficulties using the inheritance property of static characteristics, they have even greater difficulties using the inheritance of functionality [3].

2.4. Summary

The cognitive difficulties associated with the OO paradigm summarized above mostly refer to difficulties identified when observing novice OO developers. Part of the reason for this is that, while much research has been conducted to uncover novices' difficulties with OO, little effort has been invested in revealing difficulties experienced by practitioners. This research focuses on experienced software developers and identifies that they too experience difficulties in OOA/D, including some of the difficulties discussed above. In addition, this research attempts to understand the source of these difficulties based on the natural human way of thinking and its manifestation in the context of OO development.

The aim of focusing on experienced software developers rather than on novices was to exclude learning-related difficulties that can be overcome given time and experience. This enables to identify the cases where the OO approach and natural thinking conflict so persistently that even experience (at least to some extent) does not suffice to overcome it. Identifying the principles in OO that inherently conflict with natural thinking may help not only in designing proper education programs but also in bringing these difficulties to the awareness of the community towards future evolution of development approaches.

3. Research method and settings

3.1. Research approach

The objective of this exploratory study is to identify and understand difficulties professional software developers encounter when performing OOA/D, as well as their sources. When aiming to explore and understand a phenomenon and its different aspects, it is appropriate to use a qualitative research approach [40]. Specifically, the generation of a descriptive and explanatory theory calls for the use of grounded theory methodology [41,42].

In this research, data were iteratively collected from the field and analyzed applying inductive approach. No theories were pre-decided; rather, relevant theories were investigated throughout the iterations of data collection and analysis. Categories that describe types of difficulties software developers encounter when practicing OOA/D emerged and were iteratively examined and refined until their saturation was achieved. In parallel, a theory was formulated that explains the findings based on both analyzed data as well as the literature. When applying grounded theory approach, consideration of literature is allowed for guiding data analysis [42,43]. In this research, while looking for explanations of the emerging findings, additional literature review conducted with the findings in mind revealed the relevance of the dual-process theory of cognitive psychology. Analyzing the data through the lens of the dual-process theory enabled to further understand and frame the investigated phenomena.[2]

3.2. Setting and participants

The population of this study was defined as software developers who practice OOA/D on a daily basis. The main study setting was advanced OOD workshops focusing on UML, conducted in industry for OO developers (for more details about the workshops, see Appendix). These workshops were an excellent means for observing the process of OOA/D problem solving as well as collecting written solutions, from a relatively large number of participants. During these workshops, participants discussed tasks and examples of design problems and were asked at times to solve tasks either individually or in pairs.

Three instances of the workshops were selected for the study according to criteria that will be explained below. Ten to sixteen software developers participated in each workshop. Each workshop lasted five 8-h days for a total of 120 h of workshop activities that were observed and documented.

Participants were selected to be included in the study according to theoretical sampling principles [41]. However, the selection was not done one participant at a time. Since most of the study was conducted in workshops, as explained above, groups of participants were selected. Prior to each workshop, the list of participants in the workshop was made available to the researcher, including some basic data the participants were required to fill out during registration. These data included, among others, education and experience. Only workshops in which all (and in one case most) of the participants were eligible according to the theoretical sample as defined in this research were selected to be observed. There was one participant in the observed workshops that was not eligible, and no data related to this participant were included.

Several factors were considered when deciding on the sampling criteria. The first was to ensure that the subjects practice OOA/D on a daily basis. Second, only software developers that studied OO programming and design during their undergraduate studies and had worked only with the OO paradigm since their graduation were included. This was important as the scope of this research was defined to exclude difficulties stemming from prior experience in procedural programming, a topic that has already been thoroughly investigated [5,6,25].

In addition to defining the general research population profile, it was also important to ensure that specific participants chosen differ from each other in other aspects so that properties irrelevant to the phenomenon explored do not limit the

2 See an example of analyzing findings of grounded theory study through the lens of psychology in [44].

Table 1
Data collection.

Data	Collection tool	Purpose
Written solutions	Hand-written models formulated by participants during class exercises	Analyze the quality of the solutions and identify non-normative OOA/D decisions
Real design documents	Printed diagrams of developing or existing software systems the participants had worked on	Analyze the quality of the solutions and identify non-normative OOA/D decisions in real projects
Class discussions	Observations	Identify underlying assumptions and rationale of OOA/D decisions
Pair or group discussions	Observations	Identify difficulties during OOA/D problem-solving processes
Clarifying interviews	Semi-structured interviews focusing on the topic to be clarified	Clarify specific solutions suggested (in written solutions) or opinions expressed (in observed discussions).
Interviews with experts	Semi-structured interviews focusing on data analysis	Collect experts' opinions about solutions suggested by the study's participants to prevent researcher bias.

results and that different types of developers that are employed in industry will be included. To this end, the following aspects of the participating candidates' backgrounds were considered: educational background, professional experience, and position/seniority in the organization.

A total of 41 software developers employed in industry from seven different companies participated in the study, all of whom had at least a bachelor's degree in computer science, software engineering, or information systems and professional experience of 1–12 years as software developers, engaged in OOA/D. Five of the participants were development team leaders, and 36 participants were development team members.

Two main differences between the setting of the workshops and real-life software development practice should be discussed. First, during the workshops, participants worked on tasks usually within a convenient timeframe causing little, if any, time pressure. Arguably, this does not simulate the time pressure software developers experience in real-life development situations. Second, the analysis and design tasks the participants were given were naturally significantly smaller and less complex than the usual problems in practice, due to the timeframe and focus of the workshop. However, these two differences are not necessarily a limitation of the study and its findings. One would expect fewer mistakes or hasty non-normative decisions to take place in simple tasks with less time pressure than in the real-life situations of tight schedules and more complex tasks. Thus, the difficulties identified in this study within its convenient setting for the subjects are expected to be even more prominent in practice.

3.3. Data collection

Data collection was based mainly on: (1) written solutions collected from the participants after each task during the workshop, mostly represented as UML models; and (2) observations of class and pair discussions, which were audio-recorded and transcribed. In addition, individual interviews were conducted whenever the need for clarification of previously collected data arose. These interviews were aimed at gaining an understanding of the participants' thought processes that led to the solutions or opinions presented. On some occasions, participants brought examples from their own working experience, including real design documents of either developing or existing software systems. When possible, a copy of these documents was also submitted and included in the research data.

Table 1 summarizes the data collection tools used in this study and their purposes. The multiple sources of the data collected, along with the three different workshop settings, enabled triangulation, thus increasing the validity of the findings [41].

3.4. Data analysis

Data analysis was executed based on concept analysis according to the inductive analysis approach in which categories emerge from the data and are validated and refined throughout the analysis process [41,45,46]. During the data analysis, the dual-process theory was found to shed light on the emerging findings and was integrated into the analysis process.

The purpose of inductive analysis is to identify recurring themes, which serve as the basis for the categories, and to define their properties and dimensions. This was consistent with the objective of the research, i.e. to identify difficulties in OOA/D. The data-analysis process includes open, axial, and finally selective coding [47] to determine the categories of difficulties encountered in OOA/D practice and their sources.

During open coding, the data was fractured to segments, which were then evaluated and screened. The segments were composed of participants' statements, opinions, suggested models, etc. obtained from the observations, written solutions and interviews. Each of these segments was evaluated based on OO literature and validated with experts (see Table 1). The experts included two software engineering professors from two universities, the workshops' instructor (Ph.D. with many years of practical experience in OO development), and a highly experienced software engineering lab manager from one of the university computer science faculties. During the data analysis iterations, each set of data – comprised of classification of

OOA/D decisions to normative and non-normative – was validated with one of these experts, according to their availability. In all cases the experts confirmed the classification of the researcher (which was based on OO literature and individual knowledge).[3] Only segments that were classified as non-normative (i.e. not consistent with the OO paradigm) were included in the final open coding, in which they were coded and classified to iteratively emerging categories. For example, the text segment ''All of the methods and properties are defined here as public.'' analyzed in the context of the design at hand, was coded as ''difficulty to apply information hiding''. A model construct presenting two classes with no hierarchy of an abstract class from which they both should inherit (as appropriate in the context of the design task at hand), and the text ''I would instantiate an object from this [abstract] class'' were both coded as ''difficulty to work with abstract classes''.

The normative segments also played a role in the analysis. They were helpful when trying to determine whether a certain participant, making a non-normative decision, has the relevant knowledge needed to make the normative decision. These segments included segments obtained in observations and written tasks as well as follow-up interviews. In cases where not enough evidence was available, the follow-up interviews were intended to determine whether the participant had the required knowledge.

The general trend that emerged during the open coding was that in many cases subjects displayed difficulties in basic concepts of OO. Moreover, they seemed to make non-normative decisions that contradicted knowledge they evidently held. The most widely accepted explanation in cognitive psychology for this phenomenon is the dual-process theory [8,48], therefore this theory was applied for further analysis of the data.

The axial coding included the process of understanding how the categories identified in the open coding relate to each other and classifying categories under common themes, thus creating hierarchical classifications. For example, a relationship between the codes of ''difficulty to apply information hiding'' and ''difficulty to work with abstract classes'' was created and later combined to the common theme of ''difficulty to apply abstraction principles''.

In the selective coding process a theory was sought that identifies a coherent phenomenon related to the difficulties that emerged in the previous phases. The coherent phenomenon found during the analysis was that *software developers make non-normative decisions that are inconsistent with basic knowledge they evidently hold, due to different surface clues that misguide them in their thought process*. This theory emerged based on integration of the data analysis described above and the dual-process theory, which was found relevant to the phenomenon identified in the analysis process.

During the iterative data-analysis, category saturation was achieved after analyzing the second workshop's observation data. The analysis of the data collected in the third workshop was consistent with the categories already identified and characterized and with the theory that emerged. An elaborated, finding-based description of the data analysis is presented in Section 5. Section 4 presents the dual-process theory, as a basis for the analysis presented in Section 5.

4. The dual-process theory

The findings of this research presented two challenges: The first was to explain why experienced software developers exhibit difficulties in the most basic concepts of the OO paradigm. While attempting to analyze this phenomenon, a second challenging observation was made: software developers often make decisions that contradict knowledge they evidently hold. Searching cognitive psychology literature for explanations led to ''the rationality debate'' and the dual-process theory. In this paper, the theoretical framework of the dual-process theory will be based mainly on the 'heuristics and biases' research work conducted by Kahneman and Tversky over about 30 years (a summary of their work can be found in Kahneman's Nobel prize lecture [8]) and its application to mathematics education [9–11]. This theory provides the basis for the theoretical framework that was developed and applied to explain the findings of the current research.

Researchers in cognitive psychology have been analyzing human rationality and decision making for many years. Empirical evidence shows that people often make ''irrational'' decisions, in the sense that they are in contrast to logical, mathematical, or statistical processes based on known data [8,49–51]. These phenomena exemplify situations of consistent ''irrational'' decision making shared by a large percentage of the subjects. There is, however, a basic disagreement in the cognitive psychology community regarding the essence of ''irrationality'' as defined by some researchers. Western philosophy of the past several centuries, defines an irrational decision as a decision that contradicts known logical, mathematical, or statistical principles and data known to the decision maker [8,48]. Other researchers in the cognitive psychology community claim, however, that the laws of human rationality, as evolved throughout the evolution of mankind, are, in many cases, different from the rule of logic [52,53]. Thus, the debate does not concern ''irrationality'' but rather a different kind of rationality that is at times inconsistent with the formal, logical rationality.

A typical example of consistent irrational behavior is the bat and ball experiment:

Question. *A baseball bat and ball cost together one dollar and* 10 *cents. The bat costs one dollar more than the ball. How much does the ball cost?*

Almost everyone reports an initial tendency to answer '10 cents' because the sum $1.10 separates naturally into $1 and 10 cents, and 10 cents is about the right magnitude. Indeed, many intelligent people yield to this immediate impulse: 50%

[3] The role of the experts was originally planned to help evaluate complicated design decisions. However, since the study eventually focused on basic concepts and principles, the experts merely served as a precaution, in addition to OO literature, against researcher bias.

(47/93) of Princeton students and 56% (164/293) of students at the University of Michigan gave the wrong answer ([50] p. 58; [8] p. 451).

Many different theories were developed to explain phenomena of the type demonstrated above. One commonly known theory is the dual-process theory, which claims that two different, separate thinking systems operate in parallel in our minds [8,53,54]. One system, System 1 (S1), is responsible for the immediate, automatic thinking that solves familiar problems without using complex, analytical data processes. Being highly available and automatic, S1 is triggered first and tries to solve the problem it faces using simple and fast heuristics. The other system, System 2 (S2), handles problems that require more disciplined thinking processes, in which calculations and analytical data processing is required. S2 is also responsible to monitor, and override where needed, responses provided by S1.

The problem-solving capabilities of S1 depend on different personal characteristics such as age, culture, education, experience, and so on, and it may vary accordingly. For example, for adults, the task of solving simple arithmetic problems, e.g. adding two one-digit numbers, is an automatic activity that requires almost no attention and can therefore be assumed to be executed by S1. However, for a 5-year old child, the same activity requires cognitive effort and concentration, and will usually include a logical and time-consuming process (for example, counting fingers). Such an activity, requiring this kind of resources, is executed by S2. Hence, complex activities can also transfer from S2 to S1 after sufficient experience is gained. Another, more complex example is driving a car [10]. When a person learns how to drive, he or she must think about each and every activity this complex process encompasses, such as shifting gears, looking in the mirror and at the surrounding vehicles, obeying traffic lights, etc. After gaining sufficient experience, however, driving becomes an automatic activity for the most part, enabling drivers to invest their cognitive effort in a complicated conversation for example while driving.

Classic situations that demonstrate inconsistency between analytical rationality and human intuition are those that involve decision making under conditions of uncertainty. One of the most discussed problems in this area is "the Linda problem" [55,56]:

Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations.

Which is more probable?

- Linda is a bank teller.
- Linda is a bank teller and is active in the feminist movement.

Presenting the Linda problem has repeatedly resulted in a very high percentage (85%–90%) of subjects who found the second possibility (the conjunction) to be more probable than the first. This decision contradicts the laws of probability, since the probability of a conjunction of events (bank teller *and* active in a feministic organization) will always be equal or lower than the probability of one of these events (bank teller).

The dual-process theory was found to be useful in explaining cases where people make systematic "irrational" decisions that contradict their own knowledge [8]. Kahneman describes S1 as an intuitive system that activates simple heuristics for solving familiar problems. However, in its hurry, S1 may at times activate heuristics that are indeed highly available but are nevertheless inappropriate for the problem at hand, due to biases caused by surface clues. This results in non-normative decisions, namely decisions that are not consistent with formal logic, due to the evaluation of a complex situation as if it were simpler or different from what it is.

S2 is responsible for executing heuristics, calculations, and decision making in situations that require complex data processing and calculations, whose solution is not immediate and so no intuitive solution comes up. This is not to say that S1 does not deal successfully with complex situations; S1 is capable of such only based on vast personal or evolutionary experience. S2 is also responsible for monitoring decisions made by S1. When S1 activates a heuristic for solving a problem, it is S2's duty to make sure that this heuristic and the resulting solution are indeed appropriate and either accept or correct the proposed solution. In the above examples, S2 remained dormant and did not monitor S1, allowing the acceptance of a wrong decision made by S1. In the bat-and-ball example, although most subjects gave the wrong answer (10 cents), which popped into their head first, some delayed their answer for a short while and finally provided the correct answer. The difference between the subjects who gave the wrong answer and those who gave the correct answer is in the involvement of S2 in the problem-solving process.

Kahneman explains the non-normative decisions people make in light of S1's quick, almost automatic operation. Many factors influence S1 decision making; one is the emphasis placed on specific parameters within the problem. In the Linda problem, her description as being bright and outspoken, a person who participates in demonstrations, etc. leads us to anticipate an appropriate occupation. Thus, the search for the occupation that fits these characteristics overshadows other considerations, such as the probability consideration, and activates the heuristics that decides on her occupation. There is in fact a substitution of characteristics, weighting some of them as more relevant than others, causing a biased heuristic selection. In this case, Linda's characteristics gain exclusive importance over other relevant (probability-based) parameters. For most people, confidence in the S1 decision in this case is so high that S2 stays dormant and fails to activate its proper tools that are capable of handling probability issues.

Another factor that affects S1 decision-making is illustrated by the bat-and-ball example. According to the dual-process theory, this common error relates to the high accessibility of the numbers 1$ and 10 cents. These numbers appear in the text

Table 2
Categories identified as possible sources of difficulties in basic OOA/D.

Category	Description
Salience	Concepts explicitly mentioned in the problem or analysis of the system to be.
Metaphors	Metaphors used in the programming paradigm based on daily concepts.
Personal experience/ knowledge	The individual knowledge each person has based on his or her own experience regarding the problem domain.
Abstraction	Different forms of abstraction in OO (information hiding, abstract class, etc.)

preceding the question and therefore are more accessible in the initial thought than any number not mentioned in the text (e.g. 5 cents). This accessibility triggers S1 to jump up with the answer 10 cents, which with the 1$ (the difference between the bat and the ball) completes the original sum (1.10$). Since 10 cents is roughly of the right order of magnitude, it seems this is indeed the correct answer. Many people accept the answer provided by S1 while S2 stays dormant and does not execute corrective actions, thus displaying non-normative behavior. However, there are people whose S2 is activated following S1's hasty answer, and who then calculate the answer analytically to obtain the correct answer. According to Kahneman, S2 often stays dormant because processes such as substitution of the relevant characteristics of a question take place unconsciously and lead to conclusions which the subjects seemingly have no reason to doubt. This lowers the accessibility of corrective thoughts.

The dual-process theory was recruited by Leron [9,10] and Leron and Hazzan [11] to explain phenomena in the context of mathematical education. Leron [9,10] reviews insights gained through empirical research in cognitive and evolutionary psychology and their implications on mathematical thinking. For example, he analyzes the differences and contradictions between formal analytical thinking and natural human thinking. According to evolutionary psychology, S1, being ancient, is more attached to natural human thinking and its rationality works according to what was required of humans in order to survive in the environment and culture that existed hundreds of thousands and even millions of years ago. S2, which evolved much later, is capable of dealing with modern needs such as analytical thinking.

Leron and Hazzan [11] analyze students' misuse of the Lagrange theorem and its converse when solving problems in group theory. Students show a tendency to use Lagrange theorem (and generally theorems with names) in inappropriate situations. The reason for these misuses is the accessibility of these theorems in students' minds and the overweighting of some of the question's parameters while neglecting others (similar to the Linda problem). Leron and Hazzan [11] indicate that there is something unexpected in the fact that S1 errors are observed in university mathematics classes, where thinking with S2 is seemingly the name of the game.

In this paper, findings will be presented and analyzed based on the identification of thinking processes originated in S1 thinking. The next section presents examples of non-normative decisions made by experienced software developers and analyzes them in light of the phenomenon presented above: S2 stays dormant while S1 makes a non-normative decision due to the application of an inappropriate heuristic.

5. Findings

This section presents difficulties relating to basic concepts and principles of the OO paradigm, which were identified when observing software developers practicing OOA/D. Specifically, it focuses on difficulties that can be explained by the clash between the intuitive and analytical thinking modes. This section includes three sub-sections: 5.1 presents the categories of possible sources of difficulties in OOA/D that were identified, illustrates them using typical examples of observations collected in this research and analyzes these observations based on the dual-process theory; 5.2 aggregates and generalizes the research observations and their analysis and discusses the potential influence of intuitive thinking in this context; and 5.3 discusses the threats to the validity of the findings and conclusions of the research.

5.1. Sources of difficulties in basic OOA/D

Table 2 presents categories of possible sources of difficulties in basic OOA/D identified in this research (as explained in 3.4) followed by some typical examples to illustrate these categories. The analysis of the examples includes the identification of these sources as well as a complementary analysis through the lens of the dual-process theory.

5.1.1. Example 1, salience

Background: One of the first tasks in OO development is to determine the classes required to implement a requested solution.

Workshop participants were asked to perform the following task:

Design an authorization system that will route users as follows:

- An existing user will login into the system.
- A new user will register and receive authorization.

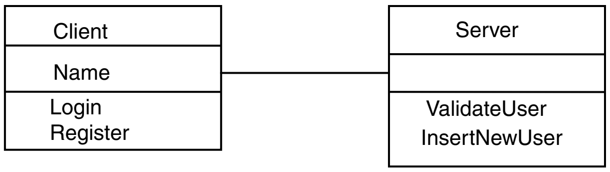

Fig. 1. Proposed solution, partial class diagram.

A typical design can look like the one depicted in Fig. 1.

Observation: The following discussion took place while the participants were working on the task in pairs.
Ron: Let's define *Login* and *Register* as objects.
Sharon: Do *Login* and *Register* seem like objects to you?
Ron: Why not?
Sharon: An object is a *Client*, for example.
Ron: *Client* is also an object. *Login* and *Register* are activated and operate within the system; therefore they can be defined as objects.
Sharon: I've never seen an object *Login*.
Ron: Don't worry, it will be ok. You'll see how I design the system, it will be just fine.
Sharon [hesitates, at last reluctantly giving in] : Ok, fine, although it doesn't sound good.

Participants' background: Ron has two bachelor's degrees, one in physics and one in software engineering, and 7 years of experience in software development.

Sharon has both a bachelor's and a master's degree in computer science, 8 years of professional experience, and has recently been promoted to team leader role.

Analysis: Ron solved several tasks during the workshop and participated in different discussions. In other events, Ron did not demonstrate difficulties in identifying objects, and specifically there was no observed tendency to define procedural elements as objects.

Ron's decision exhibits typical S1 behavior, similar to that observed in the bat-and-ball task. When searching for objects he is influenced by the surface features of the task – the salience of the terms *login* and *register* in the task description – rather than its essential (though implicit) components. Unlike the bat-and-ball phenomenon, Ron requires more than a nudge to change his mind, which seems to imply that his S2 knowledge on this matter, although it evidently exists, is not too strong either.

Sharon, in contrast, seems to have a stronger feeling for the correct objects, but this too seems to be an S1-based decision, as implied by the fact that she cannot explain her choice. She attempts to convince Ron using expressions like "doesn't seem like objects", "I've never seen an object" or "it doesn't sound good", which suggests that she is relying on her vast past experience (S1) rather than on analytical, rule-based reasoning (S2). Sharon's example, in contrast to the other examples presented in this paper, demonstrates how intuition may in fact contribute positively even in situations of formal problem solving [7].

5.1.2. Example 2, metaphor

Background: Let us consider the example presented in the introduction. The participants were given two classes, *Doctor* and *Paramedic*, and were asked to decide on an inheritance relation between the two.[4] *Doctor* was described as having all the capabilities of the paramedic as well as additional ones. Accordingly, the expected answer would be that *Doctor* inherits from *Paramedic* and extends its capabilities.[5]

Observation #1: The first reaction of one of the participants was to suggest that *Paramedic* inherits from *Doctor*. Following a short remark by the instructor, however, the participant immediately changed his mind. Similar questions in different contexts were asked in different workshops and resulted in the same reply.

Participant's background: The participant has a bachelor's degree in computer science and 12 years of experience.

Observation #2: In view of this and similar observations, we presented a group of ten software developers with a similar question in order to examine this phenomenon more directly. The answers were divided 5:5 between the two possible directions of inheritance. As in Kahneman's bat-and-ball example, the participants who chose the wrong direction required only a small nudge (with no informational or explanatory content) to immediately change their minds.

Analysis: These observations demonstrate a clear case of a wrong decision made by professionals who undoubtedly have the relevant knowledge required to make the correct decision. When asked (on a separate occasion) what inheritance means, they gave accurate answers. A possible explanation for this behavior is the mechanism used by Kahneman to explain the

4 At this point the instructor asked the participants to solve the problem using concrete classes only. Later on in the discussion, a solution using an abstract class was also discussed.

5 The discussion dealt with the "is-a" inheritance relation.

Table 3
Comparison between the meanings of the term 'inheritance' in everyday life vs. OO.

Attribute	Human inheritance	Class inheritance
Transferring the property	The property is *transferred* from the ancestor to the inheritor.	The property is *duplicated*; the inheritor receives it, but the ancestor continues to hold on to it as well.
The part of the property being transferred	In case of a single inheritor, s/he receives the entire property. In case of multiple inheritors, the property is divided between them, so that each heir receives only *part* of it.	In any case, whether single or multiple inheritors, each inherits the *entire* property.[6]
Additional property of the inheritor	The inheritor usually owns additional property *prior* to the inheritance. Of course, later on s/he will be able to gain more.	The type of the class is usually decided first, namely based on the existing class it inherits from, and *later* more properties of its own are added.[7]

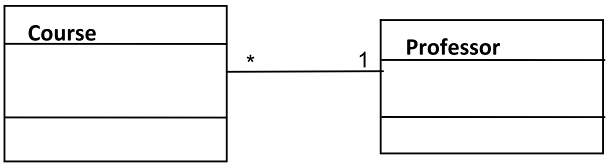

Fig. 2. Proposed solution, *Course–Professor* association.

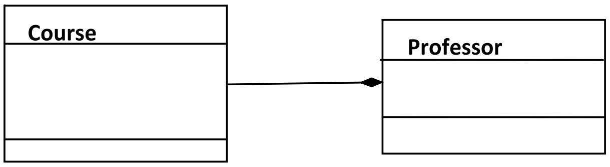

Fig. 3. The relationship between Professor and Course according to Guy's solution.

bat-and-ball phenomenon. Specifically, S1, with its quick and effortless operation, hijacks the thinking process and produces a response that seems roughly appropriate, while the slow and effortful S2 remains dormant. This analysis gains additional support from the observation that the small hint offered by the instructor did not teach the participants anything new, but only served to wake S2 up; the necessary knowledge was there all along, but the dual-system analysis explains why it was not mobilized [7].

Why would S1 and S2 clash about the meaning of *inheritance*? In people's everyday intuition (S1), inheritance is about transferring "stuff" (such as property or money), and the direction is usually from the person who has more to the one who has less. But in the OO formalism, the reverse is true: the class with more functionally inherits from the one with less. Table 3 summarizes additional differences between inheritance in human, everyday life and its meaning in OO.

Based on these significant differences between the two meanings of the term inheritance, errors such as the one illustrated above are to be expected. This may shed some light onto the vast amount of errors found in the context of inheritance in previous reported research (e.g., [1]).

5.1.3. *Example 3, personal experience/knowledge*

Background: The participants were asked to suggest a design for a course registration system at a university. The assignment specified that a professor teaches a course. Thus, participants were required to define in their solution the relation between the class *Professor* and the class *Course*. It was given that a professor can teach several courses, and that each course can be taught by only one professor at a given point in time. This one-to-many relation between these two classes can be represented by an association relation as presented in Fig. 2.

Observation: In his solution, Guy proposed the following relation between *Professor* and *Course* (Fig. 3):

Guy defined a composition relationship between these two classes; namely, the class *Course* is part of the class *Professor*. According to this relationship, *Course* cannot exist – i.e. hold data (such as syllabus, credit points, prerequisites) – without being assigned a professor. This relation is problematic since *Course* must exist and hold this data regardless of its teaching staff allocation.

Participant's background: Guy has a bachelor's degree in computer science and 3 years of experience.

Analysis: A follow-up interview with Guy explored this choice of relationship between *Professor* and *Course* and revealed that Guy perceives the professor as the person who develops the courses he or she teaches and thus has full ownership

6 Only public and protected properties are referred to here. For the sake of simplicity, all examples in this paper refer to public properties and functions.
7 For the sake of simplicity, the basic is-a inheritance relation is referred to here.

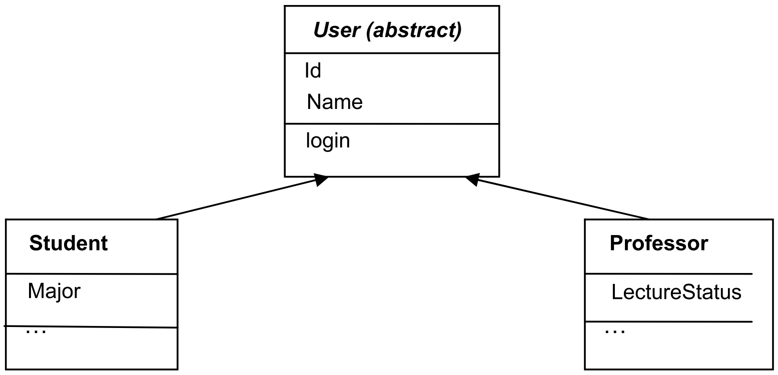

Fig. 4. Proposed solution using an abstract class.

of them. He did not take into account the possibility of courses being allocated to different professors. In this case, Guy's previous familiarity with the domain of the system he was asked to design seems to actually hinder his performance due to biases caused by his own past personal experience that contradicted the information he was given.

This effect of prior knowledge causing bias and fixation has been discussed in the general context of problem solving (see, for example, [57]). In the context of software engineering and information systems development, this difficulty is not exclusive to OOA/d and may apply to other areas, such as requirements analysis and conceptual modeling, regardless of development paradigm. Prior studies argue that pre-existing application domain knowledge impacts problem-solving effectiveness [58–60]. These studies mostly refer to the importance of application domain knowledge in requirements analysis related tasks and in particular conceptual modeling. This is not surprising, since much of what is considered to be software development is actually application domain problem solving using a software solution [61]. The phenomenon illustrated above, however, seems to justify the remark made by Baragry [34], that people automatically and subconsciously apply their accrued concepts and theories to the observed phenomena in order to understand them; consequently, people's explanatory theories do not capture the natural order of the world, rather they are subjective to the person using them.

Berge et al. [62] recommend letting students develop systems they are familiar with as their first OO development tasks. They also explain that students' individual experiences with similar systems may confuse them when deciding on software objects and the relationships between them, due to their perceptions of these objects in the physical world.

As mentioned, S1 uses perceptions based on our daily experience. In this case, Guy's perceptions that were formed based on his experience as a student added implicit information to the decision-making process; information that was not based on the requirements. It is important to note that in this example, the consideration that influenced Guy's decision is not problematic in the context of understanding OO — Guy's design was a good OO solution to the requirements, as he understood them. Rather, this situation illustrates the phenomenon of inappropriate solution due to misunderstanding of the requirements. In this case, S1 affected the way Guy understood the requirements.

5.1.4. Example 4, abstraction — abstract class

Example background: A design task given to the participants in a workshop included the two entities *Professor* and *Student*. Being users of the system, these two entities share many attributes and functionalities, but each of them has additional, different functionalities as well. This situation calls for using an abstract class, as demonstrated in Fig. 4.

Observation #1: Gil defined the classes *Professor* and *Student* separately, with no relation between them, even though they share many attributes and functionalities due to being users of the system. Failure to link these two classes will eventually result, at the implementation phase, in the creation of duplicate code. In fact, Gil had consistently avoided using abstract classes in his work.

Observation #2: Alex also avoided using an abstract class in his solution; however he did link the two classes. In his class diagram he models an inheritance relation, where the class *Professor* inherits from the class *Student*, while the latter holds the attributes of the user. This inheritance relation is problematic. Even though these two objects have a lot in common, they hold different functions as well. Defining *Professor* as inheriting from *Student* leads to *Professor* holding excess functionality (for example, summarizing credit points or calculating grade averages). The opposite inheritance direction would create an even more severe problem: *Professor* has many functions a *Student* is not allowed to have. Using an abstract class from which they both inherit solves this problem of excess functionality.

In follow-up interviews, both Gil and Alex gave correct definitions of the term abstract class, as well as reasonable examples for its use. When asked specifically why they did not apply this concept in the question above, Gil answered: "I didn't think about it; I guess I didn't look for commonalities" and Alex said: "I didn't think it was necessary". In the interviews both Gil and Alex agreed, following the question above, that a solution including an abstract class was better than the solution each of them proposed.

Participant's background: Both Gil and Alex have a bachelor's degree in computer science. Gil has 3 years of experience in OO design and Alex has 18 months of experience.

Analysis: Failure to define an abstract class when needed, especially for practitioners with relatively little experience, may stem from the way OO is initially taught. When analyzing system requirements in free text, instructors customarily recommend marking all of the nouns that appear in the text as the first stage in deciding on objects (e.g. [12]). In one of the workshops, a participant even described it explicitly in a group discussion, while trying to understand the requirements of the design task they received:

Michael: After you write the Use Case, you extract the nouns and define them as classes. Then you look for the verbs for the sequence diagram . . . [smiles in embarrassment] Right? At least this is how it was taught when I was a student . . .

It is possible that after this preliminary stage, some software designers, especially relative novices, do not continue to look for additional classes required for an efficient system design, but rather stick to the original concrete classes (with direct instances). This may force problematic and incorrect inheritance relations or duplicated code.

In a pilot study conducted in the initial stages of this research with senior students of computer science education, we asked the students to solve a different problem that required the use of an abstract class. Out of 21 students (all of whom had previously studied at least one OO programming course), 16 presented in their design only concrete classes that were mentioned in the textual requirements and worked hard, making many (unsuccessful) attempts to find appropriate inheritance relations to solve the problem. Interestingly, in a questionnaire they completed after performing the task, in which they were asked to define various OO related concepts, most of these students gave reasonable definitions for the concept "abstract class". While not all definitions were full and accurate, they did imply that these students were familiar with the concept of abstract class and were aware of the basic situations in which it might be of use. The effort they invested in solving the problem without using an abstract class may indicate that although they have basic knowledge of the concept, they have difficulty putting it to use.

Gil and Alex, both of whom have relatively little experience, exhibit a naïve approach to determining classes and designing the system. Even though they are familiar with the concept of abstract class (as was clearly evident in different discussions in which they participated), it seems that during the process of solving design problems they are unaware of this possibility or are unable to identify cases in which abstract class may be of use. This is again an example of knowledge that exists in S2, however is not mobilized during problem solving.

"Abstract class" is, naturally, an abstract concept. It is, however, important to understand that the location of a specific concept in the continuum between concrete and abstract is not an inherent property of the discussed concept, but rather a property of the attitude of the person towards this concept [63]. The closer a person is to a specific concept, the more concrete it becomes. Thus, a person who has vast experience and skills involving a specific abstract concept may be able to process it appropriately in S1 thinking. This is obviously not the case with Gil and Alex; they both have difficulties working with this abstract concept.

5.1.5. Example 5, abstraction — information hiding

Example background: Information hiding is a basic and important principle in the OO paradigm. It is the principle that enables an object to be independent and autonomous. Object *A* operates following a service request received from Object *B*. The service provider, *A*, executes the required actions independently and returns the required output to the requesting object, *B*. *B* does not know the functions executed by *A*; it is familiar only with its interface — namely the functions that are defined as public. Thus, each object contains hidden functions. Similarly, an object updates its own data. It can receive inputs from the outside, but it validates this input and changes the value of the relevant parameter.

Observation: The following dialog was observed between a student and an experienced software designer in a high-tech company. The student was a member of a project team supervised by the experienced designer.

Designer: All of the methods and properties are defined here as public.
Student: So you allow whoever uses this object to access all of its methods and properties?
Designer: Of course not. The developers in our group all know what they are allowed and what they are forbidden to do; which members are allowed for use and which should be hidden.
Student: Do you have any documentation about this?
Designer: No need for that. We all just know.

A subsequent conversation with the students revealed that it took them two weeks of hard work to find out what is allowed and what is forbidden before they could start developing the new component required for the system. Moreover, they found that the objects did not contain any validity checks for changing the values of objects' properties. Since all of these properties were defined as public, they were exposed to the entire system and subject to changes with no control over their values.

The original development team did not apply the principle of information hiding. Information hiding contributes to controlling the correctness and integrity of data and reduces the dependencies between objects. One of the benefits of this principle is that when developing a new component for the system, the developer is required to know only the interfaces with the other objects and has no need to be familiar with their code. When the functions and data are not hidden and protected by various verifications, all of the systems' components must know the laws that govern each of the objects they interact with.

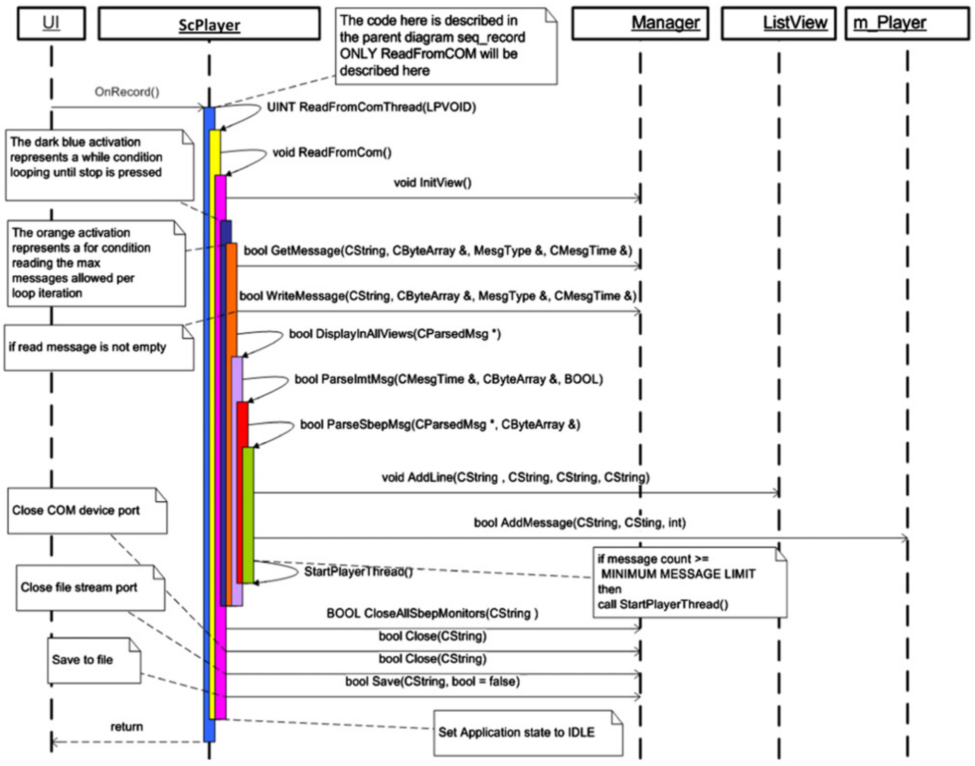
UI
ScPlayer
The code here is described in the parent diagram seq_record ONLY ReadFromCOM will be described here
Manager
ListView
m_Player
OnRecord()
UINT ReadFromComThread(LPVOID)
The dark blue activation represents a while condition looping until stop is pressed
void ReadFromCom()
void InitView()
The orange activation represents a for condition reading the max messages allowed per loop iteration
bool GetMessage(CString, CByteArray &, MesgType &, CMesgTime &)
bool WriteMessage(CString, CByteArray &, MesgType &, CMesgTime &)
if read message is not empty
bool DisplayInAllViews(CParsedMsg *)
bool ParseImtMsg(CMesgTime &, CByteArray &, BOOL)
bool ParseSbepMsg(CParsedMsg *, CByteArray &)
void AddLine(CString , CString, CString, CString)
Close COM device port
bool AddMessage(CString, CSting, int)
StartPlayerThread()
if message count >= MINIMUM MESSAGE LIMIT then call StartPlayerThread()
Close file stream port
BOOL CloseAllSbepMonitors(CString)
bool Close(CString)
bool Close(CString)
Save to file
bool Save(CString, bool = false)
return
Set Application state to IDLE

Fig. 5. A sequence diagram taken from a project's design documentation. Note: The objects' names were changed slightly for confidentiality reasons.[8]

Additional support for these findings is apparent in the following sequence diagram (Fig. 5) that is part of the design documents of the same system. In this diagram, an object *UI* (User Interface) requests service from the object *ScPlayer*. *ScPlayer* activates a function to provide this service, which calls another function for service, which in turn calls a third function, and so on. In all, eight functions are activated in this scenario, all belonging to the same object.

The reason we see all of the internal methods that operate in the object *ScPlayer* in this diagram is that they are all defined as public. The information hiding principle guides to hide internal methods of an object, leaving apparent only those methods that provide service for other objects in the system. In this case, presenting only the method *OnRecord* in the diagram, which is the method that object *UI* activates for service, would have been more consistent with the information hiding principle. According to this principle, all other methods that are activated within the object, and which have no connection to other parts of the system, should have been defined as private.

Another problem in this example is the concentration of responsibility in a single object (*ScPlayer*), while the other objects are inactive in this scenario. In fact, even when *ScPlayer* initiates the object *Manager*, there is no evidence in this diagram that such an object is indeed created. Moreover, *ScPlayer*, which is the only active object except *UI* in this scenario, performs activities that should quite clearly be executed by other objects. For example, the method *DisplayInAllView* should be performed by the object ListView. This is consistent with the findings of Détienne [31], who reports a tendency to allocate a big procedure to a single class rather than decompose it and distribute its parts between different classes.

Analysis: The approach according to which all functionalities and parameters are open to the entire system and responsibility is concentrated in one place rather than being distributed among different components of the system is reminiscent of the procedural approach, which is more concrete and better handled by S1. One of the contributions of OO technology is its high modularity. Each software component is autonomous in its operation. The integration between the different components is achieved by using the objects' interfaces, thus decreasing the complexity of the system. The design in the example above does not exploit this potential and so increases the complexity of the system and the dependency among its components, both of which lead to low-quality design which results in difficulties in the development and maintenance of the system [64].

5.1.6. Example 6, abstraction — design/code abstraction barriers

Background: This example is an excerpt from a longer discussion regarding multiple inheritance. Multiple inheritance, a situation where a certain class inherits from two or more other classes, occasionally enables a convenient and effective

[8] A syntax error appears in this diagram: The call of one method to another within the object *ScPlayer* was incorrectly marked as recursive activation. Since this is a pure syntax error, which does not appear to reflect an essential understanding difficulty, it is not discussed here.

design. It may, however, lead to problematic and dangerous situations, such as the diamond structure: a situation where classes B and C both inherit from abstract class A and implement an abstract function *f* in different ways; then class D inherits from both B and C, thus inheriting two different implementations of the same class *f*.

Observation: Participants in the workshop where the following example was observed were software developers who design and implement systems with C++, an OO programming language that supports multiple inheritance. The next section presents Roy's opinion on the solution of a problem with tempting conditions for using multiple inheritance: A class called *Fax* required several functions that already existed in other classes, some in a class called *Mail* and some in a class called *Form*. At this point of the discussion it had already been decided that the class *Fax* should inherit from *Form*, and the discussion continued to explore the possibility that *Fax* will also inherit from *Mail*.

Roy: I wouldn't use multiple inheritance. I wouldn't inherit from *Mail* [in addition to inheriting from *Form*].
Instructor: But you need the four functions from *Mail*.
Roy: Then I'll write them again.

Roy had several possibilities for solving this problem on the design level; multiple inheritance was one of them. Roy, however, felt insecure in using this solution since he was aware of its inherent risks, and apparently was not sure he would be able to avoid them. But there are other solutions that could be found on the design level, without going into the code. For example, Roy could have defined a new, separate class to hold the functionality needed by all three classes (*Fax*, *Form* and *Mail*), and each of them could have inherited directly from that class.

Roy suggests a solution on the code level. Specifically, he suggests duplicating code; a solution that poses problems and risks of integrity later on, should this code be modified.

Participant's background: Roy holds a bachelor's degree in computer science and has 7 years of experience.

Analysis: Fleury [2] found that students tend to prefer code duplication over class reuse. Students find code duplication to be a simple solution that is easy to execute as well as to understand when reading a computer program. Fleury claims that this stems from the fact that it is easier for students to read the code in a sequence following a certain scenario rather than to look for a code section located elsewhere in the program each time. The origin of this phenomenon, like other phenomena Fleury identified, is the tendency to read code line by line while trying to understand it in the context of a concrete example. This tendency, however, leads to an understanding of the program only at a very low abstraction level. The problem of not reusing is not unique to students only. Hatton [1] noted that, other than for GUI building, reuse is not an outstanding success in OO (specifically C++), despite the fact that it is the second most common reason developers switch to that paradigm in the first place.

Roy seems to be thinking on a low abstraction level as well. Instead of solving the problem on the design level – a more abstract conception level, which is also the level in which the class discussion was conducted – he suggests a solution on the more concrete code level. Much of the research found that there is a tendency, especially by students and novices, to think on a low abstraction level [2,66], while experts are able to think in a more abstract manner [67,68].

Although with 7 years of experience Roy is not a novice, he demonstrates a similar phenomenon. It seems that when certain problematic issues are involved, even experienced developers retreat to the novice way of thinking on a lower abstraction level than is appropriate. This may indicate that Roy encounters difficulty solving the problem even when he tries to find the solution using S2 thinking. Since he cannot find the proper tool for solving this problem there, he shifts his line of thinking to the code level, which is the more natural abstraction level for him, and solves the problem there. He himself is aware of the problems this code duplication solution presents, yet finds no other way to solve the problem.

5.1.7. The challenge of abstraction

Abstraction is one of the key principles in computer science in general and in the OO approach in particular [69,70]. Thinking on high abstraction levels is, however, a difficult skill to acquire [69,70]. Thinking and solving problems on a lower abstraction level than is appropriate in a specific situation is a well known and documented phenomenon in education and is very common for students and novices [2,37,66]. The subjects in this research are experienced software designers and so it could be argued that they are expected to be able to think at a high abstraction level. Their experience and expertise must, however, be examined in the relevant context. For example, although the research population includes different levels of experience in OOA/D, their work deals with writing code as well, possibly leading to a tendency to think on the concrete level of code even when they are required to think abstractly. However, design requires that one avoids unnecessary implementation constraints [69].

There are many facets to thinking on a low abstraction level. In the context of OOA/D, this may lead to phenomena such as thinking about code execution too early when performing design or analysis tasks, looking for a local rather than system-wide solution, handling concrete objects rather than classes and in particular abstract classes, etc. The tendency of software designers to sometimes think on a lower abstraction level than is appropriate for solving a specific problem leads to difficulties in the decision-making process. Facing these difficulties, designers search for the appropriate thinking tool or heuristic using the S1 mechanism based on surface clues of the problem at hand. These clues, which are often not necessarily the important, significant factors of the problem, trigger the wrong heuristic, resulting in a non-normative solution.

Kramer [69] suggests that abstraction is a teachable skill, although it must be taught indirectly through other topics. He believes that abstract skills can be acquired and enhanced by means of teaching mathematics and formal modeling.

Good abstraction skills may reduce the tendency to search for S1 heuristics based on surface clues and thus activate an inappropriate decision-making process.

5.2. Discussion

S1 of experienced software designers encompasses two types of tools that may affect their decision making in OOA/D. One type consists of OOA/D thinking tools that are originally based on OO principles and know-how. Such tools are acquired in a long learning process and are established first as S2 tools. Following vast experience in OOA/D, these tools are transferred to S1 as well, becoming part of the developer's intuitive thinking, similar to driving, for experienced drivers. The tools, which are now part of S1, enable designers to solve problems and make decisions that are consistent with OOA/D principles, but without need for the long, tedious, deductive decision-making process that uses S2. Since this research focuses on difficulties rather than on successful performance, these types of S1 tools are not dealt with here.

Another type of tools that naturally exist in people's S1 are not related to the OO approach and yet often intervene with, and influence OOA/D decision making. Such tools were identified in this research due to their negative influence on the design's decision-making process. The importance of identifying these tools and analyzing their influence on OOA/D decision making stems from the fact that they may be part of the source of difficulty in understanding and performing OOA/D.

Examples were presented above in which arguably S1 influenced the thinking and decision-making process during OOA/D activities and led to unwanted results. Each example was analyzed to identify the S1 tool activated and the reasons for this activation. The dual-process theory helped by shedding light on the theory formed in this research — that software developers make non-normative decisions that contradict knowledge that they evidently hold.

The general phenomenon identified in this research may vary in its severity and frequency according to the interrelations between S1 and S2. The question here is not why does S1 jump in with the wrong answer, but rather why does S2 stay dormant instead of correcting S1's decision? There may be several different reasons for this, each of which leads in turn to a different behavior when confronted with the contradiction.

One such phenomenon is exhibited when S1 hijacks the thinking process and provides an answer that S2 has no reason to doubt, since it seems more or less correct, unless it is confronted by an external source. For example, the inheritance direction confusion was observed quite frequently, but it was typically corrected immediately when someone else implied gently, without adding any information, that there may be a problem. This shows that although S2 knowledge was firm, S1's first reaction was convincing enough not to mobilize it.

Another phenomenon is seen when the knowledge is available in S2 but the individual does not have sufficient understanding or practical experience to mobilize it. When encountering difficulty in solving a given problem, solvers often tend to fish for appropriate tools based on clues that appear in the data [11]. This sometimes leads to attribute excessive weight to certain parameters of the problem over other, more significant ones [8]. In this research this was illustrated, for example, by the avoidance of using abstract classes. While subjects were familiar with the notion of abstract class and its role, they made significant efforts or found solutions they themselves understood were problematic, such as code duplication, and did not mobilize this knowledge at any stage. This means that S2 knowledge was not firm enough to correct S1's decision, even though S2 was triggered. By reinforcing the knowledge and tools in S2, this phenomenon can be at least partly overcome.

Some of the cognitive difficulties identified in this research have already been documented in the literature; for example, the difficulty to define objects [3,4,31,34,37] and specifically, to differentiate between declarative and procedural aspects of the solution when doing so [31], or difficulties related to inheritance and abstraction via classes, classes' hierarchy and abstract classes [1,3,21,27,28]. The main contribution of the findings of the current research is twofold:

(1) Most of the prior research in which difficulties related to OOA/D were identified focused on novices; this research focuses on software designers in industry with varying degrees of experience. This population is expected to have firmer knowledge of OOA/D concepts and principles. Identifying that they, too, experience difficulties in basic topics of OO illustrates all the more the inherent conflicts between the OO approach and natural thinking.

(2) This research identified the conflict between knowledge subjects hold and non-normative decisions they make. This led to the dual-process theory as an explanation for several difficulties practitioners (as well as novices) encounter, thus not only providing a possible explanation of the source of these difficulties, but also indicating directions to developing remedies.

In addition to the above cognitive factors, other factors may also reinforce the tendency to make S1 decisions without S2 overriding them, such as affective (emotional) and external factors. Affective factors may include the desire to solve problems while investing little effort [65] or to quickly gain the satisfaction of running the code. Another affective factor may be the hesitation to deal with a certain issue, as witnessed in the case of multiple inheritance. Roy was so disturbed by the idea that he blocked out not only this solution, but any solution on the design level. This implies that his hesitation regarding this topic may have influenced the accessibility of the relevant knowledge required. In turn, this lack of accessibility may result in an allocation mistake [8], which leads to decisions based on inappropriate, but more accessible thinking tools.

External causes, originating in the environment of the software developer, may also affect the interaction between S1 and S2 during software developers' work in organizations that require the developed software as quickly as possible. Such factors were not directly identified in the workshops, since the workshops were disconnected from the developers' regular working routines. Frequent statements made by the workshop participants, however, allude to time pressures they experience at work. For example, in an interview with the workshops' instructor, he noted: "They will always say: 'Listen, the boss is pressuring us, the deadline was a week ago. We don't have time to design, we must write the code'". Another experienced software designer who worked in several different companies over the years, referred in his interview to the pressures in the industry: "It is better to make a decision that is 80% correct than not to make a decision or to delay it for a long time". Other statements made by the participants supported the impression that designers do not always have the time they need to design the system properly. This kind of time pressure contributes to the tendency to accept decisions made by S1, without involving S2 that requires the investment of more time and effort.

In summary, several factors influence the tendency to make decisions using S1, while S2 remains dormant even though it holds knowledge to the contrary of the decision made, including cognitive, affective and external factors as discussed above. The solution for these situations is far from trivial and is beyond the scope of this paper. It is reasonable, however, to speculate, based on the data obtained in the current research, that reinforcing the knowledge and thinking tools in S2, using analysis and design methods and tools that support thinking on high abstraction levels and the commitment of organizations that encourage and strictly control design quality, may improve the decision-making processes of software designers. Moreover, bringing this phenomenon to the attention of software designers or, during education programs, to the attention of software designers to be, may increase the alertness of S2, thus increasing the likelihood of S2 overriding non-normative answers provided by S1.

5.3. Threats to validity — evaluating grounded theory

Qualitative researchers are usually not comfortable using the terms 'validity' and 'reliability' for the assessment of their research; these terms carry too many quantitative implications [71,72]. It is important to select assessment criteria that are specifically appropriate for a given qualitative methodology [72,44].

Following the work of Abraham et al. [44], who applied the grounded theory approach while analyzing the study's findings through the lens of psychology, the outcome of this work is assessed using Glaser and Strauss's [73] recommendations. Accordingly, the criteria of credibility and applicability are discussed below.

5.3.1. Credibility

Is there sufficient detail and description so that readers feel that they were vicariously in the field?

While remaining within the scope expected from a journal article, this paper tried to provide sufficient details about solutions and decision-making processes of the subjects, as were collected by the researcher. The paper includes several examples of solutions, discussions and interviews in order to illustrate the decision-making processes as well as the contexts in which the subjects conducted their tasks.

Is there sufficient evidence on how the data were gathered and how the analysis was conducted?

The Research Method and Settings section provides details about how the setting of the study and its participants were selected, so that the theoretical sampling principles were met. Additionally, the data collection tools and procedures were described in detail. The main limitation when studying cognitive processes, in general, is the lack of direct means of "recording" these processes as they actually take place in people's minds. Only the subjects' reflection on their thinking processes, whether concurrent or in retrospect, in addition to written or verbal solutions are available to the researcher. Moreover, S1 processes are automatic and immediate; a trigger for allocating a specific heuristic to solve a certain task takes place unconsciously [8], thus the subject is not able to report it. As a result, the sources of the difficulties leading to an inappropriate selection of heuristic by S1, such as the salience of specific terms (see 5.1.1), can be proposed and ratified via additional data, as was done in this study, but an irrefutable proof is not available. Triangulation, by way of different data collection techniques and time periods, provided a means to partly mitigate this limitation and further validate the collected data.

The data analysis efforts are also described in the Research Method section, that is, all three types of coding (open, axial, and selective) are detailed.

5.3.2. Applicability

Is the theory readily understandable by laymen and professionals?

At the completion of the research, the results as well as early versions of this paper were presented to several software engineers, and feedbacks showed that they understood the described situations, including solution examples and their contexts, the dual-process theory as elaborated in this paper, as well as the data analysis process and its conclusions. In order to enable the reading of this paper to people who are not experts in OO technology, basic principles are briefly explained within each example presented in the paper.

This paper emphasizes the importance of including an additional lens, based on cognitive psychology, to explain errors professional software engineers make while practicing OOA/D. The paper describes the relevance of the findings to professionals, educators and researchers.

Is the theory sufficiently general to be applicable to diverse situations and populations?

The sampling of the research participants aimed to allow the generalization of the conclusions to diverse populations. Accordingly, OO designers from different firms and domains, from different education institutes and tracks and with a range of experience and seniority were selected to participate in the study. The main limitation of this sampling was that all participants were from the same country. However, as this research analyzes psychological mechanisms which are inherent to all human beings, this limitation is somewhat mitigated. A follow-up research in additional countries would enable safer generalization of the conclusions of this research.

Another limitation concerning the generalizability of the results stems from the fact that this was an exploratory study. The phenomenon identified, namely that software designers make non-normative decisions that contradict knowledge they hold, and the categories identified as possible sources leading to these non-normative decisions, can now be studied using quantitative research methods to further establish these findings and to determine their prevalence in the decisions of software designers in industry.

The situation in which most data of this research were collected was within a workshop of advanced OOD. This setting is different from real-life software development practice, presenting less time pressure and smaller, less complex tasks. While this can be considered a limitation, in the context of the findings of this study fewer mistakes or hasty non-normative decisions are expected to take place in this setting than in the real-life situations of tight schedules and more complex tasks. Thus, the difficulties identified in this study are expected to be even more prominent in practice.

Does the theory provide the user with sufficient control to bring about change in situations?

Research works dealing with the dual-process theory and cognitive biases are typically descriptive rather than prescriptive. Overcoming the phenomenon of S1 highjacking the thinking process and leading to intuitive decisions that may not be appropriate in formal situations is a problem yet to be resolved. As a first step, the previous subsection and the next section suggest possible directions in which awareness to this phenomenon and mitigation techniques may reduce these types of errors and improve practitioners' performance. Future research may further develop possible mitigation techniques and examine their actual effect.

6. Conclusion and future work

This research set out to identify difficulties encountered by software design practitioners in the use and understanding of concepts and principles of the OO paradigm. The findings show that software designers experience difficulties in the most basic concepts and principles of OO. Moreover, the study reveals that software designers make non-normative decisions that are inconsistent with knowledge they evidently hold, possibly due to various surface clues that misguide their thought process. These misleading surface clues include salience of a certain parameter of the problem that outweighs other, more important parameters; use of metaphors, which may be misleading; personal experience or knowledge that brings irrelevant data to the new situation; and natural thinking, which is at times more concrete than the abstract level needed in OO development in general, and in OO analysis and design in particular. In order to shed light on these phenomena the dual-process theory from cognitive psychology was used.

Several possible implications stem from the findings of this research. Understanding the cognitive mechanism behind the identified difficulties can be helpful for improving education as well as state-of-the-art programming paradigms. Educational programs typically focus on providing knowledge and skills in a specific domain, in our case OOA/D. However, as shown in this and previous research, this knowledge is not always mobilized. In order to increase the accessibility to logical thinking (S2) in cases when it clashes with intuition (S1), these phenomena should be constantly brought to the awareness of OO students and practitioners. Such awareness may increase the alertness of S2. In addition, techniques to overcome these dangers should be developed and taught. One such potentially contributing technique, for example, is solving problems in pairs (similarly to pair programming, however in OOA/D tasks). Discussing a suggested solution with someone else increases the likelihood of the use of a reasoning mechanism (S2). Although not ensuring normative decisions (as illustrated in some of the examples in this paper), this, together with firmer knowledge in S2, will improve the OOA/D decision making at least in part of the cases. Some techniques for alerting S2 may also possibly be incorporated in modeling tools. Based on the tendency identified in this research to confuse inheritance direction, for example, modeling tools may be enhanced with automatic tests to identify such possible design problem. A further implication is the relevance of the research findings to researchers and developers of evolving or new programming paradigms. For example, the use of specific metaphors may be reconsidered or chosen more carefully in the future when developing new programming paradigms, taking into consideration their full original meaning in everyday life.

Two main directions lead to future research: One direction is to facilitate further validation and generalization of the findings of this research using quantitative research methods. Such research may also indicate how prevalent these phenomena are in real-world settings. Another research direction is to empirically explore whether and how these phenomena can be overcome using educational and methodological means such as bringing the dual-process theory and its challenges to software designers' awareness, providing methods and tools for making design decisions, and guiding the next generations of OO languages and modeling tools to overcome these intuitive-based challenges.

A good opportunity for quantitative research to further investigate these problems, as well as develop and evaluate possible solutions, lies in educational programs in academia and industry. In these teaching settings, educators can evaluate how prevalent these phenomena are via a preliminary test including design tasks similar to the ones presented here, and assessing whether, and how frequently, errors similar to the ones presented here (e.g., reverse inheritance direction, defining salient terms from the requirements description as objects, etc.) occur. Following this test, a single suggestion for remedy may be applied at a time (e.g. pair designing for the experiment group vs. individual designing in the control group) followed by a post-test for examining whether it led to a reduction of these phenomena. A series of such studies will hopefully find a variety of means to improve the alertness of S2 during OOA/D, which will lead to the reduction of non-normative OOA/D decisions.

Acknowledgments

I am grateful to Uri Leron for his essential guidance and help in carrying out the research and in writing the paper.

Appendix. The outline of the workshop

Advanced object-oriented design workshop

Object Oriented Analysis and Design using UML presents the concepts and techniques necessary for effective implementation of system requirements. In this intensive, hands-on workshop, we will focus on the object-oriented design aspects using Unified Modeling Language (UML), evolving from fundamental OO concepts to advanced design constructions.

Objectives

Upon completion of the course, participants should be able to:

- Apply an iterative, use case-driven, architecture-centric process to the development of a robust design model.
- Use UML to represent the design model.
- Apply the concepts of abstraction, encapsulation, inheritance and polymorphism.
- Understand the different views of software design, the key mechanisms that are defined in support of that design, and the effect of the design on the produced system.
- Describe basic design considerations, including the use of patterns.

Topics Covered

- Advanced Object-Orientation and Design Patterns
- System functional design — Collaboration, Sequence and State Charts diagrams
- System structural design — Class and Component diagrams
- System distribution alternatives — Deployment, Packaging, Installation

References

[1] L. Hatton, Does OO sync with how we think? IEEE Software 15 (3) (1998) 46–54.
[2] A.E. Fleury, Encapsulation and reuse as viewed by Java students, SIGCSE Bulletin 33 (1) (2001) 189–194.
[3] D. Svetinovic, D.M. Berry, M.W. Godfrey, Concept identification in object-oriented domain analysis: Why some students just don't get it, in: Proceedings of the 13th IEEE International Conference on Requirements Engineering RE'05, 2005, pp. 189–198.
[4] D. Svetinovic, D.M. Berry, M.W. Godfrey, Increasing quality of conceptual models: is object-oriented analysis that simple? in: Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering, Shanghai, China, 2006, pp. 19–22.
[5] D.J. Armstrong, B.C. Hardgrave, Understanding mindshift learning: the transition to object-oriented development, MIS Quarterly 31 (3) (2007) 453–474.
[6] H.J. Nelson, D.J. Armstrong, K.M. Nelson, Patterns of transition: the shift from traditional to object-oriented development, Journal of Management Information Systems 25 (4) (2009) 273–300.
[7] I. Hadar, U. Leron, How intuitive is object oriented design? Communications of the ACM 51 (5) (2008) 41–46.
[8] D. Kahneman, Nobel prize lecture: maps of bounded rationality: a perspective on intuitive judgment and choice, in: T. Frangsmyr (Ed.), Les Prix Nobel, 2002, pp. 416–499. Retrieved from http://www.nobel.se/economics/laureates/2002/kahnemann-lecture.pdf.
[9] U. Leron, Origins of mathematical thinking: a synthesis, CERME3 (2003) Bellaria, Italy.
[10] U. Leron, Mathematical thinking & human nature: consonance & conflict, in: Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Education, PME, Bergen, July 2004, pp. 217–224.
[11] U. Leron, O. Hazzan, The rationality debate: application of cognitive psychology to mathematics education, Educational Studies in Mathematics 62 (2) (2006) 105–126.
[12] G. Booch, Object-oriented Analysis and Design with Applications, second ed., Addison Wesley Longman, 1994.
[13] M.E. Sime, T.R.G. Green, D.J. Guest, Psychological evaluation of two conditional constructions used in computer languages, International Journal of Man–Machine Studies 5 (1) (1973) 105–113.
[14] E.A. Youngs, Human errors in programming, International Journal of Man–Machine Studies 6 (3) (1974) 361–376.
[15] R. Brooks, Towards a theory of the cognitive processes in computer programming, International Journal of Man–Machine Studies 9 (6) (1977) 737–751.
[16] B. Shneiderman, R. Mayer, Syntactic/semantic interactions in programmer behavior: a model and experimental results, International Journal of Computer and Information Sciences 8 (3) (1979) 219–238.
[17] A. Malhotra, J.C. Thomas, J.M. Carroll, L.A. Miller, Cognitive processes in design, International Journal of Man–Machine Studies 12 (2) (1980) 119–140.
[18] R. Jeffries, A.A. Turner, P.G. Polson, M.E. Atwood, The processes involved in designing software, in: J.R. Anderson (Ed.), Cognitive Skills and Their Acquisition, Lawrence Erlbaum Associates Inc, Hillsdale, NJ, 1981, pp. 255–283.
[19] E. Kant, A. Newell, Problem solving techniques for the design of algorithms, Information Processing and Management 28 (1984) 97–118.
[20] R. Guindon, H. Krasner, B. Curtis, Cognitive processes in software design: activities in early, upstream design, in: H.J. Bullinger, B. Shackel (Eds.), INTERACT'87, 1987, Amsterdam: North Holland, pp. 383–388.
[21] M.B. Rosson, S.R. Alpert, The cognitive consequences of object-oriented design, Human-Computer Interaction 5 (4) (1990) 345–379.
[22] M.B. Rosson, J.M. Carroll, Climbing the smalltalk mountain, SIGCHI Bulletin 21 (3) (1990) 76–79.
[23] I. Vessey, S. Conger, Requirements specification: learning object, process, and data methodologies, Communications of the ACM 37 (5) (1994) 102–113.
[24] M. Carwright, An empirical view of inheritance, Information and Software Technology 40 (14) (1988) 795–799.
[25] D.J. Armstrong, H.J. Nelson, K.M. Nelson, V.K. Narayanan, Building the IT workforce of the future: the demand for more complex, abstract, and strategic knowledge, Information Resources Management Journal 21 (2) (2008) 63–79.
[26] D.P. Tegarden, S.D. Sheetz, Cognitive activities in OO development, International Journal of Human-Computer Studies 54 (6) (2001) 779–798.

[27] S.D. Sheetz, D.P. Tegarden, Illustrating the cognitive consequences of object oriented system development, Journal of Systems and Software 59 (2) (2001) 163–179.
[28] G. White, M. Sivitanides, Cognitive differences between procedural programming and object oriented programming, Journal of Information Technology and Management 6 (4) (2005) 333–350.
[29] H.F. Ledgard, The emperor with no clothes: examining the software problem from a scientific standpoint, Communication of the ACM 44 (10) (2001) 126–128.
[30] D.J. Armstrong, The quarks of object-oriented development, Communications of the ACM 49 (2) (2006) 123–128.
[31] F. Détienne, Assessing the cognitive consequences of the object-oriented approach, Interacting with Computers 9 (1) (1997) 47–72.
[32] J. Daly, A. Brooks, J. Miller, M. Roper, M. Wood, Evaluating the effect of inheritance on the maintainability of object-oriented software, in: W. Gray, D.A. Boehm-Davis (Eds.), Empirical Studies of Programmers, Sixth Workshop. Alexandria, Virginia, US, 5–7 January 1996, pp. 39–58.
[33] B. Meyer, Object-oriented Software Construction, Prentice Hall, 1988.
[34] J. Baragry, Do we think in terms of objects and what are the consequences for software reuse, architecture and patterns? (2001). Retrieved from http://publications.nr.no/roots01.pdf (accessed 07.12.2011).
[35] J. Evermann, Y. Wand, Ontology based object-oriented domain modeling: fundamental concepts, Requirements Engineering 10 (2005) 146–160.
[36] G. Booch, Object-oriented design, Ada Letters I (3) (1982) 64–76.
[37] R. Or-Bach, I. Lavy, Cognitive activities of abstraction in object orientation: an empirical study, SIGCSE Bulletin 36 (2) (2004) 82–86.
[38] E. Rosch, Principles of categorization, in: E. Rosch, B.B. Lloyd (Eds.), Cognition and Categorization, Lawrence Erlbaum Associates, Inc, Hillsdale, NJ, 1978, pp. 24–48.
[39] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on Software Engineering 26 (8) (2000) 786–796.
[40] M. Bassey, Methods of enquiry and the conduct of case study research, in: Case Study Research in Educational Settings, Open University Press, UK, 1999, pp. 65–91.
[41] A. Strauss, J. Corbin, Basics of Qualitative Research, Sage, Newbury Park, 1990.
[42] A. Strauss, J. Corbin, Grounded theory methodology: an overview, in: N.K. Denzin, Y.S. Lincoln (Eds.), Handbook of Qualitative Research, Sage, Thousand Oaks, 1994, pp. 273–285.
[43] R. Suddaby, From the editors: what grounded theory is not, Academy of Management Journal 49 (4) (2006) 633–642.
[44] C. Abraham, M.C. Boudreau, I. Junglas, R. Watson, Enriching our theoretical repertoire: the role of evolutionary psychology in technology acceptance, European Journal of Information Systems (2011) http://dx.doi.org/10.1057/ejis.2011.25. advance online publication 21 June 2011.
[45] I.E. Seidman, Interviewing as Qualitative Research, Teachers College Press, New York, 1991.
[46] R. Bogdan, S.K. Biklen, Qualitative Research for Education: An Introduction to Theory and Methods, second ed., Toutledge, London, 1992.
[47] A. Strauss, J. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage Publications, 1998.
[48] A.K. Barbey, S.A. Sloman, Base-rate respect: from ecological rationality to dual processes, Behavioral and Brain Sciences 30 (2007) 241–297.
[49] P. Wason, Reasoning, in: B.M. Foss (Ed.), New Horizons in Psychology, Penguin, 1996.
[50] T. Gilovich, D. Griffin, D. Kahneman (Eds.), Heuristics and Biases: The Psychology of Intuitive Judgment, Cambridge University Press, 2002.
[51] R. Samuels, S. Stich, P.D. Tremoulet, Rethinking rationality: from bleak implications to Darwinian modules, in: E. LePore, Z. Pylyshyn (Eds.), What is Cognitive Science? Blackwell, 1999, pp. 21–39.
[52] L. Cosmides, J. Tooby, Are humans good intuitive statisticians after all? Rethinking some conclusions from the literature on judgment under uncertainty, Cognition 58 (1) (1966) 1–73.
[53] G. Gigerenzer, P.M. Todd, Simple Heuristics that Make Us Smart, Oxford University Press, 1999.
[54] K.E. Stanovich, R.F. West, Individual differences in reasoning: implications for the rationality debate, Behavioral and Brain Sciences 23 (5) (2000) 645–726.
[55] A. Tversky, D. Kahneman, Judgments of and by representativeness, in: D. Kahneman, P. Slovic, A. Tversky (Eds.), Judgment Under Uncertainty: Heuristics and Biases, Cambridge University Press, Cambridge, UK, 1982.
[56] A. Tversky, D. Kahneman, Extension versus intuitive reasoning: the conjunction fallacy in probability judgment, Psychological Review 90 (4) (1983) 293–315.
[57] J. Wiley, Expertise as mental set: the effects of domain knowledge in creative problem solving, Memory & Cognition 26 (4) (1998) 716–730.
[58] A. Burton-Jones, R. Weber, Understanding relationships with Attributes in entity-relationship diagrams, in: P. De, J. Degross (Eds.), Proceedings of the 20th International Conference of Information Systems, Atlanta, GA, 1999, pp. 214–228.
[59] V. Khatri, I. Vessey, V. Ramesh, P. Clay, S. Park, Understanding conceptual schemas: exploring the role of application and IS domain knowledge, Information Systems Research 17 (1) (2006) 81–99.
[60] M. Rosemann, I. Vessey, R. Weber, C. Raduescu, Aligning organizational requirements with enterprise systems capabilities: the role of domain-specific knowledge, in: Proceedings of the 13th Americas Conference on Information Systems, AMCIS 2007, Keystone, Colorado, August, 2007.
[61] B. Blum, A paradigm for the 1990s validated in the 1980s, in: Proceedings of the American Institute for Aeronautics and Astronautics AIAA Conference, 1989, pp. 502–511.
[62] O. Berge, R.E. Borge, A. Fjuk, J. Kaasbøll, T. Samuelsen, Learning object-oriented programming, presented at the Norsk Informatikkkonferanse (Norwegian Informatics Conference), 2003.
[63] U. Wilensky, Abstract meditations on the concrete and concrete implications for mathematics education, in: I. Harel, S. Papert (Eds.), Constructionism, Albex Publishing Corporation, 1991, pp. 101–108.
[64] J.M. Bieman, K. Byung-Kyoo, Measuring design-level cohesion, IEEE Transactions on Software Engineering 24 (2) (1988) 111–124.
[65] U. Leron, O. Hazzan, The world according to Johnny: a coping perspective in mathematics education, Educational Studies in Mathematics 32 (3) (1997) 265–292.
[66] D. Aharoni, U. Leron, Abstraction is hard in computer-science too, in: E. Pehkonen (Ed.), Proceedings of the 21st Conference of the International Group for the Psychology of Mathematics Education (PME), Lahti, Finland: University of Helsinki, 1997.
[67] A. Von Mayrhauser, A.M. Vans, Identification of dynamic comprehension processes during large scale maintenance, IEEE Transactions on Software Engineering 22 (6) (1996) 424–437.
[68] M. Petre, A.F. Blackwell, Mental imagery in program design and visual programming, International Journal of Human-Computer Studies 51 (1) (1999) 7–30.
[69] J. Kramer, Is abstraction the key to computing? Communications of the ACM 50 (4) (2007) 37–42.
[70] O. Hazzan, The reflective practitioner perspective in software engineering education, The Journal of Systems and Software 63 (3) (2002) 161–171.
[71] M. Leininger, Evaluation criteria and critique of qualitative research studies, in: J.M. Morse (Ed.), Critical Issues in Qualitative Research Methods, Sage Publications, Thousand Oaks, 1994, pp. 95–115.
[72] J. Corbin, A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage Publications, Thousand Oaks, CA, 2008.
[73] B. Glaser, A. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research, Aldine Publishing Company, New York, 1967.