International Journal of Engineering Education Vol. 32, No. 2(B), pp. 937–947, 2016
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2016 TEMPUS Publications.

Persistent Ideas in a Software Design Course: A Qualitative Case Study*

PAMELA FLORES and NELSON MEDINILLA
Department of Information Systems and Languages and Software Engineering, Universidad Politécnica de Madrid, Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain. E-mail: {pamela.flores, nelson}@fi.upm.es

SONIA PAMPLONA
Science and Engineering Department, Universidad a Distancia de Madrid, Carretera de La Coruña, Km. 38,500, Vía de Servicio, n^{o} 15, 28400 Collado Villalba, Madrid, Spain. E-mail: sonia.pamplona@udima.es

This study aims to discover what persistent ideas students have when designing software, and discusses possible relationships between them. The research was conducted through qualitative case study over an academic period with Master's degree students in a Software Design course. The ideas obtained as results were grouped in persistence levels: low, medium and high; additionally some ideas have been identified, that could be potentially persistent. The main contribution of this paper is focused on two aspects: (a) Software design education, which allows teachers to identify and address problems related to Software Design course; and (b) Professional impact in the industry, by warning the software industry about the main problems that students carry out, despite of the instruction.

Keywords: software design; persistent ideas; qualitative research; computer science education

1. Introduction

To increase the efficiency in software development is one of the main objectives of the software industry. This is achieved by applying the modularization criteria proposed by Parnas [1]: (a) managerial: separate groups would work on each module with little need for communication; (b) product flexibility: to make drastic changes to one module without a need to change others; (c) comprehensibility: to study the system one module at a time. To achieve these objectives is necessary to introduce the term decomposition, since this action allows us to better manage complexity [2]. However, to decompose at software level involves much more than just divide, therefore studying how to decompose during the design process is very important. Among decomposition alternatives are nearly decomposable systems, whose concept was introduced by Simon [3]. Other authors make use of the concepts of Simon, such as Booch who builds five attributes that define the complex systems [2]. Parnas, on the other hand, states that intelligent decompositions address properly the complexity of a software system [4].

One of the objectives in teaching the Software Design is learning nearly decomposable designs, having as a reference the Information Hiding Principle, introduced by Parnas [1]. However, the classroom experience and other preliminary studies [5] have led us to believe that students fail designing this way. It becomes necessary to know what persistent ideas students have with respect to the decomposition of software systems. We define persistent ideas like those ideas that students have about a particular topic and precondition the behavior of some activity, in this case, the activity of designing software. These ideas remain during the teaching and even can be maintained until after the end of the academic school period. Moreover, they can be preconceived or acquired during the instruction period.

The term "*persistent ideas*" defined for this work focuses on discovering and analyzing equivocal and wise ideas, expressed by the students. The definition of persistent ideas differs from misconceptions term, where misconceptions have generally been seen as mistakes that impede learning [6, 7] or student's ideas that are incompatible with currently accepted scientific knowledge [8]. The case study conducted in this work consists on a group of students in the Software Design course of the European Master in Software Engineering of the Escuela Técnica Superior de Ingenieros Informáticos (ETSIINF) at Universidad Politécnica de Madrid. The course focuses on teaching Software Design with object-oriented approach. The details on the content of the course can be seen in [9]. The contribution of this work is focused on two aspects:

1. Software Design Education. The persistent ideas found in this study will provide to the teachers the capacity to identify misconceptions, difficulties and trends that students have when designing software. Moreover, this knowledge will enable the teacher to be aware of the existence of these ideas and be ready to address them if necessary.

* Accepted 1 September 2015.

2. Professional impact in the industry. To warn software industry about the main problems that recent graduates and future professionals have when designing software.

This paper is organized as follows. Section 2 describes the related work. Section 3 presents the methodology, research questions, environment and indicators about research. Section 4 presents the persistence levels as results. Section 5 presents a discussion, regarding persistence ideas and their relationships. Finally, Section 6 concludes the paper and presents the future work.

2. Related work

In the literature, most of the works are related with misconceptions in software engineering, especially in the programming area. Nevertheless, these studies are in a different direction from our research due to the methodology used. For example, in [10] the authors present a work based on the research of misconceptions in algorithms and data structures through expert interviews and the analysis of 400 student's evaluations. The paper [11] presents a work, which is focused on discovering what do the students know or should know about object-oriented programming (OOP), including a cognitive point of view.

There are few studies aimed at studying the learning of the software design using qualitative methods. One of them is the work [12], which conducted a research on learning OOP from students during two academic years. The aim of the research was to see the impact of introducing a pedagogical approach called Object-First in the teaching process and identify misconceptions and difficulties around OOP. Our work also uses a qualitative methodology and is focused on discovering the persistent ideas manifested in a Software Design course, regardless of the pedagogical technique. It means that studies what students think at the beginning, during and at the end of the course.

Authors in [13] identify programming misconceptions. This work is part of the first stage of the construction of a Concept Inventory for Computing Fundamentals, inspired by the Force Concept Inventory in the field of teaching of physics [14]. Formal interviews were conducted with students in order to reveal the misconceptions and were analyzed qualitatively. The research takes as the only source of analysis the interview, while our work presented here tries to capture the dynamics of students through various resources obtained during the academic period.

Meanwhile [15] analyzes five programs sent as tasks to students in a course with the Object-First approach. This analysis found concrete evidence about learning OOP concepts and typical misconceptions previously identified in the literature of OOP. Our work presented in this paper, does not make a preliminary study of persistent ideas or concepts, rather, it seeks to discover these preconceptions through qualitative research.

Another work that fits within our research is related to "Misconceptions of Designing: A descriptive study" [16], which is focused on researching the concepts and beliefs that the students in first year have when designing. The study is conducted through an on-line descriptive survey to 520 students in the first year. The questions asked in the survey were focused on discovering what students understand or relate to the activity of design, not how they design or program. And here lies the difference with our work presented below.

Finally, Sudol and Jaspan [17] in their study construct a model of the misconceptions based on student's repeated responses and response time. This methodology was used to evaluate and compare the results between students of Computer Science and practitioners of highly regarded companies. In our study, the results correspond to the facts derived from students, which were triangulated, but were not evaluated or compared as in [17].

Our study differs from the above mentioned works due to its purely qualitative nature at a methodological level. In addition, we have studied the progress of students throughout the academic period, a research whose objectives go beyond analyzing a particular instance during the teaching. It was not considered to take ideas or misconceptions from other studies in order not to interfere in the analysis from the teacher and the researcher point of view. Finally, specific teaching approaches were not introduced at the curriculum level; neither the course objectives nor the teaching methods were modified.

3. Research methodology

This research uses a qualitative research methodology, including a case study. Qualitative methods help to understand and explain the meaning of social phenomena with the least possible modification of the natural environment in which they occur [18]. The case study is defined as the intensive research of a single object of social inquiry, for instance a classroom [19]. The study consisted of a set of observations, interviews and document analysis obtained during the academic period. The whole process was documented, discussed and analyzed between the researcher and the teacher. Experience in implementing this methodology has been acquired by the authors in previous studies [5, 20, 21].

The analysis process consists of four stages where three of them were based on Coding, which is one way of analyzing qualitative data. A *code* in a qualitative inquiry is defined most often as a word or short phrase that symbolically assigns a summative, salient, essence-capturing, and/or evocative attribute for a portion of language-based or visual data [22].

Figure 1 describes the actions taken at each stage and their respective results. *Noticing* and *Collecting* stages are performed iteratively several times until encountered codes got stabilized. The Super codes obtained in the *Grouping* stage come from the grouping of tighter codes in the Collecting stage. The grouping has been performed following the criteria of the research questions, shown in Section 3.1. The process in Fig. 1 was systematically performed for all qualitative data collected during the academic period. The treatment of the data was made with the help of the software Atlas.ti [23].

Furthermore, our study was aligned with quality criteria defined by Lincoln and Guba [24], around trustworthiness of a research, which is detailed below.

Consistency. The process of the study is presented in a way that allows traceability for audit.

Credibility. Techniques as *peer debriefing* and *triangulation* were applied throughout the research. Peer debriefing consisted of documented subsequent meetings (audio and text) between the teacher and the researcher after each activity, throughout the research process. Triangulation was performed for several iterations with validation codes as follows:

1. Codes extraction by assignment.
   After a systematic analysis and several iterations of the resources of each student (text, diagrams, interviews), we extracted the codes generated throughout the whole process for each particular task, as shown in Fig. 2.
2. Code verification by students.
   Subsequently, the generated codes in the previous step were placed on each student. Fig. 3 allowed us to analyze the evolution of codes for each student throughout the academic period.

3.1 Research questions

Regarding the problematic, we raise the following research questions:

- What ideas of Software Design persist throughout the academic period?
- Is there any relationship between these persistent ideas?

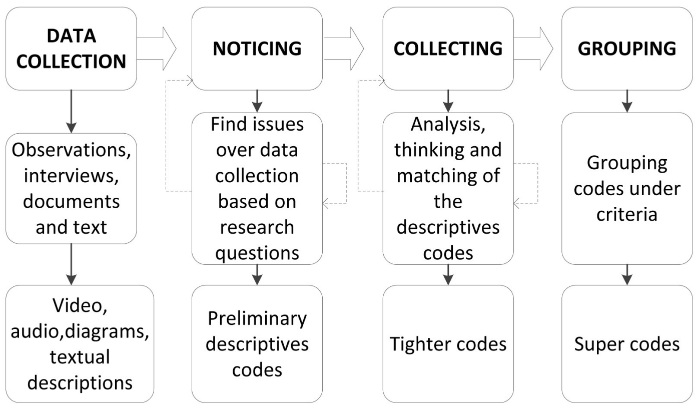
DATA COLLECTION
NOTICING
COLLECTING
GROUPING
Observations, interviews, documents and text
Find issues over data collection based on research questions
Analysis, thinking and matching of the descriptives codes
Grouping codes under criteria
Video, audio,diagrams, textual descriptions
Preliminary descriptives codes
Tighter codes
Super codes

Fig. 1. Analysis process.

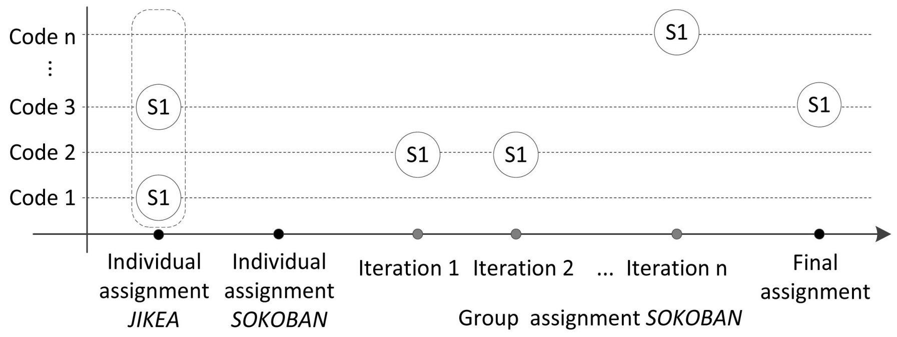

Fig. 2. Example of codes for one student (S1 = Student 1).

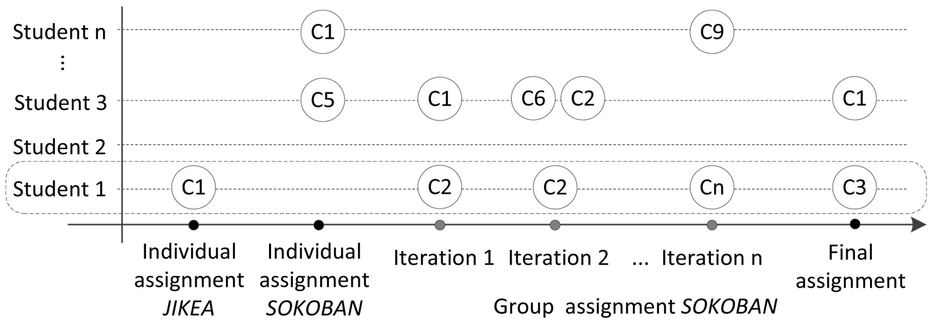

Fig. 3. Example of students by codes (Cn = Code n).

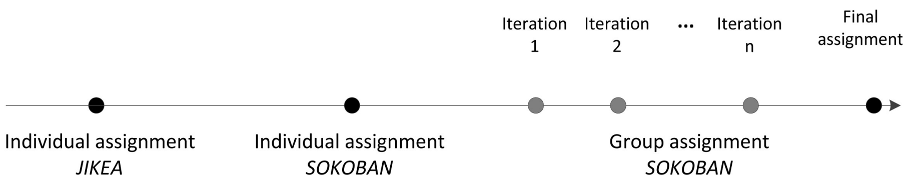

Fig. 4. Timeline of assignment during the academic period.

Table 1. Research indicators

Indicator	Detail	Description
Raw Data	Written assignments	34 documents ≈ 35000 words
	Audio from interviews	≈ 6 hours
	Audio from teacher	≈ 40 hours
	Audio from lessons	≈ 16 hours
Quotations	Text, translates, diagrams	197
Codes	Preliminary codes	259
	Tighter codes	61
	Super codes	34

3.2 Environment

The group is characterized by students of different nationalities, which have completed the first degree in Computer Science with different curricula. The number of students was thirteen out of seventeen. Four students were discarded due to instability issues during the scolar period. The case study was implemented in an academic semester period on the Software Design course. The curriculum of the course, which is focused on object-oriented design, was not modified by the research.

Three assignments were sent to students. Two of them were individual and the last one was in group. The group assignment consisted of additional iterations (versions) before the final assignment, which were also analyzed; the distribution of assignments can be seen in Fig. 4. In addition to the audio obtained through student's interviews about the assignments, we also analyzed tutorials with students and classroom observations. This allowed us to see the evolution of each student throughout the academic period. The assignments consisted in designing a small graphical application shown in classroom. The usage of a visual application exceeded clarity over written or verbal statements. This was proved in previous semesters, where written and verbal statements created ambiguity in students.

The description of the graphical applications can be seen below:

JIKEA: Small graphical application that consists in moving furniture in several rooms.
SOKOBAN: Is a type of transport puzzle, in which the player pushes boxes or crates inside a warehouse, trying to get them in different storage locations.

3.3 Indicators

The amount of data collected was massive. The summarized information about the research indicators is shown in Table 1.

4. Results

The results of the analysis of qualitative data have been classified in four persistence levels: low, medium, high and potentially persistent. In this work we focus in greater depth on the high persistent level because of the relevance for the study.

4.1 Low persistence

Low persistence are those ideas that disappeared over the academic year, but being present at some point in the timeline, could be a sign that existed before the research started. Within this level, we have found the following ideas:

- Separation in layers to hide information.
- Use of an element as a main program (`Main`).
- Identification of major functions and their subsequent distribution modules (In the style of Object Modeling Technique [25]).
- Separation of each concept into two elements, one for data and another for functions, to manage the persistence.
- Definition of classes as data.
- Inheritance used as taxonomy.
- Inheritance used to reuse code.
- Adding to a child an attribute that parent does not have.
- Separation of elements that are similar in software but are interpreted as different.
- Definition of methods with the same name that should have a similar behavior, but are totally different in their implementation.
- Definition of a concept through a property that is not related to the concept (Example: Definition of an `ObjectInterface` concept through a unique method `authorize()`).

4.2 Medium persistence

Medium persistence are those ideas that appeared sporadically in different tasks throughout the academic period. Within this level, we found the following ideas:

- Grant a high level of control to an element.
- Definition of an identifier (`ID`) as an attribute of each element.
- Simultaneous decomposition by concepts and properties (Example: Definition of a concept `Figure` like father and a son `Movable`, representing a property of the figures).
- Absence of the concept `Game` or insufficiency to define it in a game application.
- Absence of an element that acts as container or insufficiency to define it.
- Particularization of a method for the possible combinations (Example: `moveUp()`, `moveDown()`, `moveLeft()`, `moveRight()`).
- Particularization of the lists (Example: `listBox`, `listWall`, `listPlayer`).

4.3 High persistence

High persistence are those ideas that meet the following criteria: (a) appeared recurrently, it means they disappeared and reappeared on the timeline and; (b) appeared at the end of the timeline, it means that could be present throughout the

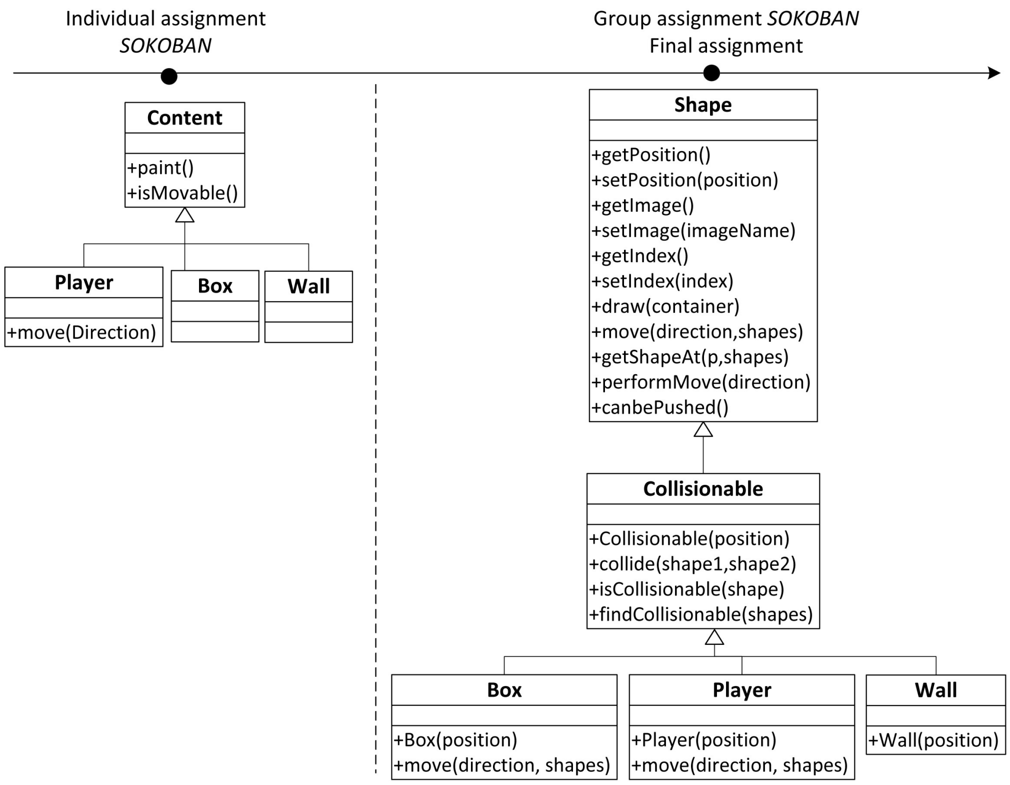

Fig. 5. Resistance to give properties to a concept that in real life does not have.

process but were not manifested before. Within this level, the following ideas are mentioned:

- Resistance to give properties to a concept that in real life does not have.

  The example shown below in Fig. 5 illustrates how students refuse to assign to the element Wall the property "move" and justify by saying that:

  "A player and a box can be moved. A wall is the only content type that is not movable".

  "The problem relies in the conceptual implication on the design. Walls cannot move in the real life".

- Preference to inherit a property rather than delegate it.

  To illustrate this example, we will use Fig. 6. As we can see on the left side of the figure, students define an element called `Collisionable`, which is delegated by the elements that use it. In the right side of the figure, the same students in a next iteration change the property `Collisionable` that before was delegated to be an element that inherits.

  Students justify the cohesive overload as follow:

  "In order to ease this decision, we decided to incorporate the `Collisionable` property by inheritance, so every figure could implicitly know if the figure next to it was collisionable or not".

- Cohesive overload of responsibilities in one element.

  We use the term cohesive when dealing with interrelated responsibilities. Fig. 7 describes an interesting case where the students try to distribute the task, but finally retake the idea of overloading. This is a clear example of the root of this idea. Fig. 7 shows different design iterations performed in group for the task *SOKOBAN*. In Iteration 1, there is an element `Collision` that is a container and responsible for handling collisions through `collide()` method. In Iteration 2, `Map` is a container and `collide()` maintains the collision management. In Iteration 3, students give a positive jump, by adding a `canOverlap()` method, which is responsible for authorizing the movement, avoiding dependencies on

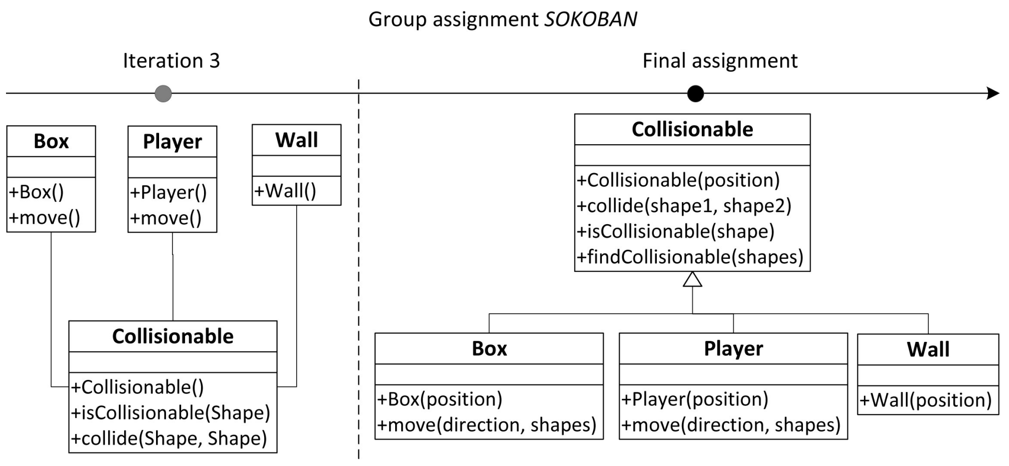

Fig. 6. Preference to inherit a property rather than delegate it.

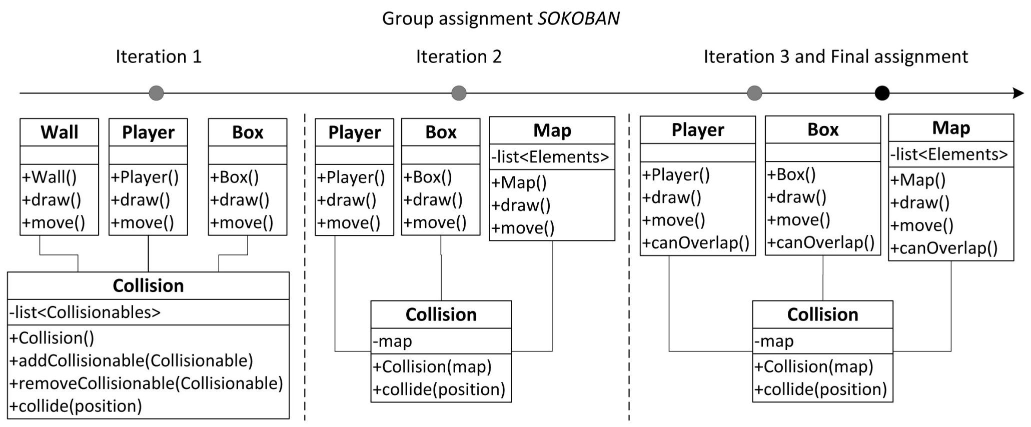

Fig. 7. Cohesive overload of responsibilities in one element.

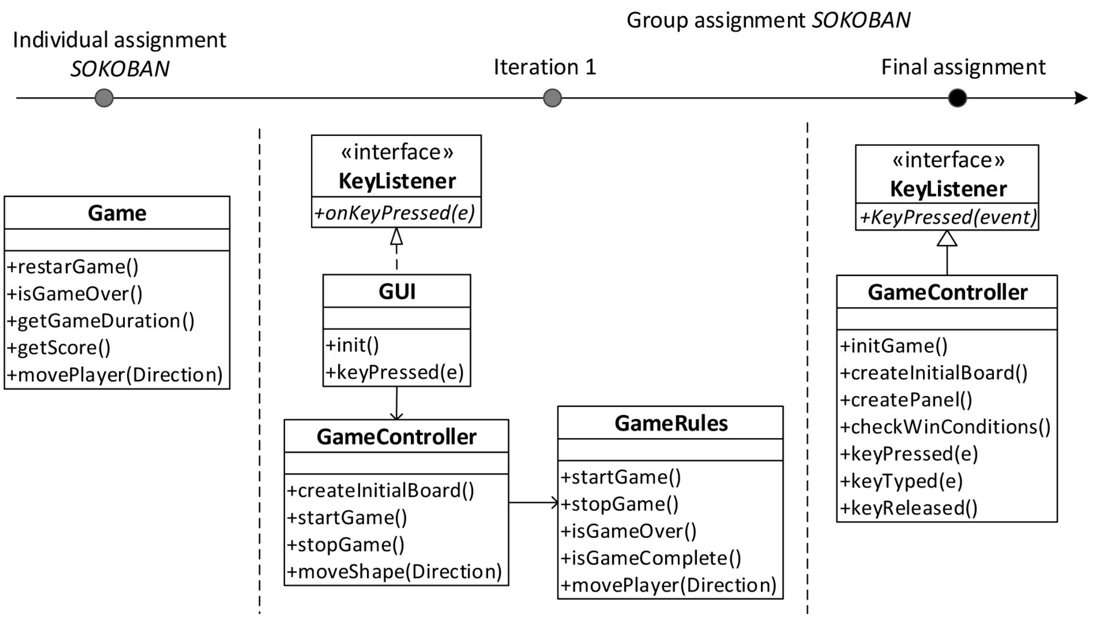

Fig. 8. Non cohesive overload of responsibilities in one element.

each element. Finally, in Iteration 4, the design remains the same as in Iteration 3, but the responsibilities as the same as in Iteration 2.

- Non cohesive overload of responsibilities in one element.

We use the term non cohesive when dealing with unrelated responsibilities between them. An interesting case of this idea is student's evolution in a timeline, which is shown in Fig. 8. In the left side of the figure, there is an element `Game`, responsible for three issues (initialize the board, win the game and hear the movement) for the task of *SOKOBAN*. In Iteration 1 of the group work, students show three elements, each one responsible for the tasks listed above. In the Final assignment, students merge again the elements into one, with a confusing use of the inheritance. Students justify the non cohesive overload as follows:

"*Fusioning of* `Controller` *and* `GameRules` *classes was done because* `Controller` *and* `GameRules` *finally had the same behavior*".

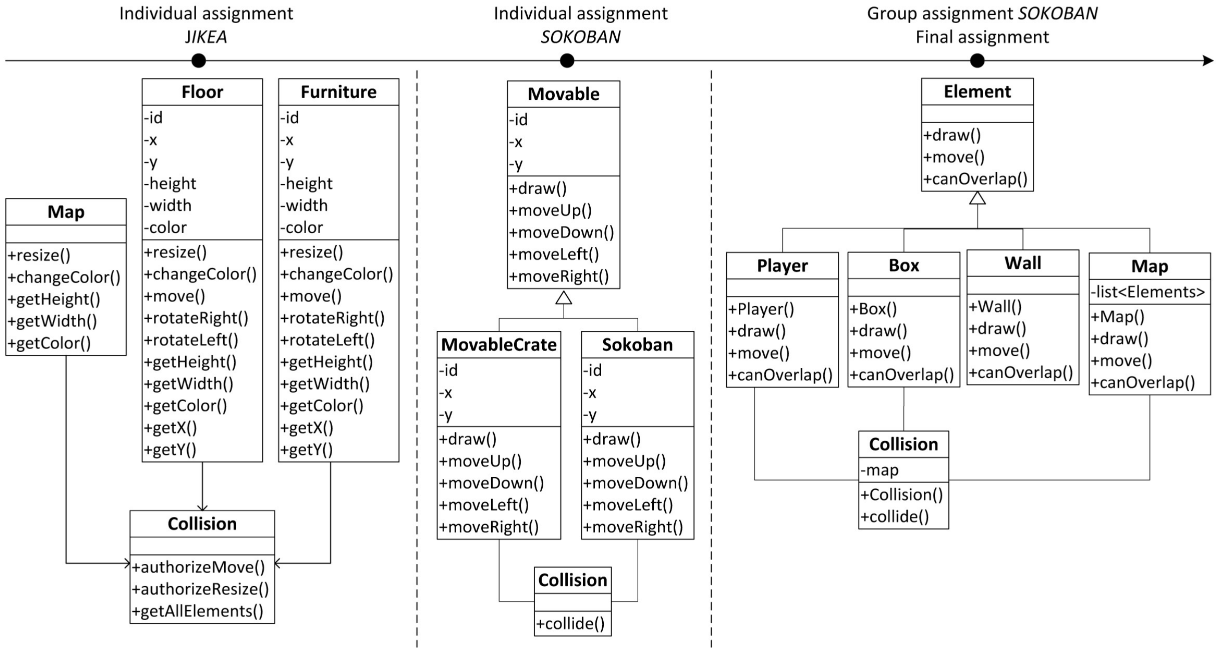

Fig. 9. Definition of a property that is invisible to the outside elements.

- Definition of a property that is invisible to the outside elements.

  An example of this idea is the evolution of a student with the individual and group assignments, which is shown in Fig. 9. In the figure, on the left side it is observed how Collision is defined as an element, which is used internally by different elements. In the central figure we can see two issues, an element Movable, explicit for the external elements, and an element Collision which is not. Finally, in the final iteration of the group assignment, predominates the use of Collision element as invisible for the inheritance.
- Information dependency from other elements to make decisions.

  This idea is manifested when one or more elements depend on information to execute an action or make a decision. It is characterized by asking or requesting information from the outside, sometimes in a univocal manner and other times in a chained manner.
- Partial use of inheritance with polymorphic purposes by adding in children particular methods that a parent does not have, and are accessed from outside.

  This idea appears in one inheritance structure, where the children are accessed from the outside through particular methods not present in the parent. It means that the polymorphism is violated with the addition of methods in a child and the contract with the parent is ignored.
- Definition in inheritance of empty children equal to their parent.

  This idea is maintained during the timeline. Students define an inheritance, where some child is empty. This means that a child will have the same behavior as the parent, which may be justified only by code reuse.
- Interface definition to uniform the diversity and hide information.

  This idea supports the uniformity through the creation of an abstract element as an interface, the same which covers all different elements of the application that enter within this new concept. This behavior may be part of the ideas that could have been acquired previously during the course, since it started with this idea and it has remained until the end.
- Not being explicit about how or where the lists are handled.

  This idea has been expressed in several ways throughout the academic period. It is closely related to the "*Absence of an element that acts as container*" that appears in the medium persistence category. It is evident when students place methods such as `getAllElements()` or `getFigures()`, without knowing where or how they handle the data structure corresponding to the lists.

4.4 Potentially persistent

Potentially persistent are those ideas which have been taken from textual phrases of students assignments or interviews. Nevertheless, they have not been reflected in the designs throughout the academic year, therefore we cannot know if they have been kept or disappeared. Within this level, there are the following ideas:

- A low dependency index is not having relationships between concrete classes and having a low number of relationships between classes.
- Confusion between encapsulation and hiding.
- The Information Hiding Principle was considered as: (a) the only known issue must be the operation that solves the problem and; (b) that parameters of each entity are hidden.
- The Information Hiding Principle is associated by not having direct relationships between entities.
- Using `get()` and `set()` to hide the internals details of the class.

5. Discussion

The ideas above suggest different relationships between them (the second research question) and also suggest common sources, some with long trajectory. The "*Preference to inherit a property rather than delegating*" was discussed in [25] twenty five years ago and remains active as detected in this study. One possible cause of this idea would be the combination of two factors. On the one hand, there is the tendency to reproduce the reality in the software structure, as it has been reported in several papers [5, 26]. On the other hand, there is the quality of software inheritance to express taxonomies [27]. From the software design point of view, the preference to inherit instead of delegating has two drawbacks: it establishes very strong dependencies and it can produce a very dangerous use of the inheritance [27].

The "*Overload of responsibilities in an element*" has also been cited in other works, for example in the case of objects [28] and in the case of the Data Flow Diagrams [29]. Assigning appropriate responsibilities to a software element is a difficult task. It became evident in the case of the "*Definition of a concept through a property that is not related to the concept*". In the present study, there have also been found elements with non cohesive responsibilities to the concept and elements with more responsibility than they should have had. Both overloads (cohesive and non cohesive) could have been related to the distribution and perception of complexity in the design. We can think that one element is simpler than two elements even if that single element is internally more complex. However, the overload may be linked to the absence of an element in order to share responsibilities, for example, "*Absence of a concept* Game *in a game application*", expressed in this work. The reason for this absence could be associated with the tendency to reproduce the reality that omits intangible concepts because they are not seen. In [13] students apply the real world semantics to declare variables. Non cohesive or disjointed overload could be a manifestation of the idea of giving a high level of control to an element. In any case, the overload harms the objectives of industrial software development.

The idea of "*Grant a high level of control to an element*" was not detected in the final exercises, for this reason it was classified as a medium persistence level. However, it could be concealed in a non cohesive overload, as mentioned above. The idea of giving a high level of control to an element could be a lag or reflection of the hierarchical organization of the modules in the Structured Design [30], supported by the idea of [3], but criticized by [31]. In the first assignment explicitly appeared the idea of Main, whereas later disappeared or got moderated. In this case study, we observed that the control level gradually decreases throughout the course. The elements with high level of control rely heavily on other elements, and are critical design points. In summary, they are harmful.

The "*Information dependency from other elements to make decisions*" is a strong idea of persistence. It is possibly related to the structured view of the software, which means functions that transform data. This view appeared notably at the beginning, for example with: "*Separation of each concept into two elements, one for data and another for functions*", the "*Definition of classes as data*" and "*Definition of an* `ID` *as an attribute of each element*". Later, the vision of separating data and functions got softened, whereas remained present the dependence on information of another element to make decision or execute something. The dependence became more evident in the elements of high level of control and then more subtle when the control level got reduced. The information dependence of another element may be further influenced by the reproduction of the reality. It is difficult to assign power of decision to a software element that "represents" a passive element of the reality.

Also, it is interesting that some students applied techniques to hide information by using interfaces or by trying not to share internal details of the class, in order to make designs of reduced dependency. However they finally failed to meet the objective. The idea that "*Use of* `get()` *and* `set()`" hides the internal details of the class, is another possible difficulty for achieving a software design with minimal dependency between elements. The greater the dependence is, the fewer conditions the software has in order to be developed industrially.

The difficulty of managing lists is a persistent problem that has been manifested in several nuances. For example, "*Not being explicit about how or where the lists are handled*", "*Absence of an element that acts as container or insufficiency to define it*". Possibly, the cause of this problem is the significant difference of the structured approach with respect to the object-oriented approach, where conceptually there is no data and all elements are variables. Recently, the work in [13] found out in their study the lack of foundations in understanding the object concept in students.

Around the inheritance and polymorphism, several ideas have been found: "*Inheritance used as taxonomy*", "*Inheritance used to reuse code*", "*Preference to inherit a property rather than delegating*", "*Definition in inheritance of empty children equal to their parent*", "*Partial use of inheritance with polymorphic purpose by adding in children particular methods that a parent does not have, and are accessed from outside*", "*Absence of an element that unifies diversity*", "*Simultaneous decomposition by concepts and properties*". They all reveal difficulties in understanding and applying inheritance, which is a means of decomposition. Inheritance and polymorphism do not exist in the structured approach, but add complexity to the object-oriented approach. In the ideas found, interesting conflicts are seen between individual (particularize) and uniform treatment of diversity.

In addition to the harmful ideas, there have also been found positive ideas: "*Interface definition to uniform the diversity and hide information*" and "*Definition of a property that is invisible to the outside elements*". Both lead to the same objective: to hide information, it means, to reduce the dependency of one element with respect to the other elements.

At the end of our study, some ideas have remained without determining their degree of persistence, because they have not been made explicit later and we cannot ensure they have disappeared. In the worst case, we have decided to classify them as potentially persistent. These ideas are primarily associated with the Information Hiding Principle [1] which has historically been a difficult technique for understanding and applying [5, 32, 33, 34]. Recently, the work [12] has detected misconception about the above mentioned Principle. There is also an idea related to the conventional concept of coupling, related with the inheritance, but not with the Principle. This idea implies that the dependency comes from the direct connections between concrete classes and the amount of relationships between classes.

Finally, the second research question (*Is there any relationship between persistent ideas founded?*) can be answered positively since there are several relationships between those ideas, which probably share common ideas and difficulties.

On the one hand there are the cultural aspects attached to the reality and to the concrete, that condition the vision of the person who designs. It means, to reproduce the reality which is perceived; omit what is not seen; particularize; classify everything; request information and decide instead of letting someone else decide. On the other hand, there are difficulties of instruction. It means the habit of the structured approach (functions that transform data); failure in understanding and applying Information Hiding Principle and failure in understanding and applying of the object-oriented approach.

To summarize, the specific ideas about software design that have emerged in this study are signs of a complex combination of the individual vision of the world and the effect of tutoring in particular.

6. Conclusions and future work

The research questions for this case study were answered. Persistent ideas found in this paper and their relationships represent a source of problems at an academical level, but also at a professional level. The contribution of this work has been focused on two aspects: the education about Software Design and the professional impact on the industry.

At an education level, the research done raises to the Software Design teacher the great challenge of trying to correct or prosecute persistent ideas that difficult the enforcement of the three modularization criteria (managerial, product flexibility and comprehensibility), which increase the efficiency at a software development level. Consequently, it will lead in improving the learning of the Software Design.

At a professional level, this research warns the software industry about the source of problems that students have despite overcoming a bachelor degree. This means that the persistent ideas are possibly transferred to industry. As part of the future work we have considered the research around the design of activities or assignments in order to detect, and correct persistent ideas that difficult the learning of the Software Design.

Acknowledgements—This work is supported by SENESCYT (Secretaría Nacional de Educación Superior, Ciencia, Tecnología e Innovación) of Ecuador.

References

1. D. Parnas. On the criteria to be used in decomposing systems into modules, *Commun. ACM*, **15**(12), 1972, pp. 1053–1058.
2. G. Booch, Object-oriented Analysis and Design with Applications (2nd Ed.). Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.
3. H. Simon, The architecture of complexity, In *Proceedings of the American Philosophical Society*, 1962, pp. 467–482.
4. D. Parnas, Software aspects of strategic defense systems, *Commun. ACM*, **28**(12), 1985, pp. 1326–1335.
5. P. Flores, N. Medinilla and S. Pamplona, What do software design students understand about information hiding?: A qualitative case study, In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pp. 61–70. ACM, 2014.
6. T. Sirkiä and J. Sorva, Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises, In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pp. 19–28. ACM, 2012.
7. J. Smith, A. Disessa and J. Roschelle, Misconceptions reconceived: A constructivist analysis of knowledge in transition, *The Journal of the Learning Sciences*, **3**(2), 1994, pp. 115–163.
8. J. Clement, D. Brown and A. Zietsman, Not all preconceptions are misconceptions: finding 'anchoring conceptions' for grounding instruction on students' intuitions, *International Journal of Science Education*, **11**(5), 1989, pp. 554–565.
9. Universidad Politécnica de Madrid, Degree Programme Structure, Available online at http://emse.fi.upm.es/en/estructura.html, 2015.
10. H. Danielsiek, W. Paul and J. Vahrenhold, Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pp. 21–26. ACM, 2012.
11. [11] P. Hubwieser and A. Mühling, What students (should) know about object oriented programming, In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pp. 77–84. ACM, 2011.
12. N. Ragonis and M. Ben-Ari, A long-term investigation of the comprehension of oop concepts by novices. 2005.
13. L. Kaczmarczyk, E. Petrick, P. East and G. Herman, Identifying student misconceptions of programming, In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 107–111. ACM, 2010.
14. D. Hestenes, M. Wells, and G. Swackhamer, Force concept inventory, *The Physics Teacher*, **30**(3), 1992, pp. 141–158.
15. K. Sanders and L. Thomas, Checklists for grading object-oriented cs1 programs: Concepts and misconceptions, In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pp. 166 –170, ACM, 2007.
16. M. McCracken, W. Newstetter and J. Chastine, Misconceptions of designing: A descriptive study, *SIGCSE Bull*, **31**(3), 1999, pp. 48–51.
17. L. Sudol and C. Jaspan, Analyzing the strength of undergraduate misconceptions about software engineering, In *Proceedings of the Sixth international workshop on Computing education research*, pp. 31–40, ACM, 2010.
18. S. Merriam, Qualitative Research and Case Study Applications in Education, Revised and Expanded from Case Study Research in Education, ERIC, 1998.
19. R. Stake et al. Case studies in science education, volume i: The case reports, 1978.
20. S. Pamplona, N. Medinilla and P. Flores, Exploring misconceptions of operating systems in an online course, In *Proceedings of the 13th Koli Calling International Conference*

on Computing Education Research, Koli Calling '13, pp. 77–86, New York, NY, USA, 2013, ACM.
21. S. Pamplona, N. Medinilla and P. Flores, Assessment for learning: A case study of an online course in operating systems, *International Journal of Engineering Education*, **31**(2), 2015, pp. 541–552.
22. J. Saldaña, *The coding manual for qualitative researchers*, Number 14, Sage, 2012.
23. T. Muhr, ATLAS/ti—A prototype for the support of text interpretation, *Qualitative sociology*, **14**(4), 1991, pp. 349–371.
24. Y. Lincoln and E. Guba, *Naturalistic Inquiry*, Volume 75, Sage, 1985.
25. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen et al. *Object-oriented modeling and design*, Volume 199, Prentice-hall Englewood Cliffs, 1991.
26. W. Haythorn, What is object-oriented design? *JOOP*, **7**(1), 1994, pp. 67–78.
27. B. Liskov, Keynote address-data abstraction and hierarchy, *ACM Sigplan Notices*, **23**(5), 1988, pp. 17–34.
28. M. Monteiro, *On the cognitive foundations of modularity*, 2011.
29. E. Yourdon and A. Armitage, *Análisis estructurado moderno*, Volume 5. Prentice-Hall, 1993.
30. E. Yourdon and L. Constantine, *Structured design: Fundamentals of a discipline of computer program and systems design*, Volume 5, Prentice-Hall Englewood Cliffs, NJ, 1979.
31. T. DeMarco, *Controlling software project*, 1982.
32. E. Berard and M. Twain, Abstraction, encapsulation, and information hiding, *Essays on Object-Oriented Software Engineering*, **1**, 1993.
33. D. Parnas, The secret history of information hiding, In *Software pioneers*, pp. 398–409, Springer, 2002.
34. P. Rogers, Encapsulation is not information hiding, *JavaWorld.com*, **5**(18), 2001, pp. 01.

Pamela Flores is a Ph.D. Professor at Escuela Politécnica Nacional, Ecuador. She obtained her Ph.D. degree in Computer Science from the Universidad Politécnica de Madrid (UPM), Spain. In that period, she was a member of the Decoroso Crespo laboratory in UPM and teaching assistant in undergraduate and graduate subjects related to Software Design. Before this, she got her Master degree in Information Technologies from UPM in 2011, and her Engineering degree in Computer Systems from Escuela Politécnica Nacional del Ecuador in 2005, where she worked as a teaching lecturer for two years. Her area of interest is Computer Science Education, with a particular emphasis on Learning in Software Design.

Nelson Medinilla is a Ph.D. Professor at Universidad Politécnica de Madrid (UPM). He is an Electrical Engineer with a degree from the Universidad de la Habana, with a PhD in Information Technology from the Universidad Politécnica de Madrid. He worked for 20 years as a professor at Instituto Superior Politécnico José Antonio Echeverría in La Habana. For the next 20 years, he taught various subjects related to software design at the Software Engineering Department of the Information Technology Faculty (Universidad Politécnica de Madrid). His areas of research include Software Design and Computer Science Education.

Sonia Pamplona is Ph.D. Professor at Universidad a Distancia de Madrid, UDIMA. She is a Computer Engineer with a Degree from the Universidad Politécnica de Madrid (UPM) and a Ph.D. in Computer Science from Universidad Politécnica de Madrid. She has worked for over 20 years in different areas of Information Technology education. She currently teaches undergraduate courses of Operating Systems and Human-Computer Interaction and a postgraduate course of Mobile Learning. Her area of research is Computer Science Education, with a special interest in online learning.