Using Conceptual Blending to Teach Software Design Principles to Undergraduates

Ashraf Gaffar
Elec. & Comp. Eng. Dept.
Iowa State University
Ames, IA
gaffar@iastate.edu

Mohamed Y. Selim
Elec. & Comp. Eng. Dept.
Iowa State University
Ames, IA
myoussef@iastate.edu

Oliver Eulenstein
Computer Science Department
Iowa State University
Ames, IA
oeulenst@iastate.edu

***Abstract*— The domain of software design is gaining a long overdue recognition as a vital discipline within software engineering, necessitating innovative approaches to its teaching in undergraduate education. Despite the growing importance of software design, academic programs often treat it as a secondary skill, overshadowed by the strong emphasis on coding. Only a few schools offer a design degree or dedicated design courses. This disparity between industry demands and educational practices underscores the need for novel pedagogical strategies. In our innovative work, we discuss a new way of effectively teaching software design-by-analogy for undergraduates to help them rapidly acquire the essential skills needed to design complex software without getting entangled in complex code generation and management. Software design does not necessarily follow the same clear delineation/separation between modules and components naturally apparent in tangible engineering domains. We employ "Conceptual Blending" to help students map their everyday experiences onto software design concepts. The process begins with students analyzing a simple two-arm watch to identify its user interface and create a finite state automaton for its interaction design. Success rates decline as the complexity of the watches increases, underscoring the software design challenges. By comparing these exercises to software interfaces, students learn to apply design techniques such as navigation modeling and prototyping, ensuring they can create intuitive, user-friendly software.**

Keywords—software design; teaching; watch; common sense mapping

I. INTRODUCTION

Software design is emerging as a new discipline within software engineering [1, 2, 3]. The relatively young domain of Computer science/se has originally focused on the challenge of managing software source code, which remains a formidable task[4]. The rapid growth in the number of lines of code in the last few decades and the associated exponential complexity required to manage and construct very large source code mandated a growing focus on developing expertise in coding. Industry also has a growing need in the number of software developers.

The construction phase of a complex product remains a major component, especially in younger domains, and often has a steep learning curve requiring the lion's share of the allocated resources, including time, labor, and special project expertise. In the cases of software engineering, design was often considered a low-priority byproduct and hence dealt with as a secondary add-on to the code. On the contrary, in more established domains like civil and mechanical engineering, a clear separation exists between the construction process and that of design. Design is put first and is given high priority and sufficient resources. Another contributing factor is the intangible nature of software. While other engineering domains have a tangible or measurable nature or both, software is hard to measure or see. The visible source code is not directly conducive to the final UI or behavior. It is safe to say that "Design" is a well-established and recognized practice in those domains, while software engineering is starting to gain recognition [5,6]. This recognition resulted in the design process becoming mature and well-defined. Additionally, design procedures, tools, and techniques are well-researched, scientifically founded, and separated from the construction of designed artifacts.

Software design does not necessarily follow the same delineation and separation of modules on a temporal or any other organizational order.

Recently, software designers and researchers advocated for software design as a new emergent discipline [7] and are working to separate it from the dominant discipline of software construction effectively. The seminal book of software design patterns by the Gang of Four (GOF) [8], considered one of the landmarks in software, is mainly about code design rather than software design. While most software developers use the book, its name has had a negative effect on teaching software design. Most undergraduate and graduate-level software curricula used this book as The Software Design book when it was a perfect coding book with nothing to do with software design. Software design is well defined today, but the name of this seminal book is often considered a misnomer that led to decades-long ambiguity and misunderstanding of what software design should mean. The emergence of a large body of knowledge and books on software design is now helping to clarify the difference [9, 10, 11.]

Many industry jobs were also asking for software engineers whose main responsibility is actually software construction or

development, adding to the confusion and lack of understanding of what "design" really means in software [12, 13.]

Early pioneers in the area of software design, like Alan Cooper, started calling for a distinction between code- and software design, and on top of acknowledging the need for different types of design, he laid out a road map to specific details and activities and new types of software design artifacts that are not code design [14, 15.] Alan Cooper also provided a clear description of different roles needed under the umbrella of software design, so it did not only become a separate discipline, but it could also be further divided into multiple collaborating roles with different design activities.

This early description led to a growing trend in the industry to define new specific disciplines and sub-disciplines in Software Design, especially in larger companies, including Interface Design, Interaction Design, Application Design, and Graphics design, all under software design [16, 17]. This separation guided our work. We focus on Interaction Design, a subtle but complex part of design that is hard to describe but essential to any software design process [19]. Today, intuitive software or software that's easy to use is mainly attributed to good interaction design. One of this paper's authors has worked for several years in Silicon Valley in one of the largest software design enterprises as a Senior Design Expert, leading a group of designers to provide all essential components of software design. We will elaborate on this later.

This recognition also resulted in the emergence of new software design methodologies that exclusively focus on design processes and provide details that are tried, tested, and true [20, 21.] User-centered design, Context-driven design, Domain Driven Design, and Goal Oriented design are just a few examples [14, 16].

Due to its young age, software design is still under-recognized in academia and is not taught the same way it is practiced in industry [22.] Only a few universities offer software design as a dedicated design course, a minor, or a major. Most Universities prefer to focus on programming theories and practices with some focus on software architect. Software design courses often use the GOF "Design Patterns" as explained earlier.

At the university level, only a few schools offer a dedicated software design degree [23.] Some examples are Stanford School of Design of D-school and CMU HCI).

Except for large software companies, the software industry often hires software developers under the name of "software engineers" [24] but hardly asks for any specific software design courses or background [25], and the job is often focused on coding [26.] That begs the question: Is academia deficient in correctly building a new generation of software designers who could lead the way into better defining the discipline of software design in industry or is Industry deficient in correctly recognizing the need for core software designers, hence discouraging academia from investing into building a new generation of designers ready to go to industry as lead designers.

II. CHALLENGES

A. Well-known Challenges

One of the most well-known challenges that -even with interest focused on design, is that there is no clear separation between software design and code design: Brad Meyers' "Why software interfaces are Hard to Design" [27] identified this challenge decades ago. Today, this legitimate concern is well taken care of in most software design methodologies. The User-Centered Design approach provides a top-down approach to identify this grey area of overlap and address it at two orthogonal fronts: the design is separated into a well-defined conceptual design and high-level, which is then elaborated into detailed design to modules and components. The other front is to repeat this progressive approach via multiple iterations with incremental details.

Another major challenge is the obscured and intangible nature of software. Other engineering disciplines are either fully visible, fully measurable, or both. Civil, Mechanical, and Electrical are some examples. There is a clear positive correlation between visibility and understandability. Most design artifacts already refer to tangible objects (floor, beams, struts, elevators, etc.) When it gets less visible, it gets harder to comprehend. The illusive nature of software makes it the prime candidate as hard to comprehend. While the only tangible artifact in software is the source code, the resultant Graphical User Interface (GUI), with its complex behavior and sophisticated look and feel, is hard to directly connect or correlate to the millions of lines of source code producing it.

This correlation is mostly missing by nature. While we can not have a clear measurement of most source code (KLOC, function points, etc.) in the first place, the size is less representative of the actual effort expenditure, or the capabilities of the resultant software. Software profiler tools show us that the actual runtime is very nonlinear in executing the lines of code in many cases. While some lines in some parts of the source code could be run millions of times during a typical execution, other large parts could be rarely, if ever, used at all.

A third challenge is the high dependence of software on logic and behavior rather than on tangible artifacts. Most engineering artefacts are physical or mechanical by nature with mostly direct functionalities. Doors open and close, switches turn on and off, and elevators go up and down. Cars move in the streets and steer with a simple steering wheel. Regarding software, we often have a very large labyrinth of interconnected functions and procedures that can offer millions of combinations of tasks and subtasks. Users are typically expected to find their way across these complex networks hidden under menus and toolbars.

This particular challenge is immense due to the growing number of features and functions in any modern software comprising many steps to complete, which we refer to as "Deep Features". The User-Centered Design provides some solutions to this software design problem that guide designers to design these invisible networks first according to the actual user routines and expectations before projecting them into menus and tool bars. Combined with multiple iterations and validation steps, these approaches often result in software that is more intuitive to use -or to interact with, hence the new role of "Interaction Design."

B. The Concept of Concept

Concepts are important building blocks in interaction design.

In software design, Interaction Design (referred to as IxD in the rest of the paper) is an important design aspect that determines the intuitiveness of deep tasks. In a well-designed interaction, the user will find deep tasks easier and more intuitive to find and complete. We measure the quality of IxD with efficiency (time to complete a complex task) as well as effectiveness (the number of complex tasks completed within a given time.)

When we introduce IxD to undergraduate students, juniors, or seniors, we start with the interaction design the first step of defining an abstract idea, a concept.

A concept is an idea or mental image corresponding to some distinct entity or class of entities or to its essential features or determines the application of a term (especially a predicate) and thus plays a part in the use of reason or language.

A concept is a new idea with insufficient details, somewhere between high-level abstraction and a full description. In IxD, concepts gradually grow from a name (probably emanating from a need or an [innovative] idea, into a detailed description of an artifact and its features. Some SW design methodologies, like UCD, devise multiple tools and techniques to help designers and users work together to effectively and efficiently identify and "grow" a new concept. This lies in the heart of software design in general and in IxD in particular. Though successive meetings and interviews (User Research), designers go through the conceptual design phase, a nice name for designing (creating) a new concept. The needs are often buried deep inside the user's mind as well as within the context (domain knowledge, business, competitors, etc.). These needs are implicit and are often hard to fully and succinctly identify. Designers' role is often to "interrogate" users and "investigate" the domain to be able to describe those needs explicitly and to materialize them into a new artifact (product). It starts as a concept, and it ends as a product. The concept starts as an abstract idea and constantly grows in details and features. Designers use different analysis and consolidation tools (post-it notes in brainstorming sessions, Questionnaires, shadowing, etc.) as well as presentation tools (PowerPoint presentation, sketching tools, prototypes, etc.) to grow and solidify the concept.

III. Examples

Using the watch as an educational model to introduce IXD

One important aspect of a concept is to determine its attributes (physical look: size, color, weight, eyes, tail, etc.) another aspect is its behavior (attitude, temper: aggressive, stubborn, intelligent, etc.)

These two aspects have long been projected into SE as the "Look and Feel". While they are often referenced in many HCI literature, they fall short of showing how to apply them in complex software applications. They need to be investigated more deeply and put into specific steps to help in the design process.

A. Conceptual Mapping and Conceptual Blending

Humans have a continuous perceptual model of the surroundings and often refer new objects to existing familiar ones to build an idea about how the new object might be. This is a transferable experience and is often emphasized in the design of new software. The desktop metaphor is one example to help users envision their computer home screen as similar to their actual desktop. While not identical, humans have the ability to understand the similarities between metaphors while neglecting the differences. This is an important feature of mind approximation, prediction, and cognition that helps the brain recognize incomplete pictures, understand ambiguous or grammatically incorrect sentences, or understand a noisy conversation by mapping it to a close match, which is defined as Conceptual Mapping [28.] In conceptual mapping, we also cognize a new object's behavior by recognizing some of its apparent features and mapping them to a similar object that we already know. This mapping allows us to cognize-via-recognition. We can, hence, predict some of the possible behaviors of the unknown object by mapping them to a known one. This powerful mind feature allows us to rapidly understand and interact with a new object by drawing similarities to the one we're familiar with.

Similarly, Conceptual Blending allows us to predict the relationships and transitions between unknown objects by drawing similarities to known objects. For example, if we know that A□ B&C, then we are asked "what 1", we can predict the possibility that 1, 2, and 3 by blending the transition between the first alphabets with the potential transition between the first three digits. Conceptual blending also allows us to blend two known concepts to produce a new one or to blend concepts of different nature or different domains by drawing parallels between the known and unknown concepts, both in features and behavior as well as transitions [28.]

We use the same brain abilities to teach the fundamentals of IxD, which is a relatively new concept within the HCI domain and is often misunderstood or ignored altogether. In some cases, it is affiliated with the interface design, while in other cases, it is affiliated with the application design, and in both cases, IxD is done implicitly as a by-product to connect the steps of deep tasks. We consider both affiliations incorrect. While these two "extremes" can work well in many cases, they could be special cases of a more general case that we will describe using an example, the Watch.

B. Simple Watch

A simple mechanical watch is shown in Figure 1. In its most simple form, it displays the time in hours and minutes. We consider the following:

- State 0.0: In, idle
- State 0.1: In, turning upward
- State 0.2: in, turning downward
- State 1.0: Out, idle
- State 1.1: Out, turning upward
- State 1.2: Out, turning downward
- Response: @1.1: Turn the Minutes arm clockwise (increase minutes)

- Response: @1.2: Turn the Minute's arm counter-clockwise (decrease minutes)
- Indirect response: Turn the Hours arm
- Multiple states
- Weird response: turn up, arm goes CW, down → CCW
- (A: visible, congeable)
- Indirect response: Moving the Hour arm
- (A: Learnable, transferable)
- Yet, intuitive

Fig. 1. Simple watch interface

Interface Design: The size of the Watch, the shape (round, rectangular…), the shape, size, and color of the two "hands", the color of the inside, the outside, and the band, etc.

Interaction Design: How the user will interact with the Watch. This can be completely ignored if the Watch is fully autonomous, i.e., capable of determining the correct time and time zone at all times. In traditional cases, interaction design would take care of how the user will be able to adjust the time. Following the "de facto standard", the user will need to pull out the watch crown (knob) and turn it upward to rotate the minute's arm to the desired time.

Constraints: While the user can adjust the minute's arm, she/he cannot directly manipulate the hour's arm. This seems normal from our acquired experience with watches, but it brings an important design principle: to look for domain-specific constraints. What kind of constraints do we have here? The exact position of the hour's hand is a function of the position of the minute's arm. Therefore, the user can not be allowed to control both arms because they are dependent independently. If one arm is sufficient, which one should it be?

Theoretically, either one will work, but then comes the accuracy of controlling the minutes indirectly through the control of the hour's hand. There could also be a technical problem of having the minutes arm rotate at a very high speed if the user changes the hours slowly.

Also, we can add one step forward and increase the minutes by turning the watch crown clockwise and decrease the minutes by turning it counterclockwise to improve our design.

TABLE I. SIMPLE WATCH STATE MACHINE TABLE

A simple Watch state machine	
Action	***Response***
0.0	ϕ
0.1	ϕ
0.2	ϕ
1.0	ϕ
1.1	Turn minutes arm CW
1.2	Turn minutes arm CCW
Turn minutes arm CW	Turn hours arm accordingly
Turn minutes arm CW	Turn hours arm accordingly

The last two raws are indirect response and constraints, respectively.

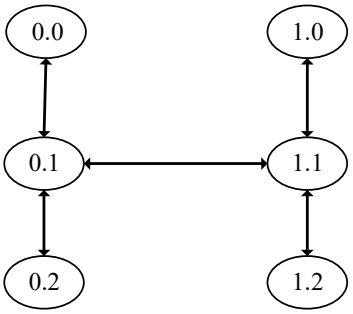

Fig. 2. Simple watch-Moore machine

Figures 2 and 3 show different representations of the finite state machine given to students for validation.

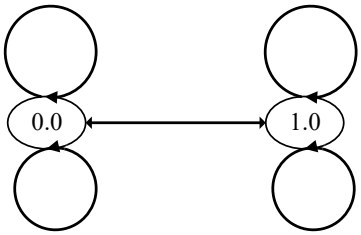

Fig. 3. Simple watch-Mealy machine

C. Advanced Watch

A slightly more complex mechanical watch is shown in Figure 4. It displays the time in hours and minutes like the previous one. Additionally, it displays the day of the month. We can identify the following states:

Fig. 4. Advanced watch interface

- State 0.0: In, idle
- State 0.1: In, turning upward
- State 0.2: in, turning downward

- State 1.0: Out one, idle
- State 1.1: Out one, turning upward
- State 1.2: Out one, turning downward

- State 2.0: Out two, idle
- State 2.1: Out two, turning upward
- State 2.2: Out two, turning downward

- Response: @ 1.2, increment the date
- Response: @2.1, turn the Minutes arm clockwise (increase minutes)
- Response: @2.2, turn the minute's arm counterclockwise (decrease minutes)

- Indirect response: @ 2.1, turn the Hours arm clockwise accordingly
- Indirect response: @2, turn the hour's arm counterclockwise accordingly
- Indirect response: @ 1.0, stop the Seconds arm

Interface Design: The size of the Watch, the shape (round, rectangular…), the shape, size, and color of the two "hands", the color of the inside, the outside, and the band, etc.

Interaction Design: How the user will interact with the Watch. This can be ignored entirely if the Watch is fully autonomous, i.e., capable of determining the correct time, time zone, and date. In traditional cases, interaction design would take care of how the user will be able to adjust the time. Following the "de facto standard", the user will need to pull out the watch crown once and turn it upward or downward to rotate the minute's arm to the desired time and pull out the watch crown twice and turn it upward or downward adjust the date to a desired one.

Constraints: While the user can adjust the minute's arm, she/he cannot directly manipulate the hour's arm. This seems normal from our acquired experience with watches, but it brings an important design principle: to look for domain-specific constraints. What kind of constraints do we have here? The exact position of the hour's hand is a function of the position of the minute's arm. Therefore, the user can not be allowed to independently control both arms because they are dependent. If one arm is sufficient, which one should it be?

Theoretically, either one will work, but then comes the accuracy of controlling the minutes indirectly through the control of the hour's hand. There could also be a technical problem of having the minutes arm rotate at a very high speed if the user changes the hours slowly.

TABLE II. ADVANCED WATCH STATE MACHINE TABLE

Advanced watch state machine	
Action	***Response***
0.0	ϕ
0.1	ϕ
0.2	ϕ
1.0	Stops seconds arm
1.1	ϕ
1.2	Turn minutes arm CW
2.0	Turn minutes arm CCW
2.1	Turn hours arm CW accordingly
2.2	Turn hours arm CCW accordingly
0.0	Start seconds arm

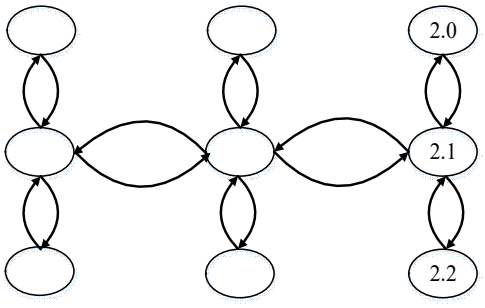

Fig. 5. Simple watch-Moor machine

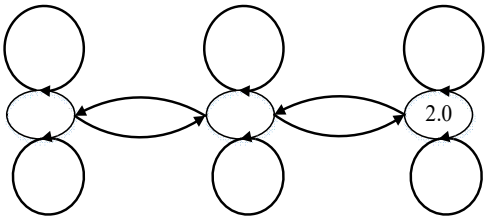

Fig. 6. Simple watch-Mealy machine

TABLE III. ADVANCED WATCH STATE MACHINE TABLE

State/ Action	Advanced watch state machine		
	In (0.0)	*Out1 (1.0)*	*Out2 (2.0)*
Idle	ϕ	ϕ	ϕ
Pull out	Go Out2 Stop sec. arm	Go out2	N/A
Push in	N/A	Go In Start sec. arm	Go out1
Turn CW	ϕ	ϕ	Increment minute arm
Turn CCW	ϕ	Increment date	Decrement minute arm

D Similar to Figures 2 and 3, Figures 5 and 6 illustrate the design of the advanced Watch using principles from Moore and Mealy-type state machines. It is clear from both figures that the FSM is more complex now for both Moore machines.

D. Complex Watch

A complex watch is shown in Figure 7. The equations are an exception to the prescribed specifications of this template. You will need to determine whether or not your equation should be typed using either Times New Roman or the Symbol font (please, no other font). Treating the equation as a graphic may be necessary to create multilevel equations, and inserting it into the text after your paper is styled.

Fig. 7. Complex watch interface

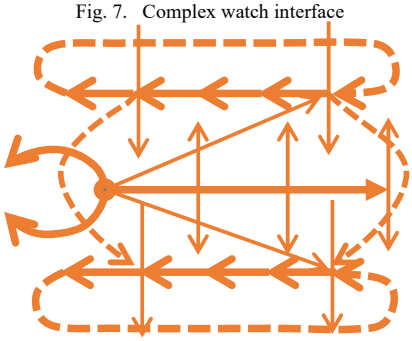

Fig. 8. Part of the complex estimation of the Watch state machine

IV. SOFTWARE INTERACTION EQUIVALENT

This approach leveraged the designer's common sense and prior experience with analog watches to introduce the interaction design concept, highlighting its significant differences from the commonly used user interface (UI). By drawing parallels between a watch's crown and a software's clickable button, students quickly understood that multiple functions could be superimposed on a single control element. For example, just as a watch's crown can be used to set the time, change modes, or start a stopwatch, a software icon can perform different actions depending on the interaction state, such as mouse-over, drag-and-drop, click, double-click, or triple-click as shown in Fig. 9.

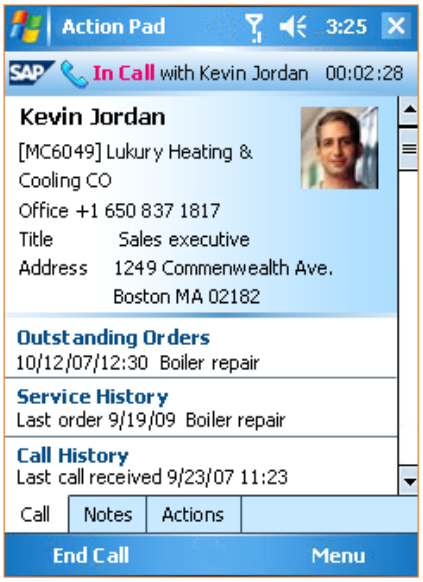

Fig. 9. Example of Action Pad Interface

This method also emphasized the complexity and invisibility of navigation in software interfaces. It taught students to select specific interaction design techniques to enhance usability and improve the overall user experience. By understanding how to map different interactions to a single control element, students learned to create more intuitive and functional UIs.

We extended this approach by introducing the concept of interaction dimensionality. This defines groups of interaction states that are orthogonal to each other, thus expanding the interaction capabilities of a graphical user interface (GUI). For instance, a single icon can support a sequence of actions (e.g., right-click and then select an option from a context menu) or combinations of multiple icons (e.g., using Ctrl+Alt+Del to reset or Ctrl+C to copy). These advanced techniques allow for richer and more versatile user interactions as shown in Fig. 10.

The detailed exploration of interaction dimensionality and its application to complex software systems is reserved for future work. This foundational understanding, however, equips students with the tools to design sophisticated and user-friendly software interfaces.

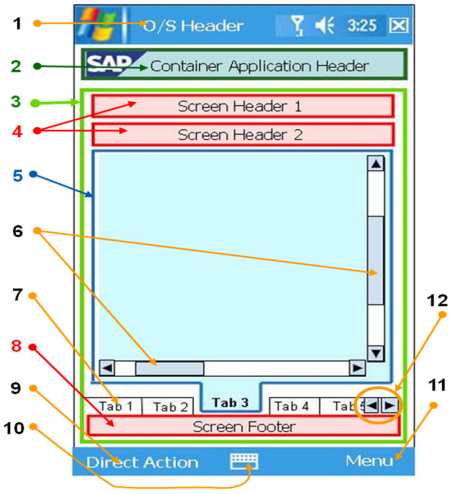

Fig. 10. Example of Interaction Dimensionality.

V. Conclusion

In conclusion, our innovative approach using conceptual blending has shown promise in enhancing undergraduate software design education. By drawing on familiar analogies such as watch mechanisms, we help students grasp complex design concepts more intuitively. While students excel with simpler tasks, increased complexity reveals the need for iterative learning. This method effectively bridges theoretical knowledge and practical application, addressing the unique challenges of software design. Our findings indicate that students initially succeed with high rates on simple tasks but struggle with more complex ones, highlighting the importance of ongoing practice. Comparing watch designs to software UIs helped students better understand user interface and interaction design. Our approach guides students through essential software design techniques, ensuring they create intuitive, user-friendly software.

Future work will refine these methods and explore their application in other areas of software engineering education. Our study underscores the need for innovative teaching methods that align with industry demands, ultimately preparing students for modern software development challenges.

References

[1] M. Shaw, "Prospects for an Engineering Discipline of Software", *IEEE Software*, vol. 7, pp. 15-24, Nov. 1990.

[2] C. H. Lung, G. Zarrabi, A. Kontogiannis, K. Elish, M. Davari, and E. Stroulia, "Teaching Software Design and Architecture through Studio Presentations", *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*, pp. 881-887, 2020.

[3] M. Caspersen and M. Kölling, "Introduction to Software Engineering", *Introductory Programming, Computing, and Software Engineering Education (SIGCSE '19)*, 2019.

[4] D. L. Parnas, "Software Engineering: An Unconsummated Marriage", *Communications of the ACM*, vol. 40, 1997.

[5] J. Bosch, "Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach", *Pearson Education*, 2019.

[6] A. Cooper, "The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity", *Sams - Pearson Education*, 2014.

[7] N. Cross, "Design Thinking: Understanding How Designers Think and Work", *Berg*, 2011.

[8] Gama, Eric, et.al "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.

[9] M. Holm, "Project-based Instruction: A Review of the Literature on Effectiveness in Prekindergarten Through 12th Grade Classrooms", *Insight: Rivier Academic Journal*, vol. 7, no. 2, pp. 1-13, 2011.

[10] S. Bell, "Project-Based Learning for the 21st Century: Skills for the Future", *The Clearing House*, vol. 83, 2010.

[11] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, et al., "Active Learning Increases Student Performance in Science Engineering and Mathematics", *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410-8415, 2014.

[12] M. Petre, "UML in Practice", *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[13] B. Chandrasekaran, "Design problem solving: A task analysis", *AI Magazine*, vol. 11, no. 4, pp. 59-70, 1990.

[14] Y. Rogers, "Interaction design gone wild: striving for wild theory", *Interactions*, vol. 18, no. 4, pp. 58-62, 2011.

[15] D. A. Norman and S. W. Draper, User-centered system design; new perspectives on human-computer interaction, CRC Press, 2013.

[16] S. Houde and C. Hill, "What do prototypes prototype?", *Handbook of Human-Computer Interaction*, vol. 2, pp. 367-381, 1997.

[17] J. Giacomin, "What is human-centered design?", *The Design Journal*, vol. 17, no. 4, pp. 606-623, 2014.

[18] E. B.-N. Sanders and P. J. Stappers, "Probes toolkits and prototypes: three approaches to making in codesigning", *CoDesign*, vol. 10, no. 1, pp. 5-14, 2014.

[19] Cooper, A. et al. About Face 3: The Essentials of Interaction Design, John Wiley & Sons Inc; 3rd edition 2007

[20] T. Fong, I. Nourbakhsh and K. Dautenhahn, "A survey of socially interactive robots", *Robotics and Autonomous Systems*, vol. 42, no. 3–4, pp. 143-166, 2003.

[21] D. Boyd, S. Golder, and G. Lotan, "Tweet tweet retweet: Conversational aspects of retweeting on Twitter", *System Sciences, 43rd Hawaii International Conference*, 2010.

[22] P. Fratzl and R. Weinkamer, "Nature's hierarchical materials", *Progress in Materials Science*, vol. 52, no. 8, pp. 1263-1334, 2007.

[23] N. P. Napier, S. J. Dekleva and X. Liu, "Transitioning to Blended Learning: Understanding Student and Faculty Perceptions", *Journal of Asynchronous Learning Networks*, vol. 16, no. 1, pp. 20-32, 2012.

[24] N. R. Council, How people learn: Brain mind experience and school: Expanded edition, National Academies Press.

[25] C. Snyder, Paper prototyping: The fast and easy way to design and refine user interfaces, Morgan Kaufmann, 2003.

[26] E. Dijkstra, "The Humble Programmer" *Communications of the ACM*, vol. 15, no. 10, pp. 859-866, 2018.

[27] Myers, Brad A, Why Are Human-Computer Interfaces Hard to Design and Implement, Carnegie-Mellon University. Department of Computer Science, 1993.

[28] Fauconnier, Gilles and Turner, Mark, "The Way We Think: Conceptual Blending And The Mind's Hidden Complexities", Basic Books Publishing, 2002.